Bachelor Thesis

Fast Learning of Diverse Robotic Skills

Valentinos Pariza

vpariz01@ucy.ac.cy

University of Cyprus

Department of Computer Science

May 2022

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

Fast Learning of Diverse Robotic Skills

Valentinos Pariza

Supervisors

Professor Chris Christodoulou

Dr. Vassilis Vassiliades

Thesis submitted in partial fulfillment of the requirements for the award of a Bachelor of Science degree in Computer Science at the University of Cyprus

May 2022

Acknowledgements

I want to thank my supervisors, Professor Chris Christodoulou and Dr Vassilis Vassiliades, for the guidance and help they provided me with throughout the whole time working on my Bachelor Thesis. I am grateful for the time and effort they devoted to supporting me to complete it.

Abstract

Quality Diversity Optimization is a recent field that has shown promising results in generating a diversity of high-performing quality robotic skills [2]. However, QD Algorithms require a lot of execution time due to the number of solution evaluations needed to achieve good results. Work done so far focused on executing these algorithms mainly on CPUs, which could take days. Thus, a good question that arises first is can we speed up the execution of these algorithms? This question was recently touched on by Lim, Allard, Grillotti and Cully [69], who tried to test the MAP-Elites and a simple variation on GPUs for robotic simulation problems. They showed that speeding QD Algorithms is possible on GPUs. Another recent study introduced the Differentiable Quality Diversity (DQD) [4], and it showed that Differentiability in QD Algorithms could provide better results in the same number of iterations. This work aims to extend the work on the two aforementioned papers to make a solid start on whether QD and DQD Algorithms implemented on GPUs can help speed up the generations of solutions for QD problems. We attempt to investigate whether QD and DQD algorithms are affected by the increase of their parallelization (i.e. batch size) on GPUs, whether there is a speedup in wall-clock time in the execution of QD and DQD Algorithms on GPU compared to CPU and last whether Differentiable QD algorithms on GPUs can help get better results and faster.

Contents

Chapter 1
Introduction

1.1 Quality-Diversity Optimisation

Quality Diversity (QD) Optimization is a recent field of Evolutionary algorithms that have shown promising results in generating a diversity of high-performing quality robotic skills [2]. Current algorithms such as MAP-Elites and variations were able to illuminate the relationship between the performance of each solution found and the diversity of the [1]. However, QD Algorithms require a lot of execution time due to the massive number of solution evaluations that ought to be executed. The work done on this branch of Evolutionary Algorithms focused on running these algorithms mainly on computer CPUs and scaling them to many Clusters' CPUs. But, QD Algorithms, like many other algorithms, depend on operations, like matrix and vector manipulation, that are highly parallelizable. This nature of the QD Algorithms' core makes it attractive to use specialized hardware, specifically GPU, to improve the performance of the QD Algorithms and specifically the runtime.

1.2 The Importance of Accelerating Quality-Diversity for Robotics

As mentioned in section 1.1, GPUs can help accelerate QD Algorithms by executing faster parallelizable operations. To be more specific, in general, QD Algorithms try to find high-performing solutions that vary across some features. These algorithms consist of four parts (1) selection of solutions from a collection, (2) variation of the selected solutions, (3) evaluation of the performance of the solutions and (4) addition of the solutions to an archive of solutions. But the most time demanding part is the evaluation step, where the solutions are evaluated in terms of performance and some diversity attributes (features are chosen as points of variation in the solutions). The evaluation functions define the problem that the QD Algorithm tries to solve. Those functions could be simple, but they can also include complex operations, like simulations. For example, we could have a problem that provides as a solution the parameters of a policy (e.g., parameters and weights of a neural network) for a robot's movements and as evaluation functions, a function that says how well the robot walks at a specific direction using a given policy and another function that is used for estimating how much time each robot's leg touched the ground on average (i.e. the latter function is the function that defines the variation of the) with the same policy. But, evaluating the policy in terms of those criteria (i.e., estimating the value of those functions) includes having the robot use that policy in either the real world or a simulation to see how it performs. Evaluations for such tasks are initially done via simulations, which are cheaper and faster than real-world evaluations. But physics simulations include complex and time expensive computations that require a significant amount of time when many of them are needed to be executed for the same problem. Getting good results on QD Algorithms on modern CPUs for complex tasks like teaching a robot to walk in many directions can take days. This need to make QD Algorithms execute faster is the source of motivation for our work. Our work primarily investigates the potential acceleration of running QD Algorithms on GPUs instead on CPUs as well as whether differentiability in QD (Differentiable Quality Diversity Optimization - DQD) can play a role in allowing better and faster exploration of solutions to be achieved by QD Algorithms on GPUs.

1.3 Previous Quality-Diversity Research related to our work

Our work depends on the foundations of all the QD work that has been done so far. But we primarily rely on a couple of recent results. More specifically, a recent study from Fontaine and Nikolaidis [4] introduced the Differentiable Quality Diversity (DQD) DQD uses the first derivative of the evaluation functions concerning the solution to guide the exploration of diverse high-performing solutions, and it showed that it is a promising approach for exploring more varied and higher performing solutions. More specifically, they showed that the Differentiable versions of some QD algorithms perform better than their original QD Algorithms regarding the number of diverse solutions they return and the quality of those solutions. But the tests were made on CPUs and for a minimal number of batch sizes ranging between 32 and 100. A more recent work from Lim, Allard, Grillotti and Cully [69] tested for the first time the performance of the one of the simplest QD Algorithms; MAP-Elites (line), on GPUs. More specifically, they tested those two QD Algorithms on different simulation tasks on the BRAX simulator. They observed that the performance of MAP-Elites (line) on those simulation problems was not statistically affected by increasing the batch size they used. They also showed that those two QD Algorithms being executed with more significant batch sizes finish earlier than when using smaller batch sizes. The QD Algorithms being executed on GPUs finish quicker than those on CPUs. Those observations showed the potential of exploiting GPUs for running MAP-Elites and MAP-Elites (line) faster. But their work is limited to only a minimal subset of the state-of-the-art QD and DQD Algorithms, which raises the question of whether those results and observations apply to the other QD and DQD algorithms.

1.4 Objectives

Our work examines the following three goals: (a) Effect of batch size on the Performance of QD and DQD Algorithms, (b) Runtime of QD and DQD Algorithms on GPU compared to CPU and (c) Performance of DQD Algorithms vs QD Algorithms on GPUs. In the first goal, we want to examine whether the algorithms are affected by increasing the batch size (the in-algorithm parallel manipulation of solutions) of QD Algorithms. In the second one, we want to examine the potential runtime improvement of QD and DQD Algorithms on GPUs vs CPUs. Lastly, in the third goal, we want to see whether Differentiable QD can help get better results faster on GPU and thus save some execution time. We structure this document based on those goals, and we start with a comprehensive but concise background revision of QD and associated work (Chapter 2). We move on to discuss the Domains (i.e. Problems) that we use for our experiments (Chapter 3); later in Chapter 4, we discuss the design and implementation of frameworks and QD Algorithms that we developed for our experiments. We discuss the experiments we performed in Chapter 5, and in chapter 6, we discuss our conclusion, some lessons learned and future work.

Chapter 2

Background

2.1 Mathematical Background

2.1.1 Function

A function f is a process that associates an object of a set X (called Domain of the function) to a single object of a set Y (called Subdomain of the function). This correspondence can be written as $f: X \rightarrow Y$.

2.1.2 Mathematical Optimization

Given a real-valued function f(x), an optimization problem is the identification of a solution x belonging to the function's domain that minimises (in which case is called a Minimisation Problem) or maximises (in which case is called a Maximisation Problem) the value of that function.

More specifically, given a function, $f: \mathbb{R} \rightarrow \mathbb{R}$, from real numbers to the real numbers, we seek a solution $x_0 \in \mathbb{R}$ $(of\ the\ Domain)$, such that:

- In case of minimization, we want $f(x_0) \leq f(x)$ for all $x \in \mathbb{R}$.
- In the case of maximization, we want $f(x_0) \geq f(x)$ for all $x \in \mathbb{R}$.

2.1.3 Finite Difference Method (FDM)

The Finite Difference Method (FDM), or derivative of a function that estimates the derivative of a function. Given a first-order differentiable function f, its derivative at a point x is:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

2.1.4 Chain Rule in Derivatives

The chain rule is a formula that specifies how to compute the derivatives of composites functions. A composite function is a function that receives as its domain values the values of the subdomain of a second value. A composite function is written as:

$$f(x) = h(g(x))$$

The derivative of the composite function is specified by the chain rule below:

$$\frac{df}{dx} = \frac{dh}{dg} \cdot \frac{dg}{dx}$$

Or differently written as:

$$f(x) = h'(g(x)) \cdot g'(x)$$

2.1.5 Gradient

The gradient of a first-order differentiable function $f: \mathbb{R}^k \to \mathbb{R}$ is the vector field ∇f (∇ is the Del, or nabla symbol denoting the vector differential operator) in which each point x_i is the first-order partial derivative of the function concerning the ith value of a vector that the function receives as input. That is, the gradient of the function f is a function $\nabla f: \mathbb{R}^k \to \mathbb{R}^k$ defined as:

$$\nabla f(x) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1}(x) \\ \vdots \\ \frac{\partial f}{\partial x_k}(x) \end{bmatrix}$$

2.1.6 Jacobian Matrix

The Jacobian Matrix is the matrix of all first-order partial derivatives of a vector function. Given a vector function $f: \mathbb{R}^k \to \mathbb{R}^m$, whose first-order partial derivatives exist on $\mathbb{R}^k$, then the Jacobian matrix J of f, is the $m \times k$ matrix whose entry in row i and column j equals $J_{ij} = \frac{\partial f_i}{\partial x_j}$ (a partial derivative of f function's jth input with respect to the function's ith output).

$$J_f = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_k} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_k} \end{bmatrix}$$

2.1.7 Tanh Activation Function

The hyperbolic tangent activation function, also referred as Tanh (or TanH or tanh) is a function that takes any real value as input and outputs values in the range (-1,1). That is, tanh function $f: \mathbb{R} \to (-1,1)$ is defined as:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

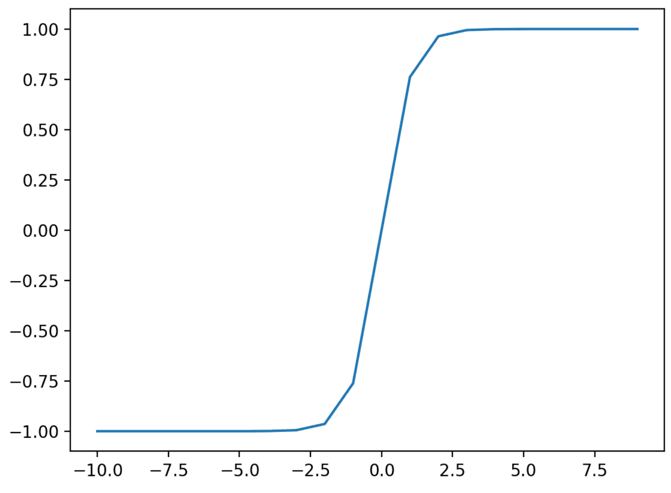


Figure 2.1. Hyoerbolic Tangent Activation Function.

2.1.8 Maximum

A local maximum value of a continuous function $f: A \rightarrow B$ is defined as the value x ∈ A such that for each point x_l, $a \leq\ x_l < x$ it applies that $f(x_l) \leq f(x)$ and for each point x_r $x < x_r \leq b$, it applies $f(x_r) \leq f(x)$ for some a ∈ A and b ∈ A. If the $f(x_i) < f(x)$, for any x_i ∈ A, then the x is a global maximum.

2.1.9 Minimum

A local minimum value of a continuous function $f: A \rightarrow B$ is defined as the value x ∈ A such that for each point x_l, $a \leq\ x_l < x$ it applies that $f(x) \leq f(x_l)$ and for each point x_r, $x < x_r \leq b$ it applies $f(x) \leq f(x_r)$ for some a ∈ A and b ∈ A. If the $f(x_i) > f(x)$, for any x_i ∈ A, then the x is a global minimum.

2.1.10 Mean

Given a population of n real number $x_1, \dots, x_n$ the mean or average μ is defined as:

$$\mu = \frac{\sum_{i=1}^{n} x_i}{n}$$

2.1.11 Standard Deviation

Given a population of n real number $x_1, \dots, x_n$ Standard Deviation σ is defined as:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \mu)^2}{n}}$$

where μ is the population mean.

2.1.12 Median

Given a population of n real number $x_1, \dots, x_n$, where n is an odd number, then the median is the number x_i from the population that has $floor\left(\frac{n}{2}\right)$ elements of the population that have a value lower than it and $floor\left(\frac{n}{2}\right)$ that has a value greater than it, where $floor\left(\frac{n}{2}\right)$ is the integer part of the division of n over 2. If n is an even number, then the mean can be the x_i from the population that has either $\frac{n}{2} - 1$ population elements on its left (with lower values) and $\frac{n}{2}$ population elements on its right (with greater value) or $\frac{n}{2}$ on its left and $\frac{n}{2} - 1$ on its right.

2.1.13 Median Absolute Deviation (MAD)

Median Absolute Deviation of a population of real numbers $x_1, \dots, x_n$ is the median of the absolute deviations from the population's median m (median of population). That is:

$$MAD = median(|x_i - m|)$$

2.1.14 Dot Product

The dot product between two vectors v_1 and v_2 both of them of size n are defined as:

$$dot = v_1 \cdot v_2 = \sum_{i=1}^{n} v_1^{(i)} \cdot v_2^{(i)}$$

where $v_j^{(i)}$ represents the ith element of the vector j.

2.1.15 Rastrigin Function

Rastrigin function is defined by the function f(x) defined as follows:

$$f(x) = 10d + \sum_{i=1}^{d} [x_i^2 - 10\cos(2\pi x_i)]$$

where d is the number of dimensions of the problem.The function has many local minima that make it high multimodal, and those minima are distributed regularly.

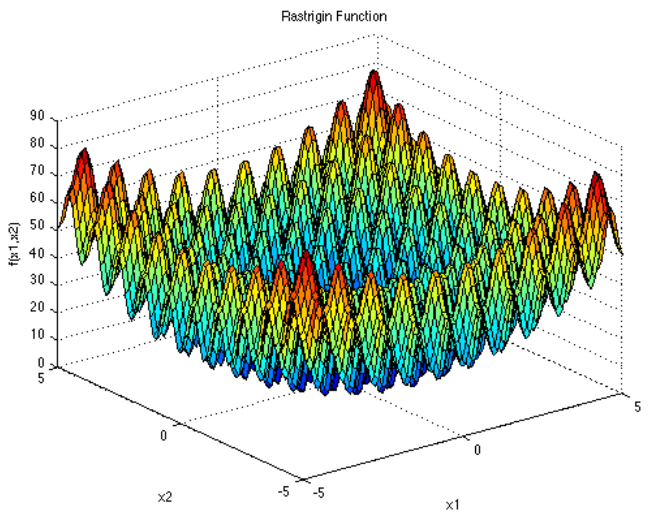


Figure 2.2. Two-dimensional (d = 2) Rastrigin Function with x1, x2 its parameters, from https://www.sfu.ca/~ssurjano/rastr.html

The function is usually evaluated for values of its parameters $x_i \in [-5.12, 5.12]$ for all i = 1, 2, …, d. The global minimum is located at x = 0, where 0 is the zero vector of length d with all its elements equal to zero.

2.2 Artificial Neural Networks

2.2.1 Regression

Regression is a technique for learning the relationship between a set of independent variables or features and a dependent variable or outcome (in the case of many dependent outcomes, we can frame as many independent regression problems as the dependent outcomes are, each of them concerned with a single dependent outcome). In Machine Learning, it is used as a predictive type of model, in which an algorithm is used to predict continuous outcomes.

The simplest form of Regression is Linear Regression, where the relationship between the independent variables and the outcome is linear. In the general case, the linear model we have is the following:

$$y = a_1 \cdot x_1 + \ a_2 \cdot x_2 + \cdots + a_n \cdot x_n + b$$

And the relationship we want to learn is defined by the $a_1, a_2, \dots, a_n$ parameters of the equation above.

2.2.2 Cost/Loss Function

The cost function or loss function is a function used to learn the relationship between the output and the input for a function f. More specifically, given an input x and an output of a function f, f(x), the cost function can tell us how far away the value/solution of a different function g on x is with respect to f(x), That is the difference between g(x) and f(x).

2.2.3 Supervised Learning

In a supervised learning context, an algorithm tries to learn a function f such that for any input x defined for the function f to generate f(x). For an algorithm to learn that function f, it is provided with a set of input values and the expected output values (these are called training datasets). The algorithm uses both the inputs and the expected outputs to learn the relationship between them.

2.2.4 Artificial Neuron

An Artificial Neuron in Machine Learning is considered the smallest and most basic computational unit of an Artificial Neural Network (ANN). They are inspired by the biological neurons and are simply mathematical functions taking some inputs and giving an output.

The most well known artificial neuron is the McCulloch & Pitts (MCP) neuron which receives a set of input signals and outputs a single output. The output is just the application of a function (called activation function) on the weighted sum of its input signals. That is, if X is a vector of the input signals of size n and W is a list of weights for those input signals, then the output of the McCulloch & Pitts is:

$$act_f(X \cdot W) = act_f(\sum_{i=1}^{n} X_i W_i)$$

Where $X \cdot W$ is the dot product between the two vectors and act_f is an activation function.

2.2.5 Artificial Neural Network (ANN)

Artificial Neural Networks (ANNs) or Neural Networks (NNs) are networks of artificial neurons that mimic on a high-level the biological neural networks of animals' and humans' brains. In reality, ANNs are complex functions that have a set of inputs and outputs. It is usually graphically represented as a graph with nodes and edges. The nodes are artificial neurons, and the edges are considered synapses used to share information between neurons. An artificial neuron; in the network; receives a signal, processes it and passes the signal further to potentially another neuron.

The typical structure of an ANN is a layered network (=graph) of nodes constituting a layer of input nodes that receive input signals, a layer of output nodes that output signals and many layers of nodes, called hidden layers that just perform the core processing of network signals.

2.2.6 Multi-Layer Perceptron

2.2.6.1 Introduction to MLPs

Multi-Layer Perceptron (MLP) or also Vanilla" Neural Networks. is a type of a feed-forward ANN. Feed-forward means that the processing flow of the input signals follows a single direction from the input layer to the output layer and not the other way. MLPs consist of at least three layers of nodes; an input layer, one or two hidden layers, and an output layer. The input nodes are simple nodes that just forward the signal to the next hidden layer, whereas the hidden layers and the output layer contain McCulloch & Pitts artificial neurons.

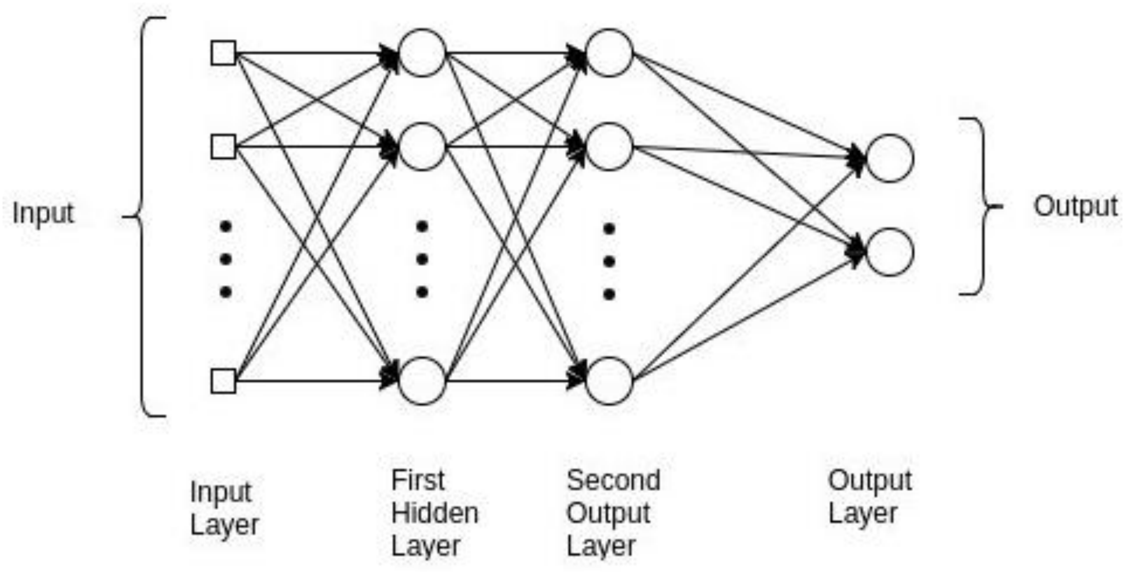


Figure 2.3. Multi-Layer Perceptron with three layers with McCulloch & Pitts neurons (two hidden and an output layer). Note that the input layer does not contain any neurons.

2.2.6.2 Architecture

The architecture of an MLP is defined as the number of hidden layers being used, the number of nodes in each layer, and whether it is a fully-connected or sparse network. In a fully connected network, each node of a layer is connected with each node o the next layer (except the output layer that does not have any further layers in front of it). An example of that can be seen in the image above of a fully connected MLP. A sparse network is one where some of those connections that exist in the fully-connected network do not exist. Note that each edge is weighted by a specific number. The architecture of an ANN and its weights define the function that the ANN approximates.

Because an MLP depends on its architecture to define a function, different architectures can be restricted to a different class of functions. For instance, in the classification problem where we try to categorize/classify an input point to a group of specific characteristics, a single layer MLP (only output layer) is capable of creating a linear function (first entry in the figure below) that simply can separate two different classes in their input space by a line. Two-layer MLP (1 hidden & 1 output layer) is capable of creating a function that separates input points by two an open convex or closed region (second entry below). Three-layer MLP (2 hidden & 1 output layer) is capable of creating a function that separates input points, using arbitrary complex shapes (third entry below) that are capable of separating any classes. Thus, based on Kolmogorov Theorem, no more than three layers are needed in an MLP network. That is, they can approximate any function with at most three layers. Take into consideration that this theorem also depends on the use of non-linear activation function in each neuron.

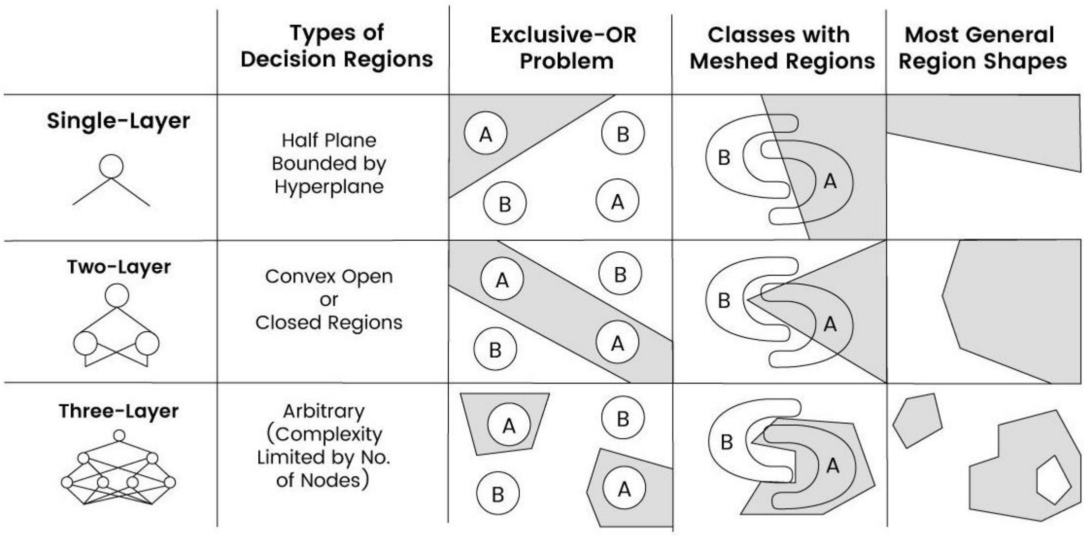

Types of Decision Regions
Exclusive-OR Problem
Classes with Meshed Regions
Most General Region Shapes
Single-Layer
Half Plane Bounded by Hyperplane
Two-Layer
Convex Open or Closed Regions
Three-Layer
Arbitrary (Complexity Limited by No. of Nodes)
A
B

Figure 2.4. Three types of Multi-Layer Perceptron (MLP) as distinguished by the different types of Decision Regions they create. **Single-Layer or Perceptron:** Solves only Linearly Separable problems. Creates Decision Lines/Planes/Hyperplanes. **Two-Layer:** Solves Non-linearly Separable problems. Constructs Convex Regions**.** **Three-Layer:** Separates any classes. Any Arbitrary Convex Region. <https://www.verypossible.com/insights/machine-learning-algorithms-what-is-a-neural-network>

2.2.6.3 Activation Function

Activation functions are usually used by artificial neurons of ANN, and they are functions applied to the inputs of a neuron. There are different activation functions, including the Heaviside Step Function (outputs one if the inputs are greater than zero; otherwise 0), the Sigmoid/Logistic (a smoother differentiable version of the Heaviside Step Function), Tanh (described in section 2.1.7), etc.

2.2.6.4 Forward Propagation

In forward propagation, the input data received by a network's input layer are propagated forward through the network until the output layer, where the latter's outputs are the output of the network. That is, it is a process of an ANN to take a set of inputs and generate a set of outputs. This process is defined from left to right, taking layer by layer and doing the following: For each layer, take the output of each node as the application of an activation function to the weighted sum of the input signals to the node. The output of each node in a layer is the set of input signals for the next layer. The output signals of the output layer are the outputs of the ANN.

2.2.6.5 How an Artificial Neural Network learns to approximate a function

An Artificial Neural Network has an architecture and a set of weights, one for each edge connecting two network nodes. The weights of an ANN are adjustable, and they are responsible for weighting the signals going from one layer to another. For a given ANN architecture, they define the function that the ANN represents. A specific set of values of weights approximates a different function than a different set of values for the weights.

The most well known and used way for learning those weights is the back-propagation method, where for a given initial set of weights for an ANN (usually random), the output values of the ANN for a given set of input values are evaluated on how well it approximates the expected value of the function the ANN tries to match and the error between the ANN's output and the function's desired output is propagated backwards (from the output to the input layer) as feedback to adjust the weights of the network.

But, there are also other ways more straightforward but potentially less efficient in defining those weights. For example, we could randomly choose those weights until those weights lead the ANN to approximate the function; that is, the ANN generates approximately the same outputs with the function it tries to learn for the same inputs. The issue, though is that the more the weights of the ANN and the greater the set of values those weights can take, the more difficult it is to find those weights randomly.

Quality Diversity which we will discuss later (section 2.5), is another way of finding those weights in a more guided way than just randomly choosing those weights.

2.3 Optimisers Background

2.3.1 Optimisers

Optimisers are used in different fields with the sole purpose of helping improve a set of coefficients so that to optimise (maximise or minimise) a specific function. For example, an optimizer can be used in a neural network to adapt the neural network's weights to minimize the difference between a set of expected outputs and the actual outputs of a neural network. Below, you can see some of them.

2.3.2 Gradient Descent (GD)

Gradient Descent (GD) was introduced by Rumelhart et al. [51] and is the most straightforward optimization algorithm for minimizing first-order differentiable functions. The idea of the gradient descent is that since the gradient of a function always points toward the maximum point, then by moving the point of the function where the gradient was derived, in a direction opposite; that is, negative; to the direction of the gradient, then the point derived will most likely (likely because it depends on the step and on whether the point is already a local minimum) give a value for the function lower than before. That is, it will approach a local minimum.

2.3.2.1 GD Mathematical Definition

Consider a first-order differentiable vector function f whose minimum value we seek for. If the derivative of the function with respect to its vector input x is:

$$\nabla f(x) = \langle \frac{\delta f}{\delta x_1}, \frac{\delta f}{\delta x_2}, \dots, \frac{\delta f}{\delta x_n} \rangle$$

Then the GD algorithm amends each value x_i of the input vector x using the derivative of the function's output with respect to the ith value of the vector ($\nabla f(x_i)$) as follows:

$$x_i{}' = x_i - a\nabla f(x_i)$$

Where a is a real number between 0 and 1 chosen to represent the step of the change on the vector's value.

2.3.3 Momentum

The momentum is an extension of gradient descent that, at each iteration, replaces the current gradient with a "momentum", m which is an aggregate of gradients. This aggregate is an exponential moving average of past gradients, including the current one up to time t (see below). The update process of solutions θ becomes:

$$\theta_{t+1} = \theta_t - a \cdot m_t$$

where

$$m_t = \beta \cdot m_{t-1} + (1 - \beta) \cdot \frac{\partial f}{\partial \theta_t}$$

A common value for the hyperparameter β is 0.9.

2.3.4 Adaptive Moment Estimation (Adam)

Adaptive Moment Estimation (Adam) is a stochastic gradient-based optimizer that was proposed by Kingma and Ba [51].

Adam stores an exponentially decaying average of past squared gradients v (like Adadelta [52] and RMSprop [53]) and also kept an exponentially decaying average of past gradients m, similar to momentum. Their update process is:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) {g_t}^2$$

The parameter m is the first moment, just like the momentum that records the past normalized gradient. It is initialized to a vector of zeros.

The v is the second moment, as introduced in Adaptive Gradient Descent & RMSprop. It is initialized to a vector of zeros.

The parameters β_1 and β_2 control the decay rates of the exponential moving averages gradient (initialized AS β_1= 0.9, β_2= 0.999).

The updates of m and v moments based on the equations above make them biased towards 0, especially during the initial steps after they start from 0 values. To counteract these biases, ADAM uses bias-corrected first and second-moment estimates, which extend the moment updates from above with the following two updates:

$$\widehat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Last, the bias-fixed $\widehat{m}_t$ and $\hat{v}_t$ moments are used to update the given parameters θ as follows:

$$\theta_t = \theta_{t-1} - a \cdot \frac{\widehat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon}$$

Where ε is proposed by its authors to be equal to 10^{-8} and a = 0.001.

```
**Require:** $\alpha$: Stepsize
**Require:** $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
**Require:** $f(\theta)$: Stochastic objective function with parameters $\theta$
**Require:** $\theta_0$: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  **while** $\theta_t$ not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep $t$)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t/(1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t/(1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t/(\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  **end while**
  **return** $\theta_t$ (Resulting parameters)
```

Figure 2.5. Adam optimizer algorithm pseudocode for stochastic optimization from Kingma and Ba[50]

2.4 Evolutionary Algorithms & Strategies

2.4.1 Introduction to Evolutionary Algorithms & Strategies

Evolutionary algorithms are algorithms that use evolutionary computation inspired by nature to solve different problems, usually Optimization problems More specifically, evolutionary algorithms simulate the process of natural selection and are an effective tool for discovering high-performing reinforcement-learning policies.

Evolution strategies (ES) are a family of evolutionary algorithms that specialize in optimizing continuous spaces by sampling a generation (a population of solutions) and gradually moving the population toward areas of highest fitness.

2.4.2 (μ/μ, λ)-ES

One canonical type of Evolutionary Strategy is the (μ/μ, λ) Evolutionary Strategy abbreviated as (μ/μ, λ)-ES, where a population of λ sample solutions is created, and then from the population, the μ most high-performing solutions are selected to generate new samples in the next generation. The speciality of the (μ/μ, λ)-ES is that it recombines the μ best-sampled solutions through weighted average into one mean that describes the centre of the population distribution of the next generation. Thus, in a few words, the μ best-sampled solutions will create the centre of distribution (mean value) for the sampled solutions of the next iteration. This can be seen as moving the focus of exploration (centre of distribution) in the solution space towards areas with high-performing solutions, always based on the solutions that have already been discovered.

2.4.3 CMA-ES

The Covariance Matrix Adaptation Evolution Strategy (CMA-ES) is a particular type of the (μ/μ, λ)-ES. It is considered one of the most competitive derivative-free optimizers for single-objective optimization of continuous spaces [34]. CMA-ES models the sampling distribution of the population as a multivariate normal distribution N(m, C), where m is the distribution mean and C is its covariance matrix. There are two main steps in CMA-ES: the selection and ranking of the μ most high-performing solutions, which update the next generation's sampling distribution, N (m, C). Last, CMA-ES maintains a history of aggregate changes to m called an evolution path, which helps search for solutions (similar to how momentum works in stochastic gradient descent). For a more in-depth explanation of the steps performed in CMA-ES, refer to the Hansen tutorial in CMA-ES [35].

2.5 Quality-Diversity Background

2.5.1 Introduction to Quality-Diversity

Quality Diversity (QD) Optimisation (or illumination) is a recent branch of Evolutionary algorithms whose primary goal is to generate a large set of diverse solutions that satisfy two attributes: (a) All solutions produced are high-performing solutions (High Performing Solutions), and (b) the solutions created differ concerning a set of specific characteristics (Solutions Diversity). The assumption taken by QD algorithms is that solving those two problems together is likely to be faster than by independent constrained optimizations.

Based on the definition above, the overall performance of a QD algorithm is defined by scores on the quality of the produced collection of solutions according to:

1) The coverage of the feature space (across which the diversity of the solutions is set).
2) The uniformity of the feature space coverage.
3) The performance of the solution found for each type of solution from feature space.

QD algorithms are potent exploration algorithms that seem to efficiently solve sparse-reward hard-exploration tasks in robotics [54]. In addition to that, Quality-Diversity (QD) algorithms have recently shown to be an auspicious and valuable tool in the field of robotics[55, 56], where these algorithms can be used to generate a repertoire of diverse and high-performing robotic skills, together with by solving a single instance of the QD problem. This repertoire can then be used for rapid adaptation to unknown mechanical damage [57, 3, 58] and coupled with planning algorithms to perform long-horizon tasks [57].

2.5.2 QD Problem Definition

Quality Diversity considers an objective function $f : R^n \to R$ in a continuous n-dimensional space R^nAnd k behavioural functions $m_i: R^n \to R$, or altogether a function $m: R^n \to R^k$. Then considering that $m(R^n) = B$ is the behavioural space with the behavioural function's values; QD's goal is to find a solution $\mathbf{x} \in R^n$ for every $b \in$ B such that $m(\mathbf{x}) = \boldsymbol{b}$ and $f(\mathbf{x})$ are maximized.

The objective function f evaluates a solution (described by n parameters) for a specific problem and returns a value (named objective value) denoting how good the solution is for the particular problem. For example, in the scenario of teaching a robot to walk forward, a solution could be a policy on the robot's low-level movements in each situation. The objective function could be the distance the robot covers forward with a specific solution. The k behavioural functions or the combined single behavioural function give a collection of k values (or a vector of k values) for k characteristics of a solution and is called a behavioural or feature descriptor. In the example of the robot moving forward, the behavioural descriptor could be the energy consumed with the specific solution, the final state of the robot at the end, how much time each robot's leg touches the ground on average etc. Sometimes, the objective function and behaviour function can be seen as part of the same function, which is usually called an evaluation function.

From the Evolutionary Algorithms' jargon, a QD solution is called an organism, phenotype, or individual. A genome or genotype describes a solution, and it is the representation of the solution that is used in an algorithm to generate other solutions. The actions performed by the organism (=solution) are the organism's behaviour, and they create what we defined previously as the behavioural descriptor. The performance of a QD Algorithm is called fitness, and the expression, simulation or function that gives the fitness value is called fitness function.

2.5.3 QD and Illumination

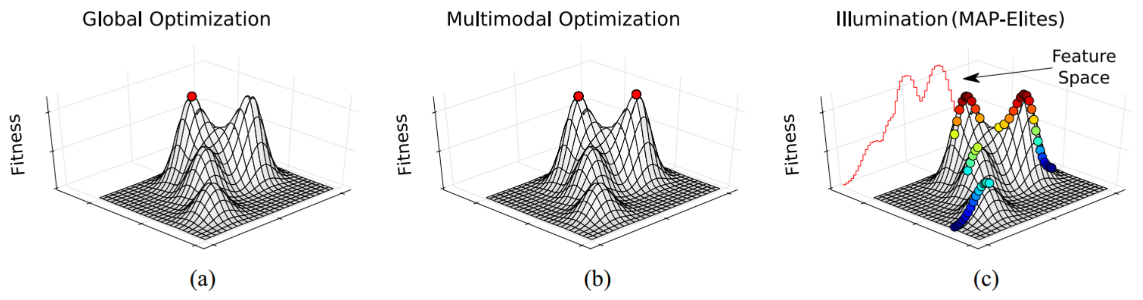


Fig 2.6. Difference between Global, Multimodal and QD (Illumination) Optimisation from Chatzilygeroudis, Cully, Vassiliades, and Mouret [2]

With the introduction of the Multi-dimensional Archive of Phenotypic Elites (MAP-Elites) algorithm, the concept of illumination, which was initially introduced in the field of evolutionary robotics (to encourage diversity in space), was introduced in QD, which made clearer the difference between Quality Diversity Optimisation and other Optimisation approaches. More specifically, MAP-Elites was the first algorithm in QD to be considered an "illumination algorithm", a type of an algorithm that illuminates the fitness potential of each area of the feature space and takes into consideration tradeoffs between performance and the features of the solutions that we are interested in.

The concept of the Illumination in QD draws a harder line between what QD aims to do and what Global Optimisation and Multimodal Optimisation try to do. More specifically, Algorithms for global optimization aim to find a single global optimum of the underlying parameter space in interest. Multimodal optimization (MMO) algorithms can be considered an extension of Global Optimisation that aims at finding multiple optima of the parameter space. In contrast to those two approaches, Illumination algorithms, such as MAP-Elites, aim at discovering significantly more solutions than Multimodal Optimisation, where each solution is the elite of some local neighbourhood defined in some feature space of interest. This process is like “illuminating” the feature space with the highest-performing solutions, thus called Illumination.

2.5.4 Early QD Algorithms

2.5.4.1 Random Sampling

The most basic approach used by different fields of Sciences is Random Sampling, where you randomly choose from the solution space solutions to evaluate using the objective and behavioural functions. It is mainly used as a benchmark to compare it against new algorithms in different fields.

2.5.4.2 Novelty Search + Local Competition (NS+LC)

There are many ways to solve the QD problem, even using approaches from other Optimisation sectors, like Multimodal Optimisation and Multi-Task Optimisation, but there are a lot of specialities in the QD Problem that encourage new approaches. For example, finding the most high-performing solutions while preserving diversity in the set of the selected solutions. Regarding this matter in 2011, Lehman and Stanley [5] said that it is not fair to allow any solution to compete against any other solution based on the same performance objective for a position in the set of the selected ones. This appears because each solution may be associated with different interesting features (= different behavioural descriptors) and thus may be constrained to different values of a performance objective. It is like comparing a leopard with a turtle for speed. But the point here is that we are not interested only in the speed (=objective function) but in other characteristics of them as well so that to encourage diversity.

Lehman and Stanley proposed the algorithm Novelty Search + Local Competition (NS+LC) [5] to promote diversity in the feature space (=behavioural space) but at the same time have each organism (=solution) compete on performance only with other organisms that are close to it in the feature space. NS+LC uses a multi-objective algorithm to accomplish two objectives: (1) maximise an organism’s performance with respect to its closest 15 neighbours in the feature space, and (2) maximise a novelty objective that encourages diversity of organisms selected by rewarding organisms the further they are in feature space from their 15 closest neighbours. Maximizing an organism's performance has two phases, one for local competition concerning an organism's neighbours and a second for comparing the relative scores of organisms with their neighbours globally (=with all the other organisms), which is considered a global competition.

NS+LS has its own drawbacks, including, firstly, a very costly search of neighbours of $n \cdot log(n)$ complexity [28]. Secondly, it could fall into a phenomenon called "cycling", where it searches in the same place of the feature space twice or more consecutively. It can happen because of how it handles the already discovered solutions, the selected solutions (=in what is called archive) and a set of solutions, named population, used to generate new solutions for evaluation. Thirdly, it does not evenly and simultaneously search the feature space for new solutions. Instead, it focuses on exploring an area of the feature space with either many unexplored solutions or many high performing already discovered solutions. Anything in between them could be potentially skipped, and thus many interesting areas could be ignored, leaving them unexploited and unexplored.

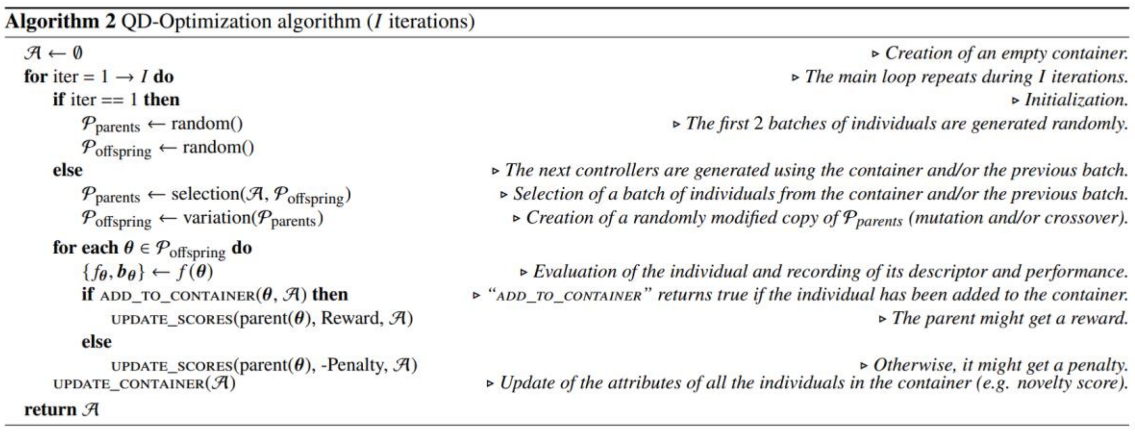
2.5.4.3 Multi-Objective Landscape Exploration (MOLE)

Multi-Objective Landscape Exploration (MOLE) was introduced by Clune et al. [26] and is a multi-objective optimization search [27] that has two objectives: (1) find organisms with high performance, (2) each organism to be as far from others already discovered organisms as possible, where the distance is measured in user-defined Cartesian feature space with connection costs on the x-axis and modularity on the y-axis[26].

A few things to note about MOLE are that it considers one global performance competition for all organisms. Thus, a few high-performing solutions will dominate the set with the selected solutions prohibiting other solutions with slightly less performance from being discovered and selected. Secondly, it does not evenly and simultaneously search the feature space for new solutions just like in NS + LS. Instead, it focuses on exploring an area of the feature space with either many unexplored solutions or many high performing already discovered solutions. Anything in between them could be potentially skipped, and thus many interesting areas could be ignored, leaving them unexploited and unexplored.

2.5.5 QD Framework

Cully and Demiris [29] developed a Quality Diversity Optimisation framework to make the study and development of QD algorithms easier and more accessible. They discussed how two of the most well-studied QD algorithms (MAP-Elites and NS+LC) could be instantiated from the same high-level algorithm (Fig 2.5.5) with different choices in some of its main steps.



Algorithm 2 QD-Optimization algorithm (I iterations)

```
$\mathcal{A} \leftarrow \emptyset$                                         ▹ Creation of an empty container.
for iter = $1 \rightarrow I$ do                                   ▹ The main loop repeats during I iterations.
    if iter == 1 then                                             ▹ Initialization.
        $\mathcal{P}_{\text{parents}} \leftarrow$ random()           ▹ The first 2 batches of individuals are generated randomly.
        $\mathcal{P}_{\text{offspring}} \leftarrow$ random()
    else                                                          ▹ The next controllers are generated using the container and/or the previous batch.
        $\mathcal{P}_{\text{parents}} \leftarrow$ selection($\mathcal{A}, \mathcal{P}_{\text{offspring}}$)   ▹ Selection of a batch of individuals from the container and/or the previous batch.
        $\mathcal{P}_{\text{offspring}} \leftarrow$ variation($\mathcal{P}_{\text{parents}}$)   ▹ Creation of a randomly modified copy of $\mathcal{P}_{parents}$ (mutation and/or crossover).
    for each $\theta \in \mathcal{P}_{\text{offspring}}$ do
        $\{f_{\theta}, b_{\theta}\} \leftarrow f(\theta)$          ▹ Evaluation of the individual and recording of its descriptor and performance.
        if ADD_TO_CONTAINER($\theta, \mathcal{A}$) then           ▹ "ADD_TO_CONTAINER" returns true if the individual has been added to the container.
            UPDATE_SCORES(parent($\theta$), Reward, $\mathcal{A}$)    ▹ The parent might get a reward.
        else
            UPDATE_SCORES(parent($\theta$), -Penalty, $\mathcal{A}$)  ▹ Otherwise, it might get a penalty.
    UPDATE_CONTAINER($\mathcal{A}$)                               ▹ Update of the attributes of all the individuals in the container (e.g. novelty score).
return $\mathcal{A}$
```

Fig 2.7. Pseudocode Snippet of QD Optimisation Algorithm from Cully and Demiris [29]

The introduced QD Optimisation high-level algorithm has three different operators via which QD algorithms can vary. (1) The data structure used to store the solutions discovered so far called container, (2) the way to generate new solutions from a set of solutions (population), called selection operator and (3) the type scores used inside the algorithm for the container and the selection operator called population scores. Despite this definition of the QD framework, there can be variations in other operators mentioned, like mutation or cross-over operators. Later we will see some latest examples of variation operators, including how differentiability can potentially help mutate individuals towards undiscovered and higher-performing solutions. Still, we stick to the latest updated QD framework mentioned in Chatzilygeroudis, Cully, Vassiliades & Mouret [2].

Choosing the types of components for those three parts instantiates a QD Algorithm that works as follows:

Randomly initialize your initial values in the first iteration and repeat four steps until a stopping condition

1. Produce a new set of individuals (=solutions $P_{parents}$) using the selector operator that will be altered to create new individuals ($P_{offspring}$)
2. Evaluate each individual from $P_{offspring}$ using an objective function to get its performance score (=objective value) and a behavioural function to get its behavioural descriptor
3. Potentially add an individual generated to the container of individuals considering already added individuals to the collection and the scores of the individual
4. Update the population scores

Below you can see more details for each of the three main parts of a QD Algorithm

2.5.5.1 Containers

A container is an ordered collection used to store the best and most diverse solutions discovered by the QD Algorithm.

2.5.5.1.1 N-dimensional grid structure

The most popular type of container is the **N-dimensional grid structure.** This container discretises the whole Behavioural space (feature space) into a grid of cells, where each cell represents a different type of solution. Usually, this container stores a single solution per cell (e.g. used in MAP-Elites), but some implementations use more than one solution per cell, e.g. for multi-objective optimization or noisy optimization [30, 31].

2.5.5.1.2 Centroidal Voronoi Tessellation

Centroidal Voronoi Tessellation (CVT) container was introduced in Vassiliades, Chatzilygeroudis, and Mouret [7] and is used in high-dimensional spaces where the N-dimensional grid structure is impractical due to the great amount of computer memory required for storage (i.e. number of cells times the size of memory needed for storing solutions per cell is significant).

Centroidal Voronoi tessellation Container uses the method from computational geometry with the same name to partition a high-dimensional space into well-spread geometric regions. Due to the complexity of constructing Voronoi tessellations to partition a space into geometric areas, it uses a simpler method based on Monte Carlo that applies an approach similar to k-means on a set of random points and random centroids. The latter process forces several centroids to be well spread in the feature space. Those centroids can then be used as the geometric sites (i.e. the centres of the geometric regions similar to those returned from Centroidal Voronoi tessellation).

2.5.5.1.3 Distanced-based archive

The distanced-based archive keeps the solutions not ordered but in an unstructured array using the solutions' behavioural descriptors and their Euclidean distance. The way that a solution is added to this archive is by examining whether the Euclidean distance of a given solution with another solution from the container is greater than a prespecified threshold (denoting that the solution is far away in the feature space from the solutions already discovered) or if it has better performance than its neighbours.

2.5.5.2 Selection Operators

Selection operators are responsible for generating new solutions based on a given population of solutions. Below you can see the most common ones.

2.5.5.2.1 No Selection

No Selection does not use the container to generate new solutions but directly samples new solutions from the solutions' parameter space.

2.5.5.2.2 Uniform Random Selection

Uniform Random Selection samples uniformly new solutions from the solutions stored in the container. The new solutions are solutions from the container perturbed with noise.

2.5.5.2.3 Score Proportionate Selection

Score Proportionate Selection is an extension of random sampling approaches working on the container that applies score-based weighting on the selection. It biases the selection of new individuals based on a particular score called population score.

2.5.5.3 Population Score

In order to generate new solutions and maintain a container, a QD algorithm uses a population score. Evolutionary algorithms usually use the fitness score, which considers each solution's performance only. That is, the selection biases solutions with higher fitness (=performance). There is also the novelty score that favors solutions with greater distance (in feature space) from other solutions in a container. Moreover, the curiosity score estimates the propensity of a solution to generate solutions that are added to the container. Last, Go-Explore [32,33] introduced a new score that biases the selection towards newly discovered solutions with the idea that newly discovered solutions contain interesting features that can lead to more unexplored regions in the feature space.

2.5.6 Criteria for Measuring the QD Algorithms

There are many different ways to quantify the quality of QD Algorithms. The most used measures for the performance of a QD algorithm focus on the produced collection of solutions and examine two different criteria:

1. The performance of the solution found for each cell of the container (type of solution). This measures how much the solutions found were optimized.
2. The coverage of the behaviour space (how much of the feature space is explored/covered).

There are a lot of different specific measures that can be derived from those criteria, like Global Performance (highest performance of a solution found divided by the highest performance of a solution that is possible to be found in the defined space), Global Reliability (average across all cells of the division between the highest performing solution found for a cell by a specific algorithm and the highest possible performance of a solution that was ever found by any algorithm for that cell – all cells for which a solution was not found are considered as 0) and Precision (same as Global Reliability but does not include cells for which a solution was not found) [1].

2.5.7 The concept of Emitters

The concept of Emitters in QD was Introduced by Fontaine, Nikolaidis, Togelis and Hoover [6] with the proposal of the CMA-ME algorithm. An Emitter is an instance of a QD Algorithm that uses a specific selection operator (how to select new solutions at each iteration) and an adaptation rule (how to update/adapt the QD instance's rule for selecting new solutions in the next iteration). Different QD Emitters can use different selection operators and/or different adaptation rules. Solutions generated by the emitters are saved in a single unified archive based on their corresponding behaviours.

In general, the use of the selection operator and adaptation rule is defined with two methods following an ask-tell interface adopted from Pycma [66] and described below:

2.5.7.1 Ask

The ask method of an emitter operates on an emitter and does not usually take any other argument except the emitter itself. It basically implements the selection operator. It interacts with an archive (container) to generate some solutions using a selection operator, and returns those solutions. The ask method can be specified on each emitter, and it can perform different operations for each emitter.

2.5.7.2 Tell

This method is used to provide feedback to the selector operator and adapt its rules so that to guide the selector operator towards better solution generation in the next iteration. This adaptation mechanism, includes updating the archive (=container) based on the solutions given as argument, and using a policy for adding solutions to the container with the help of a population score. The tell method usually takes as arguments some solutions (e.g. the solutions given from the ask method), the performance of each solution (objective values) and the behavioural descriptor of each solution (and, of course, the emitter it operates on). Like the ask method, the tell method can be specific on each emitter and can perform different operations for each emitter.

2.5.8 Quality-Diversity Emitters/Algorithms

There are different QD Algorithms developed, as well as variations of them. Here we will describe the most used ones in the literature on QD.

2.5.8.1 MAP-Elites

Multi-dimensional Archive of Phenotypic Elites (MAP-Elites) is the first of the simplest and most well-studied QD algorithms that were also the first to introduce the concept of Illumination. It was inspired by Novelty Search + Local Competition (NS+LC) [5] and the Multi-Objective Landscape Exploration algorithm (MOLE) [26]. In a few words, MAP-Elites generates a large diversity of high-performing solutions that are different according to specific features of interest. MAP-Elites explores more of the available solutions search space, and it tends to find a better overall solution than state-of-the-art search algorithms. Because of Isotropic Gaussian Distribution in its core, it is also called MAP-Elites Iso to differentiate it from other variations.

```
procedure MAP-ELITES ALGORITHM (SIMPLE, DEFAULT VERSION)
    ($\mathcal{P} \leftarrow \emptyset, \mathcal{X} \leftarrow \emptyset$)    ▷ Create an empty, $N$-dimensional map of elites: {solutions $\mathcal{X}$ and their performances $\mathcal{P}$}
    for iter $= 1 \rightarrow I$ do    ▷ Repeat for $I$ iterations.
        if iter $< G$ then    ▷ Initialize by generating $G$ random solutions
            $\mathbf{x}' \leftarrow$ random_solution()
        else    ▷ All subsequent solutions are generated from elites in the map
            $\mathbf{x} \leftarrow$ random_selection($\mathcal{X}$)    ▷ Randomly select an elite $x$ from the map $\mathcal{X}$
            $\mathbf{x}' \leftarrow$ random_variation($\mathbf{x}$)    ▷ Create $x'$, a randomly modified copy of $x$ (via mutation and/or crossover)
        $\mathbf{b}' \leftarrow$feature_descriptor($\mathbf{x}'$)    ▷ Simulate the candidate solution $x'$ and record its feature descriptor $\mathbf{b}'$
        $p' \leftarrow$performance($\mathbf{x}'$)    ▷ Record the performance $p'$ of $x'$
        if $\mathcal{P}(\mathbf{b}') = \emptyset$ or $\mathcal{P}(\mathbf{b}') < p'$ then    ▷ If the appropriate cell is empty or its occupants's performance is $\leq p'$, then
            $\mathcal{P}(\mathbf{b}') \leftarrow p'$    ▷ store the performance of $x'$ in the map of elites according to its feature descriptor $\mathbf{b}'$
            $\mathcal{X}(\mathbf{b}') \leftarrow \mathbf{x}'$    ▷ store the solution $x'$ in the map of elites according to its feature descriptor $\mathbf{b}'$
    return feature-performance map ($\mathcal{P}$ and $\mathcal{X}$)
```

Figure 2.8. MAP-Elites pseudocode from Mouret and Clune [1]

MAP-Elites is quite simple, both conceptually and to implement. The pseudocode of the algorithm can be seen above.

2.5.8.1.1 Algorithmic Decisions

Initially, for the algorithm to work, three decisions need to be made, described in the next three steps:

First, a user chooses an objective function f(θ) that evaluates the performance of a solution θ. For example, if searching for robot morphologies, the objective function could be how fast the robot is.

Second, the user chooses the features of the solutions (N dimensions of solution variation) to define the feature space. One feature (dimension of interest) for robot morphologies could be how tall the robot is; another could be its weight, a third could be its energy consumption per meter moved, etc. Using those features, we can define a behavioural function m(θ) to return a 1-d vector of length N representing the solution's features.

Third, each feature (dimension of variation) is discretized based on user preference or available computational resources. This granularity could be set manually or automatically based on the available resources. For instance, if we have a feature space defined by two features taking values between 0 and 1 inclusive, and we wanted to use the n-dimension grid container, we could describe each dimension with ten cells. Thus 10 times 10 for both dimensions would give us 100 different behavioural descriptors (cells representing a different combination of the two features in the feature space) to discretise the feature space.

2.5.8.1.2 Mapping Solutions Descriptions to Solutions

Before we dive into the main steps of the algorithm, there are two things to consider:

First, Given a particular discretization, MAP-Elites searches for the highest performing solution for each cell in the N-dimensional feature space. For example, MAP-Elites will search for the fastest robot that is tall, heavy, efficient etc. The search is done in the space of the descriptions of solutions, referred to as a search space. In our example, the search space contains all possible descriptions of robot morphologies (note that we search the space of descriptions of robot morphologies and not directly the morphologies or the feature space). Using the Evolutionary jargon, the description of the robot morphology is a genome or genotype, and the robot morphology itself is the phenotype, or px.

Second, there are two main ways two map genotypes (descriptions of solutions) to phenotypes (actual solutions). Using direct encoding, in which each element in the genome specifies an independent component of the phenotype [36, 37, 38]. In that case, it is straightforward to map genotypes into phenotypes and then measure performance and features (evaluating the phenotype in a simulator or the real world if necessary). Using indirect encoding, also known as generative or developmental encoding, in which information in the genome can be reused to affect many parts of the phenotype. The latter showed that they could improve regularity, performance, and evolvability [36 - 45]. In other words, a complex process can exist that maps genome x → to phenotype px → to features bx and performance fx.

2.5.8.1.3 Main Algorithm

MAP-Elites starts by randomly generating λ genomes (created from a fixed distribution $\theta \sim N(0, I)$ where I is the covariance matrix of the distribution restricted to a multiplication between a scalar σ^2 and an identity matrix I ; in this case σ^2 is 1) and determining the performance and features of each of them using the objective and behavioural functions, respectively. In random order, those genomes are placed into the cells to which they belong in the feature space. In the case when multiple genomes map to the same cell, the highest-performing genome per cell is retained (there are also variations of this step that can allow more than one). At that point, the algorithm is considered fully initialized and repeats the following steps until a termination criterion is reached.

(1) λ cells in the feature map are uniformly randomly selected, and the genomes of those cells produce offspring's via mutation and/or crossover. Each genome selected is perturbed with a fixed-variance σ isotropic Gaussian noise. If θ is a chosen solution, then the solution generated from that solution is going to be $\theta' = \theta_i + \sigma \cdot N(0, I)$.

(2) The features and performance of the offspring solutions are determined with the use of the behavioural and objective functions, and for each offspring solution, (a) if its corresponding cell is empty, the offspring is placed in the cell, or (b) if the offspring is higher-performing than the current genome stored in the cell, then the latter is replaced (discarded from the container) by the former.

There are a lot of conditions that can be used as termination criteria, such as if a set amount of time expires, a fixed amount of computational resources are consumed, or some property of the archive is accomplished (e.g. percentage of the container's cells being filled).

2.5.8.1.4 MAP-Elites Details

MAP-Elites does not guarantee to fill all the cells in the feature space because:

(1) There may be no genome that maps to a particular cell in the feature space. For example, it may be impractical for a robot to have a certain height and weight (i.e. due to laws of nature).

(2) Even if a genome exists for a specific cell of the feature space, the algorithm may not generate a genome for that cell.

Moreover, MAP-Elites can map many genotypes to the same cell in the feature space. In reality, there might be an infinite number of genotypes for some or all cells of feature space. For example, many different blueprints of robot morphology can produce a robot with the same height, weight, and energy consumption. Thus, because of this difficulty in understanding with certainty which cells of the feature space will be filled with genomes, it is not possible to consider searching the feature space directly.

2.5.8.2 MAP-Elites (Iso-LineDD)

Multi-dimensional Archive of Phenotypic Elites with Isotropic and line Distance-Dependent variation is a variation of MAP-Elites that differs only on the variation step. More specifically, it picks two genotypes from the container (instead of one) and mutates one of them using Isotropic Gaussian Noise (just in MAP-Elites) plus the distance between those two genotypes (their distance simulates the correlation between the two solutions). Below we describe in depth some concepts and, finally the Iso-LineDD variation operator.

2.5.8.2.1 Elite Hypervolume

Vassiliades and Mouret [19] supported that all the high-performing solutions (named elites) of the search space, as found by MAP-Elites, are likely to be concentrated in a sub-part of the genotypic space. This statement was derived from the observation that many genotypes can be mapped to the same behavioural cell in the feature space because the relationship between a genotype and its behaviour can be non-linear. Thus high-performing solutions can be located around the same area. Therefore, they introduced the "elite hypervolume", describing this subpart of genotypic space that can be seen in the image below.

Mathematically, an Elite Hypervolume H is the subset of the n-dimensional valued genotype space, assumed symbolized X (H $\subset$ X), that encloses a set of m individuals, E, each of them being the highest-performing solution (i.e. an elite) of its corresponding are/niche in the feature space (= behaviour space):

$$E = \left\{ \arg\max_{x_1} f(x_1), \ldots, \arg\max_{x_q} f(x_q) \right\} \subseteq H \;\; s.t.\; x_i \in C_i$$

Where $C_i \subset X$, and C_i is the subset of the genotype space that corresponds to the *i*th region in feature space, for i = 1 … m, m $\leq$ k (k is the niche capacity).

The goal of an Illumination algorithm with the Elite Hypervolume is to find the set E since finding the H is computationally expensive.

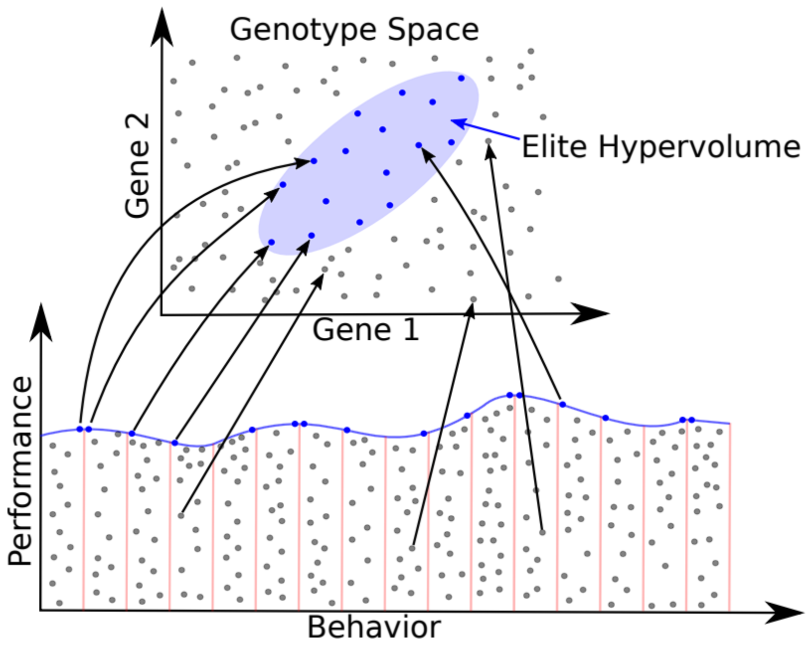


Fig 2.9. The concept of Elite Hypervolume from Vassiliades and Mouret [19]

2.5.8.2.2 <u>Directional Variation</u>

Vassiliades and Mouret [19] used the idea of elite hypervolume to support that if we assume that we have two genotype elites that share a large part of their genome (part of their solutions' descriptions is similar), then we could bias the variation operator (variation operator applied on genotypes) to guide the genome generation towards producing new candidates in the elite hypervolume, and thus finding other elites more efficiently.

These similarities between different genotype elites can be extracted by extracting correlations on the level of the genotypes and then sampling new genotypes based on those correlations. For example, such a way of generating new genotypes can be done using a multivariate Gaussian distribution N (μ, Σ), where the covariance matrix Σ models the correlations.

Below you can see three different ways of sampling new genotypes using genotypes' similarities even with the way mentioned above. The different approaches can create a different variation operator for a QD Algorithm. Below you can see a description of three of them from Vassiliades and Mouret [19]:

2.5.8.2.2.1 Iso

Use an Isotropic Gaussian distribution to sample new elites around one genotype. The second elite is not used. This selection operator is the one used by the MAP-Elites.

2.5.8.2.2.2 LineDD

Distance-Dependent line variance (LineDD) uses the correlation between two elites and samples a new elite around the first genotype but only across the line that connects the two elites.

2.5.8.2.2.3 Iso + LineDD

It is a combination of Iso + LineDD operators that generates a new elite from a two-parent elite by sampling a new elite around the first elite using Isotropic Gaussian distribution biased towards the direction of correlation of the second elite thus taking into consideration the correlation between those two elites.

More specifically, this selector operator works as follows:

1) Select uniformly randomly two elite genotypes x_i and x_j from the container.

2) Generate a new genotype as follows:

$$x_i' = x_i + \sigma_1 \cdot N(0, I) + \sigma_2 \cdot (x_j - x_i) \cdot N(0,1)$$

Note that the direction used here is considered positive $x_j - x_i$, whereas we could also use a negative direction, that is, $x_i - x_j$ in the equation above.

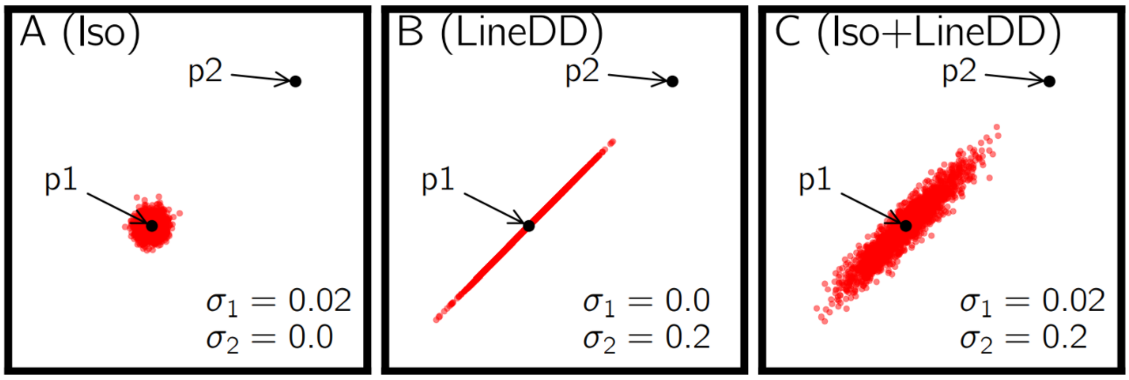


Fig 2.10. Variation Operators that use the Direction Correlation between two elites to bias the variation of one of them from Vassiliades and Mouret [19]

Note that, in the variation operators mentioned above, we used the word elite referring to elites of Elite Hypervolume because those three variation operators were a discussion on why Iso+LineDD could perform better especially when used on an Elite Hypervolume.

2.5.8.3 CMA-ME

Covariance Matrix Adaptation MapElites (CMA-ME) is a scheduling algorithm that combines MAP-Elites and CMA-ES for an improved search of solutions in search space. CMA-ME introduced the concept that the emitters discussed above can be defined using specific types of emitters that are an extension of CMA-ES instances.

To clarify how CMA-ME is an emitter that uses emitters, consider the pseudocode below of the general structure of CMA-ME. That is, CMA-ME generates solutions in search space in a round-robin fashion. The solutions are generated using emitters. Each solution is generated in the same way for all emitters by sampling from the distribution N(m, C). The method *generate_solution* below is the *ask* method from the Emitters framework above, and it operates in the same way for each emitter working with.

On the other hand, the method *return_solution* is the *tell* method from the Emitters framework above and represents the procedure for adapting the sampling distribution(the Covariance Matric of distribution N(m, C)) of an Emitter and maintaining the sampled population for further processing in next iterations. Here the *tell* method depends on the type of each emitter since it requires different operations per Emitter.

There are three different types of CMA-ME Emitters: Improvement, Optimising and the Random Direction, all of which are discussed in detail below.

```
Algorithm 1: Covariance Matrix Adaptation MAP-Elites

CMA-ME ($evaluate, n$)
    input : An evaluation function $evaluate$ which computes
            a behavior characterization and fitness, and a
            desired number of solutions $n$.
    result: Generate $n$ solutions storing elites in a map $M$.
    Initialize population of emitters E
    for $i \leftarrow 1$ to $n$ do
        Select emitter $e$ from $E$ which has generated the least
          solutions out of all emitters in $E$
        $x_i \leftarrow$ generate_solution($e$)
        $\beta_i, fitness \leftarrow evaluate(x_i)$
        return_solution($e, x_i, \beta_i, fitness$)
    end
```

Fig 2.11. Pseudocode Snippet of general CMA-ME from Fontaine, Nikolaidis, Togelis and Hoover [6]

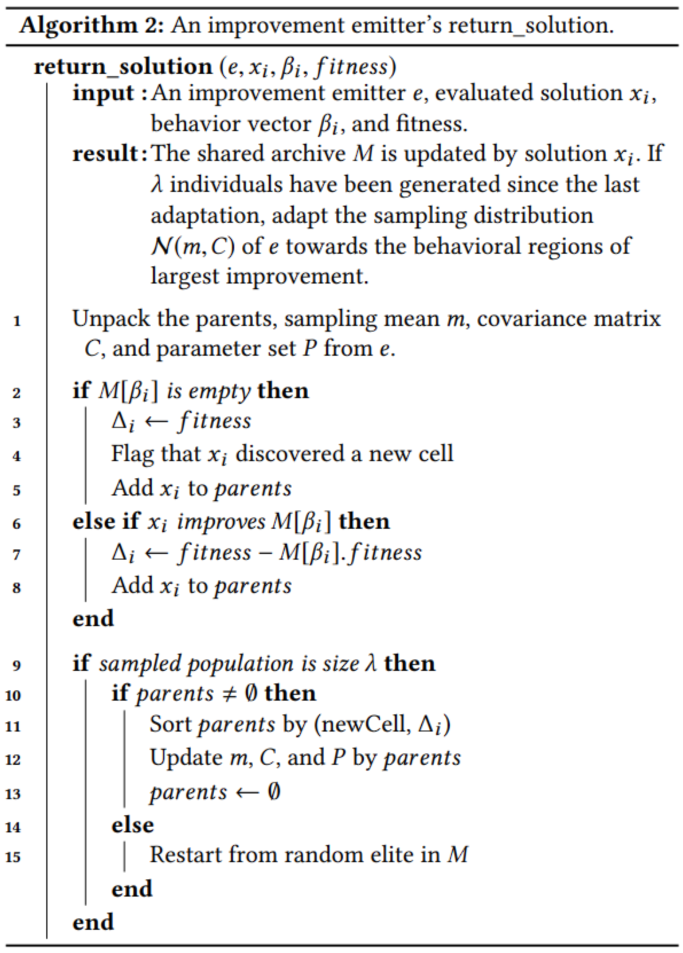
2.5.8.3.1 **CMA-ES Instances vs CMA-ME Emitters**

An optimizing Emitter is an extension of a CMA-ES instance that uses the CMA-ES instance according to a rule for ranking solutions to Each emitter maintains a sampling mean m, a covariance matrix C, and a parameter set P that contains additional CMA-ES related parameters (e.g., evolution path). One of the differences between Emitters and CMA-ES instances is that CMA-ES instances increase the likelihood of successful future evolution steps, whereas CMA-ME Emitters take an additional step to what a CMA-ES instance does by adjusting the ranking rules that guide which solutions are updating the container should be used to update the Covariance Matrix of a CMA-ES Instance used by the Emitter. This guided update of the Covariance Matrix is done to maximize the likelihood that future steps in a given direction will result in archive improvements. Another difference is that while CMA-ES restarts its search based on the best current solution, emitters rank the solutions that will update the container based on specific to each Emitter rules that guide when the search will be restarted and when the sampling distribution of a CMA-ES instance should be adapted.

2.5.8.3.2 **Improvement Emitter**

Improvement emitters adjust their goals based on where progress (improvement in the performance of the archive) is currently being made. The pseudocode above shows the implementation of the *return_solution* (method *tell* of the CMA-ME Emitter).

At each iteration, solutions are generated from distribution N(m, C), and each of them is given to the tell function of the emitter (return_solution method as appears in the pseudocode above). Each solution given to the return_solution method is mapped to a behaviour β_i and a cell $M[\beta_i]$ in the archive (container or map). If the cell is empty (line 2), or if β_i has higher fitness than the existing solution in the cell (line 6), x_i is added to the new generation's parents, and the container (=archive) is updated. Note that each solution's improvement to the archive is recorded (lines 4 and 7). The process repeats until the generation of solutions x_i reaches size λ (line 9), where the emitter is adapted. When the time to adapt the Emitter comes, if there are parents that improved the archive, all the parents are ranked before being given to update the Emitter's m, C and P parameters. They are ranked firstly based on whether they fill a new cell, then by whether they improve an existing one and last by not changing anything. Each of those groups is ranked in descending order by the improvement in the objective value that they provided to the archive when they were added to the archive (i.e. ranked by the difference between their performance and the performance of the solution occupying the cell they refer to in the container; if the cell is not occupied then the performance of the parent is considered only) (line 11). If, at an iteration, the parents do not improve the map, the emitter restarts(line 15).



```
Algorithm 2: An improvement emitter's return_solution.

return_solution ($e, x_i, \beta_i, fitness$)
    input : An improvement emitter $e$, evaluated solution $x_i$,
            behavior vector $\beta_i$, and fitness.
    result: The shared archive $M$ is updated by solution $x_i$. If
            $\lambda$ individuals have been generated since the last
            adaptation, adapt the sampling distribution
            $\mathcal{N}(m, C)$ of $e$ towards the behavioral regions of
            largest improvement.
1   Unpack the parents, sampling mean $m$, covariance matrix
     $C$, and parameter set $P$ from $e$.
2   if $M[\beta_i]$ is empty then
3       $\Delta_i \leftarrow fitness$
4       Flag that $x_i$ discovered a new cell
5       Add $x_i$ to $parents$
6   else if $x_i$ improves $M[\beta_i]$ then
7       $\Delta_i \leftarrow fitness - M[\beta_i].fitness$
8       Add $x_i$ to $parents$
    end
9   if sampled population is size $\lambda$ then
10      if $parents \neq \emptyset$ then
11          Sort $parents$ by (newCell, $\Delta_i$)
12          Update $m$, $C$, and $P$ by $parents$
13          $parents \leftarrow \emptyset$
14      else
15          Restart from random elite in $M$
        end
    end
```

Fig 2.12. Pseudocode Snippet of CMA-ME Improvement Emmiter's tell (here called return_solution) method from Fontaine, Nikolaidis, Togelis and Hoover [6]

2.5.8.3.3 **Optimising Emitter**

Optimising Emitter is an exploration to ask the question of whether the restarts alone in CMA-ES are enough to promote good exploration as they are in multi-modal methods [6].

Overall, at each iteration, solutions are generated from distribution N(m, C), and then the solutions that will update the Emitter later are the ones that improve the archive (fill an empty cell of the container or replace a solution in a cell with a more high-performing solution). Those solutions are recorded and used when λ such solutions are collected to update the parameters m, C and P of the Emitter. Before they update the Emitter, they are ranked in descending order by their performance (objective values), prioritising the solutions that fill empty cells in the archive. This can be thought of as going toward areas in the space with high-performing solutions that are likely to discover new solutions but not necessarily improve existing archive cells.

The only conceptual difference with a CMA-ES instance is that when the Emitter is restarted, the mean m of the distribution N(m, C) selected is chosen from the location of an elite rather than the fittest solution discovered so far.

2.5.8.3.4 **Random Direction Emitter**

Random direction Emitter tries to perform a "random walk" in the feature space to find high-performing solutions. This means that it operates a search in the feature space. But doing so poses the issue of working with a non-linear encoding or mapping from solutions (genotypes) to behavioural descriptors, thus making the exact calculation of the inverse mapping from a behavioural descriptor to the right solution very difficult or even impossible. Random direction emitter is designed to estimate this inverse encoding of this correspondence problem.

In general, at each iteration, solutions are generated from the distribution N(m, C), and the solutions that will update the Emitter later are the ones that improve the archive (fill a new cell of the container or replace a solution with a more high-performing solution). Those solutions are recorded and used when λ such solutions are gathered to update the parameters m, C and P of the Emitter. Before they update the Emitter, they are ranked in descending order based on their projection value (dot product between a behaviour descriptor of a solution and a bias vector denoting the direction which is defined when initializing or restarting the CMA-ES instance), giving priority first to the solutions that fill empty cells in the archive.

More specifically, when a Random Direction Emitter restarts, it emulates a step in a random walk by selecting a bias vector v_{β} that indicates a random direction to move toward in behaviour space. At later iterations (where a restart does not happen) and when having accumulated λ solutions that improve the archive (parents of the next generation), this random direction selected is enforced by creating a new covariance matrix for the CMA-ES instance such that it biases the search in the direction v_β. To create this new covariance matrix, solutions θ_i from the search, space is mapped to behaviour space β_i. The mean m_β of the behaviour descriptors of the λ parents is calculated in behaviour space, and each direction of the parents' behavioural descriptors with respect to the mean is calculated. The parents used to update the Emitter are ranked by their projection value (dot product between behaviour descriptor and the bias vector v_β) against the line $m_\beta + v_\beta$. If, at an iteration, none of the solutions examined to improve the archive, the emitter restarts from a randomly chosen elite with a new bias vector v_β.

2.5.9 Differentiable Quality-Diversity (DQD)

Differentiable Quality Diversity (DQD) Optimisation is an extension of Quality Diversity concerned with how the gradients of the objective and behavioural functions can help guide the search for high-performing yet qualitatively diverse solutions in QD. It was introduced by Fontaine and Nikolaidis [4].

2.5.9.1 DQD Problem Definition

To help define the DQD problem, let us relax some constraints of the general QD Problem first. First of all, let us consider that the feature space (or behaviour space) B $\in R^k$ is discretised (defined above in the containers section and used for the first time in MAP-Elites) via a tessellation method. Let us call T the tessellation of B and let us assume that the tessellation T has M unique cells. In each cell i a solution θ_i can be stored. The occupants of the cells in T at any single moment form an archive of solutions. Each solution can be evaluated using an objective function f(θ_i) that measures its performance and a behavioural function m(θ_i) that provides its features or behaviour descriptors that determine its cell position in the Tesselation T. Thus, the goal of QD is to find a set of solutions θ_i, i $\in \{1, \ldots, \mathrm{M}\}$, such that each θ_i occupies one unique cell in T and that each solution maximises the objective value of its cell in the archive:

$$max \sum_{i=1}^{M} f(\theta_i)$$

Using this relaxed definition of QD, the DQD problem is defined as a QD problem where both the objective function f and the behaviour function m are first-order differentiable.

2.5.9.2 Gradient Arborescence

The idea of Gradient arborescence is similar to the idea of gradient ascent. It makes greedy ascending steps based on the objective function f. In addition to the ascending steps trend towards higher objective values, gradient arborescence, unlike gradient ascent, encourages exploration by branching via the behavioural function m. Note that there is no meaning of ascending or descending with the behavioural function m since the goal is not maximisation nor minimization of a function, but instead an exploration of its value space. The term arborescence was adopted from the minimum arborescence problem in graph theory [48].

2.5.9.3 DQD Emitters/Algorithms

2.5.9.3.1 <u>OMG-MEGA</u>

Objective and Measure Gradient MAP-Elites via Gradient Arborescence (OMG-MEGA) is the exploitation of the gradients of an objective function f and a behaviour function m that are first-order differentiable derived in MAP-Elites.

The idea is that maximizing a linear combination of scalar behavioural functions (also called here measures) $\sum_{j=1}^{k} c_j\, m_j(\theta)$, where c is a k-dimensional vector of coefficients (or otherwise maximizing the dot product of a vectorized behavioural function m with a vector of coefficients c: $m \cdot c$) enables movement in a k-dimensional feature space. Including the objective function f in the linear sum enables movement in an objective-measure space. Thus, the problem can be considered maximizing the following function:

$$g(\theta) = |c_0| f(\theta) + \sum_{j=1}^{k} c_j\, m_j(\theta)$$

Note that the coefficient in front of the objective function needs to be positive so that maximizing g will result in maximising g (and not minimising). The direction in the feature space is determined by the sign and magnitude of the coefficients c_j.

Based on this maximization problem, a gradient function can be derived that can be used to perturb a solution θ (in the variation operator). This gradient function is:

$$\nabla \mathrm{g}(\theta) = |c_0| \nabla f(\theta) + \sum_{j=1}^{k} c_j \nabla m_j(\theta)$$

Thus, the gradient perturbation/variation of a solution θ_i is defined as:

$$\theta' = \theta_i + \nabla \mathrm{g}(\theta_i) = |c_0| \nabla f(\theta_i) + \sum_{j=1}^{k} c_j \nabla m_j(\theta_i)$$

The coefficients c are sampled from $N(0, \sigma_g I)$. Last, the gradients are normalized to balance the contribution of each function.

Below you can see more information on how this variation operator can be used in MAP-Elites creating two variations of DQD Algorithms.

2.5.9.3.1.1 OMG-MEGA (iso)

The algorithm works similar to the MAP-Elites with Isotropic Gaussian distribution. It repeats the four steps below until a termination condition:

1. Initially, the algorithm selects λ solutions from container and perturbs them with Isotropic Gaussian noise.
2. It derives each solution's objective value and behaviour descriptor by evaluating it with the objective and behavioural functions, respectively. It also derives the derivatives of the objective function f and the behavioural function m with respect to each solution, vectors ∇f and $\nabla \mathrm{m}$, respectively.
3. Uses the gradient perturbation for OMG-MEGA above to create a new solution for each of the initial λ solutions.
4. Stores the original λ solutions as well as the perturbed ones in the container.

2.5.9.3.1.2 OMG-MEGA (line)

The algorithm works exactly as the OMG-MEGA (iso), with the only difference the step 1, where instead of Isotropic Gaussian distribution, it uses the selection operator with Iso + LineDD.

2.5.9.3.2 OG-MAP-Elites

Objective Gradient MAP-Elites (OG-MAP-Elites) operates only on performing ascending steps on the objective function and not using the gradients of the behavioural function to explore the feature space. Thus, the maximization is done on the function:

$$g(\theta) = |c_0| f(\theta)$$

And therefore, the gradient perturbation/variation of a solution θ_i is defined as:

$$\theta' = \theta_i + \nabla \mathrm{g}(\theta_i) = |c_0| \nabla f(\theta_i)$$

2.5.9.3.2.1 OG-MAP-Elites (iso)

The algorithm works exactly as OMG-MEGA, but instead of considering the derivatives of both the objective and behavioural functions, it only considers the ones for the objective function. Other than that, they are exactly the same.

Initially, the algorithm samples λ solutions from Isotropic Gaussian distribution. It repeats the four steps below until a termination condition:

1. It derives each solution's objective value and behaviour descriptor by evaluating it with the objective and behavioural functions, respectively. It also derives the derivatives of the objective function f, vector ∇f.
2. Uses the gradient perturbation for OG-MEGA above to create a new solution for each of the initial λ solutions.
3. Stores the original λ solutions as well as the perturbed ones in the container.
4. Stores the original λ solutions as well as the perturbed ones in the container.

2.5.9.3.2.2 OG-MAP-Elites (line)

The algorithm works exactly as the OG-MEGA (iso), with the only difference the step 1, where instead of Isotropic Gaussian distribution, it uses the selection operator with Iso + LineDD.

2.5.9.3.3 <u>CMA-MEGA</u>

Covariance Matrix Adaptation MAP-Elites via a Gradient Arborescence (CMA-MEGA) uses Gradient Arborescence in the same way as OMG-MEGA, but in addition to that, it uses CMA-ES to optimise the c coefficients of the g(θ) sum so that to search for high-performing solutions and improve the exploration of the feature space.

More specifically, the goal remains the same as in CMA-MEGA. To maximise the following sum:

$$g(\theta) = |c_0| f(\theta) + \sum_{j=1}^{k} c_j\, m_j(\theta)$$

Where $\sum_{j=1}^{k} c_j\, m_j(\theta)$ enables movement in a k-dimensional feature space and $|c_0| f(\theta)$ enables movement in objective space. The c are coefficients that are adjusted using CMA-ES. From the function above, the following gradient equation can be derived:

$$\nabla \mathrm{g}(\theta) = |c_0| \nabla f(\theta) + \sum_{j=1}^{k} c_j\, \nabla m_j(\theta)$$

Thus, the gradient perturbation of the solutions using the gradient equation is:

$$\theta' = \theta_i + \nabla \mathrm{g}(\theta_i) = |c_0| \nabla f(\theta_i) + \sum_{j=1}^{k} c_j\, \nabla m_j(\theta_i)$$

In OMG-MEGA, the coefficients c in the equations above are sampled from a fixed Gaussian distribution $N(0, \sigma_g I)$. The idea of CMA-MEGA is to adapt those coefficients to help both the exploration of the feature space and the search for high-performing solutions.

Thus the selection of coefficients c is considered an optimization problem with the goal of maximizing the QD objective:

$$max \sum_{i=1}^{M} f(\theta_i\,)$$

The coefficients c are modelled as a distribution of a k + 1-dimensional Gaussian N (μ, Σ). Given those Gaussian distributions and a solution θ, the coefficients can be sampled from the distribution c ∼ N (μ, Σ), and can be used to perturb θ using the gradient perturbation above. Last, the N (μ, Σ) Gaussians are adapted towards the direction of maximum increase of the QD objective.

The way N (μ, Σ) is updated is: Sample a population of λ coefficients from c_i ∼ N (μ, Σ) and generate λ solutions θ_i using the gradient perturbation. Then compute Δ_i from CMA-ME's improvement ranking (Δ_i is the improvement that each solution adds to a cell of the archive; the same approach as the CMA-ME Improvement Emitter) for each candidate solution θ_i. Update N (μ, Σ) with CMA-ES update rules for the ranking Δ_i To dynamically adapt the distribution of coefficients c to maximize the QD objective.

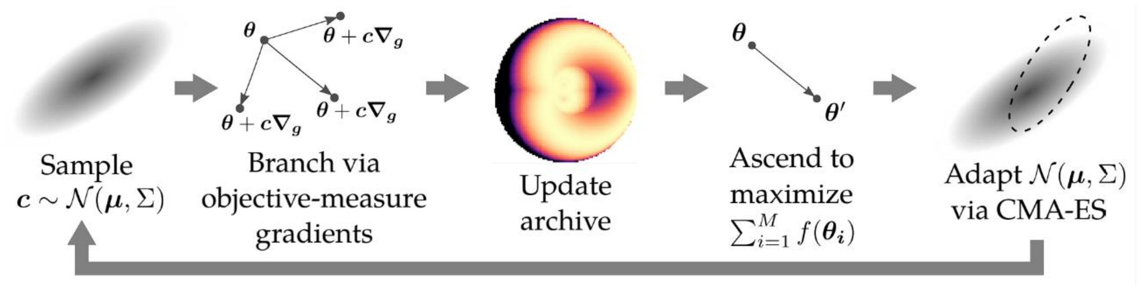

θ
$\theta + c\nabla_g$
$\theta + c\nabla_g$
$\theta + c\nabla_g$
θ
θ'
Sample $c \sim \mathcal{N}(\mu, \Sigma)$
Branch via objective-measure gradients
Update archive
Ascend to maximize $\sum_{i=1}^{M} f(\theta_i)$
Adapt $\mathcal{N}(\mu, \Sigma)$ via CMA-ES

Fig. 2.13: Overview of CMA-MEGA from Fontaine and Nikolaidis [4]

The figure below shows the pseudocode for CMA-MEGA. In line 3, a current solution is evaluated, returning an objective value f, vector of behaviour values m, a gradient vector for the objective function (gradient of a function with a vector as input and a scalar as output) and a Jacobian matrix for the behaviour function (gradient of a function with a vector as input and a vector as output). The objective and behaviour gradients are normalised for stability (line 4). Then a solution is added to the archive (line 5) only if the solution discovers an empty cell in the archive or if it improves an existing cell. The feature space is tessellated, and thus the behavioural function m places a solution θ into one of the M cells of the archive. In line 7, a population of λ coefficients is sampled from a multi-variate Gaussian retained by CMA-ES. In lines 8-9, the solutions are perturbed using the gradient perturbation equation described above. Then the perturbed solutions are evaluated (line 10) and added to the archive (line 11). Note that adding the new solutions to the archive includes computing the improvement that each solution adds to its cell in the archive. This improvement Δ_i is the difference in the objective value between the sampled solution θ_i and the existing solution, if one exists, or as the absolute objective value of the sampled solution if θ_i belongs to an empty cell. In line 13, those improvement values are used to rank the sampled gradients ∇_i. As in CMA-ME, the ranking prioritises first all samples that discover new cells and subsequently all samples that improve existing cells, retaining an in-class descending order by their improvement value Δ_i. Then an ascending gradient step is computed as a linear combination of gradients (line 14), following the recombination weights wi from CMA-ES [35] based on the computed improvement ranking. These weights correspond to the log-likelihood probabilities of the samples in the natural gradient interpretation of CMA-ES [49]. Afterwards, in line 16, CMA-ES adapts the multi-variate Gaussian parameters μ and Σ as well as internal search parameters p of the Emitter, using the improvement ranking of the coefficients.

2.5.9.3.4 CMA-MEGA (ADAM)

The only difference between CMA-MEGA and CMA-MEGA(ADAM) is in the use of the ascending gradient step. That is, the ascending gradient step computed in line 14 and used in line 15 is now used in an Adam gradient optimization step [50] that replaces line 15.

Algorithm 1 Covariance Matrix Adaptation MAP-Elites via a Gradient Aborescence (CMA-MEGA)

```
CMA-MEGA ($evaluate, \boldsymbol{\theta_0}, N, \lambda, \eta, \sigma_g$)
    input : An evaluation function $evaluate$ which computes the objective, the measures, gradients
            of the objective and measures, an initial solution $\boldsymbol{\theta_0}$, a desired number of iterations $N$, a
            branching population size $\lambda$, a learning rate $\eta$, and an initial step size for CMA-ES $\sigma_g$.
    result : Generate $N(\lambda + 1)$ solutions storing elites in an archive $A$.
1   Initialize solution parameters $\boldsymbol{\theta}$ to $\boldsymbol{\theta_0}$, CMA-ES parameters $\boldsymbol{\mu} = \mathbf{0}$, $\Sigma = \sigma_g I$, and $\boldsymbol{p}$, where we
     let $\boldsymbol{p}$ be the CMA-ES internal parameters.
2   for $iter \leftarrow 1$ to $N$ do
3       $f, \boldsymbol{\nabla}_f, \boldsymbol{m}, \boldsymbol{\nabla}_{\boldsymbol{m}} \leftarrow$ evaluate($\boldsymbol{\theta}$)
4       $\boldsymbol{\nabla}_f \leftarrow$ normalize($\boldsymbol{\nabla}_f$), $\boldsymbol{\nabla}_m \leftarrow$ normalize($\boldsymbol{\nabla}_m$)
5       update_archive($\boldsymbol{\theta}, f, \boldsymbol{m}$)
6       for $i \leftarrow 1$ to $\lambda$ do
7           $\boldsymbol{c} \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$
8           $\boldsymbol{\nabla}_i \leftarrow c_0 \boldsymbol{\nabla}_f + \sum_{j=1}^{k} c_j \boldsymbol{\nabla}_{m_j}$
9           $\boldsymbol{\theta}'_i \leftarrow \boldsymbol{\theta} + \boldsymbol{\nabla}_i$
10          $f', *, \boldsymbol{m}', * \leftarrow$ evaluate($\boldsymbol{\theta}'_i$)
11          $\Delta_i \leftarrow$ update_archive($\boldsymbol{\theta}'_i, f', \boldsymbol{m}'$)
12      end
13      rank $\boldsymbol{\nabla}_i$ by $\Delta_i$
14      $\boldsymbol{\nabla}_{\text{step}} \leftarrow \sum_{i=1}^{\lambda} w_i \boldsymbol{\nabla}_{\text{rank[i]}}$
15      $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \boldsymbol{\nabla}_{\text{step}}$
16      Adapt CMA-ES parameters $\boldsymbol{\mu}, \Sigma, \boldsymbol{p}$ based on improvement ranking $\Delta_i$
17      if there is no change in the archive then
18          Restart CMA-ES with $\boldsymbol{\mu} = 0, \Sigma = \sigma_g I$.
19          Set $\boldsymbol{\theta}$ to a randomly selected existing cell $\boldsymbol{\theta_i}$ from the archive
20      end
21  end
```

Fig. 2.14: Pseudocode of CMA-MEGA from Fontaine and Nikolaidis [4]

2.5.9.4 Importance of Normalizing Gradients in DQD

Fontaine and Nikolaidis [4] showed with experiments that using unnormalized gradients is likely to allow the objective gradients to dominate over behaviour gradients and for positions in feature space far ay from an optimum. This was observed with OMG-MEGA Emitter that when was used with unnormalised gradients, it filled cells near the global optimum but did not fill cells further below in the feature space to positions with lower objective values. In contrast, when it was used with normalized gradients, it could fill the archive more evenly.

Chapter 3
Domains

A QD problem requires an objective and a behavioural function. Each objective and behavioural function selection can define a different domain. Initially, we conducted our experiments on three toy domains (i.e. experimental domains with relatively simple functions) which we define below:

3.1 Rastrigin Function with a simple encoding

3.1.1 Objective Function – rastrigin(x)

We used the Rastrigin function (2.1.15); which takes as input a vector of n values; as an objective function with the additional modifications:

1. The Rastrigin function is shifted so that the optimal value is at x = 2.048. This is done to avoid having the global minimum point of the rastrigin function at position x' = [0,0,0,...,0], i.e. f(0,0,0,...,0) = 0 because the position x' = [0,0,0,...,0] is usually used as an initial point for the search.
2. We flip the function up-side-down so that the problem becomes a maximization problem (because Rastrigin is used in minimization problems), and we normalize its output value between 0 and 100 (inclusive) by considering that the maximum output it can have is 0 (best solution) and the minimum (worst solution) to be at the value of the Rastrigin function at -5.12 minus the shift (=2.048).

Refer to A.1.1 for the implementation of the objective function.

3.1.2 Behaviour Function – b_simple(x)

The behaviour function is defined using a simple encoding from a description of a solution (genotype) to the behaviour descriptor. From a genotype x described by a vector $< x_1, x_2, \dots, x_n >$ the behavioural function returns as a behavioural descriptor of the genotype a vector containing the first two entries of the genotype's vector; i.e. $< x_1, x_2 >$.

The partial derivative of the b_simple function with respect to a genotype vector $< x_1, x_2, \dots, x_n >$ is the Jacobian matrix

$$\begin{bmatrix} 1 & 0 & 0 \dots 0 \\ 0 & 1 & 0 \dots 0 \end{bmatrix}$$

Note that the boundaries of the behavioural space are [0,1] for each dimension of the behavioural space.

For the implementation of this function, please refer to section A.1.2 (Appendix A).

3.2 Rastrigin Function with Distorted Behavior Space

3.2.1 Objective Function – rastrigin(x)

The same as was defined in section 3.1.1.1.

3.2.2 Behaviour Function – b_distorted(x)

This behaviour function aims to introduce a high degree of distortion in the behaviour space. Distortion in a behaviour space is caused by dimensionality reduction from search space to behaviour space. The dimensionality reduction does not need to be complex. A linear projection from a high-dimensional search space to a low-dimensional behaviour space distorts the distribution of solutions in behaviour space pretty well. The behavioural function b_distorted(x) is defined with the help of a clip function:

$$clip(x_i) = \begin{cases} x_i \ if - 5.12 \leq x_i \leq 5.12 \\ \dfrac{5.12}{x_i} \ otherwise \end{cases}$$

And the b_distorted(x) is defined as:

$$\text{b\_distorted(x)} = \left(\sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} clip(x_i), \sum_{j=\lfloor \frac{n}{2} \rfloor + 1}^{n} clip(x_j) \right)$$

Note that the partial derivative of the function for any value of the input values in [−5.12, 5.12] is 1.

What makes this behavioural function more difficult for a QD algorithm to navigate in the behavioural space defined by it is that the behavioural descriptors of a solution are computed from all the parameters of the solution's descriptions and not a subpart of it like in the 3.1.2. In addition to this, when a solution description's parameters contribute equally to the definition of a behavioural descriptor (i.e. are equal), then a QD Algorithm needs to move in a direction in the search space, taking into consideration potentially updating all of the parameters so that to navigate to extremes in the behaviour space. That is, there are more parameters to be updated to navigate in the behaviour space. The difficulty becomes bigger as the size of the problem becomes bigger, i.e. the number of the Rastrigin function's dimensions increases. This can be seen from the observation that when sampling uniformly randomly solutions from the search space and projecting them to the behaviour space, each behavioural descriptor is the sum of n uniform random variables that, when divided by n for normalization between 0 and 1, result in the Bates Distribution shown in the figure below. This observation and the behaviour function were first introduced in Fontaine, Nikolaidis, Togelis and Hoover [6].

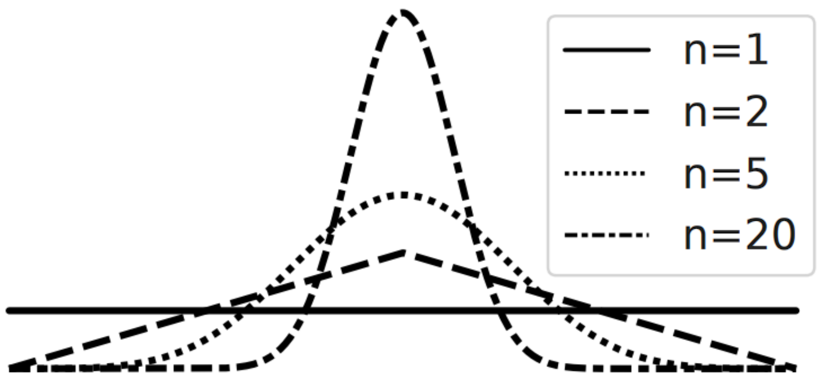


Fig. 3.1: A Bates Distribution as it changes with the number of random variables and shows the effect of narrowing behaviour spaces when formed by a linear projection from Fontaine, Nikolaidis, Togelis and Hoover [6].

With this function, the authors expected the CMA-ME Algorithm to perform better than MAP-Elites by better covering the distorted behaviour space.

Note that the boundaries of the behavioural space are $[-\frac{5.12\cdot n}{2}, \frac{5.12\cdot n}{2}]$ for each dimension of the behavioural space.

For the implementation of this function, please refer to section A.1.3 (Appendix A).

3.3 Arm Repertoire

Arm repertoire is a problem where an arm tries to move in space. The arm is fixed in one position, and it changes the angles of different joints to accomplish a goal.

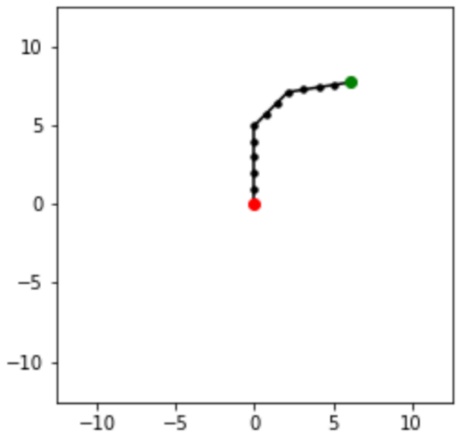


Fig. 3.26: Arm Repertoire visualization from the Pyribs Tutorial [65].

The arm has two sets of parameters, the lengths of each link between two joints and the angles of the joints. Consider the lengths of the links to be a vector v= <l_1, l_2, …,l_n> and the angles of the joints to be x= <θ_1, θ_2, …,θ_n>.

3.3.1 Objective Function – grasp_obj(x)

The goal of the objective function grasp_obj(x) is to minimise the variance of the joint angles. That is, to maximise the negative of the variance of the joint angles. It is defined as:

$$\text{grasp\_obj(x)} = 100 \cdot (1 - \frac{\sum_{i=1}^{n}(x_i - \mu)}{n-1})$$

where μ is the mean of the joint angles, note that the input parameters to the objective function are the angles of the joints, whereas the links are considered constants and are decided beforehand (in our experiments, we used unit-length links).

Note that the boundaries of the behavioural space are $[-s, s]$ where $s = \sum_{i=1}^{n} l_i$ for each dimension of the behavioural space.

The implementation of this function can be found in A.1.4 (Appendix A).

3.3.2 Behavioural Function – grasp_bds(x)

The behavioural function grasp_bds computes the final x and y coordinates of the arm's end-effector using the forward kinematics equations (https://en.wikipedia.org/wiki/Forward_kinematics):

$$x = l_1 \cos(\theta_1) + l_2 \cos(\theta_1 + \theta_2) + \cdots + l_n \cos(\theta_1 + \theta_2 + \cdots + \theta_n)$$

$$y = l_1 \sin(\theta_1) + l_2 \sin(\theta_1 + \theta_2) + \cdots + l_n \sin(\theta_1 + \theta_2 + \cdots + \theta_n)$$

An implementation of this function can be found in section A.1.5 (Appendix A).

Again, the links are considered constants and are decided beforehand (in our experiments, we used unit-length links). Thus, the input parameters to the function are the joint angles.

Chapter 4

Design & Implementation

4.1 Hardware Acceleration for Machine Learning

In the modern time, where vast amounts of data are easily accessible, Machine Learning and especially Deep Learning can be significantly benefited. The only request is the ask for speed. That is, the processing of a vast amount of data in a timely fashion. This is done with the use of specialized hardware accelerators.

In the mid-2000, Graphical Processing Units (GPU) began being used to train neural networks[59] because of their capabilities of high parallelization and high memory bandwidth. With this opportunity, the general-purpose GPUs started being introduced in every part of computation in Sciences, especially with access to specialized GPU compatible code for Deep Learning methods [60]. This enabled a few orders of magnitude faster training of networks on general-purpose GPUs than on CPUs.

To exploit the incredible power of GPUs, accessible and efficient frameworks have been developed, such as Tensorflow[61], Torch [62], Pytorch[63] and Jax[64], to allow running different numerical computations on GPUs efficiently. The access to such frameworks caused an explosion in the development and research conducted on Machine and Deep Learning and allowed people without any experience in how to use GPUs to build fast ML and DL models.

Based on these, our decision was to use the JAX to implement our QD Algorithms and conduct our experiments.

4.2 JAX

4.2.1 Introduction to JAX

JAX is a Machine Learning framework developed for bringing together Autograd (library for automatically differentiating native Python and Numpy code) and XLA (Optimising Compiler for accelerated Linear Algebra and Machine Learning used with TensorFlow) and making them available to be used on specialized accelerators, i.e., GPUs and TPUs. Even though it is developed for specialized accelerators, JAX also supports CPU as well.

The two main powers of JAX are the ***jit*** (just in time compilation optimiser) and *grad* (auto-differentiation of methods) that are developed in accordance with XLA and Autograd, respectively.

4.2.2 JAX JIT

As we mentioned, JAX uses XLA to compile and run NumPy programs on GPUs and TPUs. JAX's compilation for the specialized accelerators happens under the hood by default, with library calls getting just-in-time (jit) compiled and executed.

These just-in-time compilations performed by JAX are also available to be performed by a user. JAX provides the one-function API for jit to just-in-time compile Python functions into XLA-optimized kernels.

Note that code compiled with jit is always executed on the specified device that we configured JAX to run with. Thus, all data used by the optimised code (passed as arguments to a jit compiled function) are transferred to the corresponding device. This means that the overhead of moving the data to the device needs to be taken into consideration when developing code for JAX.

4.2.3 JAX GRAD

In addition to optimizing code with the JAX compiler, JAX provides an API interface to transform functions. One transformation is the Automatic Differentiation. That is, JAX provides, among many others, the function ***grad*** that can compute gradients of functions.

JAX's ***grad*** is used to compute the derivatives of functions that can take as input any tree-like data structure but which return a scalar value. More advanced autodiff methods for allowing the computation of gradients on methods that return multiple values, JAX provides the ***jacfwd*** and ***jacrev*** for forward-mode jacobian-vector computation and reverse-mode jacobian-vector computation, respectively.

4.2.4 JIT Constraints

JAX provides a set of amazing capabilities for helping you develop efficient algorithms that are able to run on many devices (GPU, TPU and CPU as well) with absolutely no change to your code. The capabilities come to the point that even JAX optimized code for CPUs can run faster on CPUs than similar code that is developed to run only on CPUs. As you might understand, this power comes with a cost. Developing algorithms on JAX is not the same as normally developing code in python. In order for JAX to optimize your code for a specific device (i.e., using just-in-time compilation), JAX requires a set of constraints in your python code to be true in order to optimize it. Let's explore these constraints by quickly overviewing the JIT mechanisms.

Jit works by tracing your code and by depending on the values of static variables. That is, there are two main concepts of how jit works:

1. Jit (as well as other function transformations) works by tracing a pure Python function (all data the function uses become available as arguments and all its changes appear only on the outputs – no side-effects) to determine its effect on the inputs
2. The inputs of a jit compiled function need to be of a predetermined shape and type (i.e. static)

In addition to that JAX-JIT does not allow the use of conventional python if-statements due to the difficulty of tracing multiple control flows in JAX compile-time. These constraints are required so that by tracing the code and recording the exact sequence of operations to be performed (jit compiling), we can use the same sequence of optimized operations efficiently within XLA but with different inputs.

4.2.5 Static vs Traced Operations & Data

Static Data are the data that are available at compile time of a function with XLA, whereas traced data are the data that become available at the run-time in XLA.

In the same way, static operations are evaluated at compile-time in Python, whereas traced operations are compiled and evaluated at run-time in XLA. The distinction of static vs traced operations in code is supported by the introduction of the JAX's implemented library of NumPy, which makes available most of the NumPy methods but as traced operations. That is, someone should use JAX's NumPy when they want traced operations in JAX and NumPy for static operations.

To understand the distinction between those two things, consider the following example:

```
import jax
import numpy as np
import jax.numpy as jnp

def my_funct(x):
    return x.reshape(jnp.array(x.shape).prod())

x = jnp.ones((2, 3))
# Optimise the my_funct and execute it
jax.jit(my_funct)(x)
```

Code Snippet 4.1: Example of error from JAX-JIT compilation due to the creation of an array with a non-static shape

The code above will not work because at Python compile-time, the shape of x becomes available to method *my_funct*, but because of the use of ***jax.numpy.prod*** instead of ***numpy.prod*** the outcome of the ***jax.numpy***.prod will give a traced value and not a static one; thus, the function will not know what to do.

If we replace jnp.prod with np.prod it will work because np.prod is a static operation, and thus it outputs static values and not traced values when used within jit compiled code.

The following code will work:

```
import jax
import numpy as np
import jax.numpy as jnp

def my_funct(x):
  return x.reshape((np.prod(x.shape),))

x = jnp.ones((2, 3))
```

```
# Optimise the my_funct and execute it
jax.jit(my_funct)(x)
```

Code Snippet 4.2: Static Operations that can be JAX-JIT compiled

4.2.6 JAX Classes, Objects and Pytrees

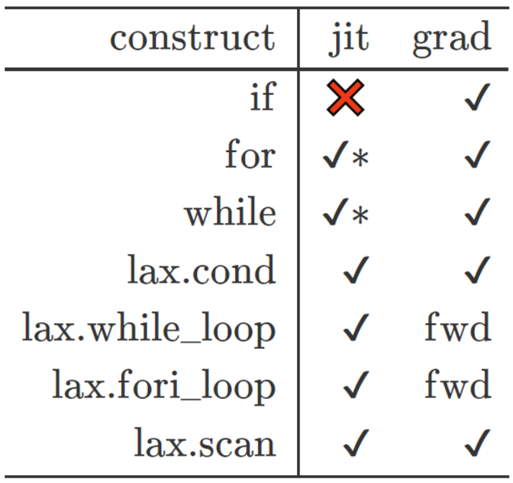
As we mentioned above, JAX Jit compilation operates on pure functions, takes a set of inputs and produces a set of outputs. Any set of side-effects like accessing or changing global variables may cause the optimized code not to work properly.

At the same time, JAX optimized functions cannot receive and operate on any kind of Object. The elementary data they accept are numerical data (i.e. integers, floating-point numbers) but not strings. In addition to that, they can receive any tree-like data structure built out of container-like Python objects. These data structures are called PyTrees, and they include lists, tuples, and dictionaries. In general, there are two rules for creating PyTrees: (1) an object that is not in the PyTree container registry of supported PyTrees is considered a lead PyTree, (2) any object that is in the PyTree supported registry of accepted PyTrees, and which contains PyTrees is considered a PyTree. Last, any other object that is passed as an argument and used in a JIT-compiled code needs to be a data class (i.e. flax.struct.dataclass). The definition of data classes allows someone to define a class that can instantiate objects that are more considered primarily as objects carrying data rather than carrying operations. Modifying a data class in the same way as modifying attributes of a Python object is not permitted. The way to do that is by using the method *replace* on the data class object in order to change the desired attribute. Objects of a data class can also have instance methods but remember those methods cannot have side effects on the objects. But, dataclasses are immutable. Thus, most of the time, it is more desirable to define and use static methods of objects rather than instance methods.

4.2.7 JAX control flow

Function transformers that implement Auto-differentiation allow any control flow operation to be used in the function. On the other hand, though, JAX JIT-compiled code cannot use if-statements, nor for-loops and while-loops, unless those control flow operations can be determined statically, i.e. with static data at JIT, compile-time.

More specifically, JIT optimization traces the output of a function to its inputs with the goal of identifying an exact sequence of operations to be done. The if-statement splits the flow of a program in two different directions. Tracing two different control flows can lead to an exponential number of different outputs at the end, which would make compiling a function impractical because of requiring a great amount of time and computer memory.



construct	jit	grad
if	✗	✓
for	✓∗	✓
while	✓∗	✓
lax.cond	✓	✓
lax.while_loop	✓	fwd
lax.fori_loop	✓	fwd
lax.scan	✓	✓

Fig. 4,1: Operators that can be used in JIT-compiled functions and GRAD transformed functions. Note that * denotes that it is possible to use the loop only if the number of iterations is statically determined. The loop will be unrolled to the contents of the loop's body as many times as the number of iterations. Taken from JAX’s official website tutorials

On the other hand, loops like for-loop and while-loop are allowed only if the number of iterations of the loop can be determined at JIT compile time. Even in that case, JIT compiling code that includes *for-loops* and/or *while-loop* takes a lot of time due to unrolling the loop as many times as the number of iterations and which consequently can lead to inefficient code.

To replace the conventional control-flow statements, JAX introduced methods that can do approximately the same work as Python’s control flow, with the only cost that they will be potentially performing more operations to accomplish them. More precisely, if-statement in JIT-compiled code is accomplished with *lax.cond* method, which can be considered equivalent to the function defined by the pseudocode below:

```
def cond(pred, true_function, false_function, operand):
  if pred:
    return true_function(operand)
  else:
    return false_function(operand)
```

Code Snippet 4.3: The meaning of cond function in JAX

The only consequence of using this method is that it leads to executing both the true and the false functions provided as arguments. This might not be a problem even for expensive operations because of the optimizations performed, but sometimes it might be a drawback of performing a significantly more number of operations than a normal python code. To avoid such cases, it is recommended that the lax.cond method does not use too time-demanding functions for the true and false functions, in order not to see any performance degrading.

In the same manner, JAX provides the lax.while_loop for the Pythonic while-loop and the lax.fori_loop for the Pythonic for-loop, implemented with the low-level lax.scan operation.

4.3 Developing Optimized and Efficient QD Algorithms on JAX

The implementation of the QD Algorithms we used in our experiments was done using the JAX to allow our algorithms to be executed on GPUs. To do that, we wanted our algorithms to be developed satisfying the following requirements:

1. Minimize the overhead of moving data between CPUs and GPUs
2. Use a custom-made framework to implement QD algorithms using the ask-tell interface from Pycma [66]
3. Improve Emitters' efficiency for GPUs

Below, we discuss the design and implementation decisions that led us to create efficient implementations of QD Emitters on JAX and to create a new framework for developing efficient QD algorithms on JAX.

4.3.1 Minimizing the overhead of moving data between CPU and GPU

To accomplish minimizing the time cost of moving data from CPU to GPU and vice versa, we decided to minimize the number of times that we move data between GPU and CPU by adopting two tactics:

1. The core logic of an iteration of a QD algorithm will be executed on GPU only (Minimize the number of times data are moved between CPU and GPU)
2. All of the data that a QD algorithm will use at a specific iteration on the GPU will be available to the GPU from the beginning of the iteration (Move all the required data to the GPU from the beginning of the iteration), and no other data transfers will take place for the rest of the iteration.

The last tactic allows iterations on GPU after the first one to use the already output data from the previous iteration and thus the algorithm can run all the iterations on GPU without any interrupt.

4.3.2 A new framework for building QD Algorithms with the ask-tell interface

Our algorithms on JAX were implemented using the ask-tell interface from Pycma [66]. More specifically, a QD Algorithm or Emitter provides two methods:

```
def ask(static_settings, emitter, repertoire, key)
```

Code Snippet 4.4: Method Signature of the ask method for a QD Emitter

```
def tell(static_settings, emitter, solutions, objective_values, behavior_values,
dead, repertoire, key)
```

Code Snippet 4.5: Method Signature of the tell method for a QD Emitter

Where static_settings is a named-tuple with static settings used to define complete arguments of a method before compiling the method (the idea will be explained below), the emitter is a data class holding data for the QD emitter; solutions is an array of solutions, objective values are the values as derived from an objective function on the solutions, behavioural values are the behavioural descriptors as derived from the behavioural function on the solutions, dead in an array denoting whether a solution should be ignored, the repertoire is the container where the solutions are stored, and the key is the Pseudo-Random Number Generator State of a JAX PRNG instance (used for generating a sequence pseudo-random numbers).

A DQD Emitter, in addition to the two method signatures above it, also has another two methods

```
def ask_grad_estimate(static_settings, emitter, repertoire, key)
```

Code Snippet 4.6: Method Signature of the gradient ask method of DQD Emitter

```
def tell_jacobian(static_settings, emitter, solutions, objective_values,
behavior_values, dead, repertoire, key, jacobian)
```

Code Snippet 4.7: Method Signature of the gradient tell method of a DQD Emitter where the jacobian is a matrix holding the jacobian matrix of the behaviour function for each of the solutions given and the gradient vector of the objective function for each of the solutions given.

The reason for using two different ask-tell pairs of methods for the DQD Emitters is because, as you remember, the DQD Emitters (e.g., OMG-MEGA (iso) section 2.5.9.3.1.1) have two evaluations of solutions that happen per iteration, one with the sampled solutions from the archive and second the gradient perturbation on the sampled solutions. Thus, two sets of solutions are generated and, consequently, two different additions to the container per iteration.

4.3.2.1 The need for an object-oriented implementation of QD Emitters in JAX

The implementation of those Emitters' methods (QD and DQD) on the CPU is usually done in an object-oriented way. That is, an Emitter is an object carrying the necessary data and providing those methods as instance methods. But, as we said above in section 4.3.1, we want all of those methods to be executed in a JIT optimized code. We know, though, that JAX implementation and object-oriented programming cannot exist together in the form that we know them.

Thus, to address this issue and accomplish the two requirements aforementioned, we defined a new framework for building QD Algorithms or Emitters in JAX

4.3.2.2 Building Python-like Objects from Data-classes

The idea of the new framework is to use the data classes (i.e. flax.struct.dataclass) as a Classes of objects but eliminate side effects. I.e. let's say that we want to create a Person class with a single method to set and get the height of a person. In normal python, we could do that like the following code:

```python
class Person:
    def __init__(self, height):
        self.height = height

    def get_height(self):
        return self.height

    def set_height(self, height):
        self.height = height
```

Code Snippet 4.8: Person class in python

Instead in JAX we could do define the class above in the following way:

```python
import flax

@flax.struct.dataclass
class Person:
    height: int

    @classmethod
    def create(cls, height):
        return cls.__init__(height)

    def get_height(person):
        return person.height

    def set_height(person, height):
       person = person.replace(height = height)
       return person
```

Code Snippet 4.9: Person class as a DataClass

We can see that the get methods do not change unless they update the state of the data class, whereas the set methods change by using the replace method of a data class to change its state, and also they return the new altered object back since each data-class is immutable. Last, the __init__ method of a data class is already defined and cannot be overwritten. It requires passing all the data elements of the data class. To be able to define our own init method, we define the create a class method that allows us to perform any operations before we create a data class and which can create any number of arguments we want.

Thus, there are three important things when creating a data class from a Python Object.

1. Define the data attributes of the data class; i.e. include the data elements (=attributes) of the object at the top of the class (below its name definition) with their appropriate type
2. Use a class method to define your own __init__ (e.g. create)
3. Each method should that alters a data-class object passed as an argument should return back the altered data-class object

Note that because data-class objects are immutable, no other attribute can be added dynamically to the object during run-time.

Despite this ease of converting approximately any Python class to a Data-class, we haven't talked about how those objects can actually be used in JIT optimized code. How can we develop complex QD Algorithms with such objects?

4.3.2.3 Implementing QD Emitters in JAX

Most of our programs and algorithms use different types of control-flow statements that are really useful and important. But at the same time, they are expensive as computer operations compared to simple arithmetic operations. Unfortunately, most of the time, they cannot be removed from our code or expressed to avoid them. The same thing applies to QD Algorithms. We cannot avoid using if-statements, but what we can do is to separate the logic and the if-statements that depend on the initialization of the algorithm with the core logic of the algorithm.

The last part that concludes our framework and which defines how complex object-oriented QD Algorithms (from CPU) can be implemented in JAX and in such a manner to be more time-efficient depends on the observation that most of the QD algorithms' control-flow statements depend on values of parameters that are set at the time of the object's initialization and then they are never changed. User-defined parameters that define what the QD algorithm will do usually do not change. Thus, it is clear that since the JIT compilation can be done manually in runtime (using jax.jit), we could isolate the attributes of an object that are defined in its initialization and never change from the rest of the attributes and make available their values to the methods that we want to JIT-compile before they are jit compiled. To make this idea clearer, let's say that in the example of the Python Person Class above, we had an additional field for the person denoting whether the person is alive or dead and also an if-statement in set_height checking that the person is alive and then updating the Person object, i.e.,

```python
class Person:
    def __init__(self, height, is_alive):
        self.height = height
        self.is_alive = is_alive

    def get_height(self):
        return self.height

    def set_height(self, height):
        if self.is_alive:
            self.height = height
```

Code 4.10: Person class in python that uses if-statements

If we were going to keep the if-statement in the Data-Class implementation of the Person class (look at the code snippet below), and then JIT compile the set_height method as follows:

```
jax.jit(person.set_height)(person,178)
```

Code Snippet 4.11: JIT Compiling the set_height method of Person data-class in JAX. The method uses an if-statement that is not statically determined, and thus the method won't be able to be optimized

we would get an error from the JAX JIT Compiler because the if-statement JAX doesn't know which side of the if-statement to execute when it is executed by the XLA (after its compilation).

```
import flax

@flax.struct.dataclass
class Person:
    height: int
    is_alive: int # we use int instead of bool because JAX JIT compilation does not
                  # accept as an attribute of a data-class

    @classmethod
    def create(cls, height, is_alive):
        return cls.__init__(height, is_alive)

    def get_height(person):
        return person.height

    def set_height(person, height):
        if person.is_alive == 1:
            person.replace(height = height)
        return person
```

Code Snippet 4.12: Person data-class in JAX that uses if-statements (method set_height will not be able to be optimised)

But we can clearly see that once the Data-class instance of Person is initialized, the is_alive attribute never changes and thus, all the if-statements depending on the is_alive could be replaced with the one or the other sides of the if-statement (depending on the value of the is_alive in the specific if-statement).

To accomplish this change to the code after the initialization of an object, we change the signature of the method to receive the is_alive as an additional argument and then provide the argument is_alive to the method set_height before it is compiled. Doing so, and thus compiling the new method with the argument, is_alive determined, the if-statement in the set_height can be statically determined in runtime (JIT compile time). More specifically, this would be the new signature of the set_height method:

```
    def set_height(person, is_alive,  height):
        if is_alive == 1:
            person.replace(height = height)
        return person
```

Code Snippet 4.13: Redefining the method set_height of the Person data-class in JAX

And below, you can see the use of functools.partial that creates a new method from a method given by partially filling parameters of the function given with arguments. In this case, we created a new set_height method that takes as an argument the person's data-class object and the height (since the is_alive was "embedded" in the new method).

```
jax.jit(functools.partial(person.set_height, is_alive = person.is_alive)) (person,
178)
```

Code Snippet 11: Compiling the method set_height of a data-class object of the Person data-class in JAX by providing the argument is_alive that determines the outcome of the if-statement used in the method

Because of the fact that an object can have many such parameters that can be statically defined, we enforce this idea to the whole data class with the following class:

1. The method for creating and initializing a data-class instance returns; in addition to the data-class instance, a data structure of static settings containing a set of parameters and methods that are used by any method of the data-class instances to statically determine control flows, dynamic shapes of arrays etc. (dynamic operations and data) before being compiled by JAX JIT.
2. Each method of the data-class receives as an additional argument, the static settings

Thus, a method *f* of an object *o* of a data-class defined in the way above will be compiled and executed in the following way.

```
jax.jit(functools.partial(o.f, static_settings)) (…)
```

Code Snippet 4.14: Compiling and Executing a method of the QD Emitters framework

For completeness of our explanation, consider below the implementation of the Person Python Class as a Data-class, that each of its methods (except the one used for initialization – create method) can be JIT-compiled when before that, the static_settings data structure is provided.

```python
import jax
import functools
import flax
from collections import namedtuple

@flax.struct.dataclass
class Person:
    height: int
    is_alive: int # we use int instead of bool because JAX JIT compilation does not
                  # accept as an attribute of a data-class

    @classmethod
    def create(cls, height, is_alive):
        # define the static settings as a namedtuple
        static_settings = dict()
        static_settings['is_alive'] = is_alive
        StaticSettings = namedtuple('StaticSettings', static_settings)
        ssd = StaticSettings(**static_settings)
        return ssd, cls.__init__(height, is_alive)

   # private method used to access specific attributes from the static settings
    def _is_alive(ssd):
        return ssd.is_alive

    def get_height(person):
        return person.height

    def set_height(ssd, person,  height):
        # use the static_settings’ attributes via private methods
        # specifically defined to access its attributes
        if _is_alive(ssd) == 1:
            person.replace(height = height)
        return person

ssd, p = Person.create(160, 1)
jax.jit(functools.partial(p.set_height, ssd))(p,178)
```

Code Snippet 4.15: Redefining the method set_height of the Person data-class in JAX

4.3.3 Improve Emitters’ efficiency for GPUs

In the previous section (4.3.1), we talked about how we can transform object-oriented QD Emitters implemented in Python (for CPU) to code that can be JIT-compiled in JAX. The use of JIT compilation optimizes our Emitters for GPU. An additional step that we took to create more efficient implementations of QD Algorithms is that we transform inefficient code into efficient code. More specifically, we converted loops to vectorized approaches, where instead of using for-loops to operate over arrays, we used NumPy operations over arrays. Doing so allows for better parallelization and memory usage in GPUs.

Another change we made to our implementations that improved the performance of the QD Emitters is the minimization of the use of JAX's structured control-flow functions (i.e. lax.cond, lax.fori_loop and lax.while_loop) in our code by replacing many of them with arithmetic vectorised operations that could lead to the same outcome.

4.4 Implementations of QD and DQD Emitters on Jax

We have implemented all of the available QD Algorithms from Pyribs in JAX. We created two versions for each of them, the *Array Version*, where the Emitters require that a solution is strictly a one-dimensional array and the *PyTree version*, where the Emitters can work with solutions of any PyTree structure. These implementations are:

- Gaussian Emitter implements MAP-Elites (iso) (Code for the Array Version is found in Appendix A's section A.3.1 and for the PyTree Version in Appendix A's section A.5.1)
- Iso Line Emitter implements the MAP-Elites (iso+lineDD) (Code for the Array Version is found in Appendix A's section A.3.2 and for the PyTree Version in Appendix A's section A.5.2)
- Improvement Emitter implements CMA-ME (Improvement) (Code for the Array Version is found in Appendix A's section A.3.3 and for the PyTree Version in Appendix A's section A.5.3)
- Optimizing Emitter implements CMA-ME (Optimizing) (Code for the Array Version is found in Appendix A's section A.3.4 and for the PyTree Version in Appendix A's section A.5.4)

- Random Direction Emitter implements CMA-ME (Random Direction) (Code for the Array Version is found in Appendix A's section A.3.5 and for the PyTree Version in Appendix A's section A.5.5)

All the QD Emitters operate on a batch of solutions called batch size or population size instead of a single solution. Thus, they generate a number of solutions at each iteration proportional to the batch size selected for them.

4.5 Optimizers

We have implemented in JAX three Optimisers that are used by the QD and DQD Emitters. These are:

- Covariance Matrix Adaptation Evolution Strategy (Appendix A - Section A.2.1)
- Adam Optimiser (Appendix A – Section A.2.2)
- Gradient Ascent Optimiser (Appendix A – Section A.2.3)

4.6 Containers

We have adapted the N-dimensional Grid Archive implementation for our own purposes from the Lim, Grillotti, Allard and Cully [69]. The implementation of our own version can be found in Appendix A, section A.7.1.

Chapter 5

Experiments, Results, and Discussion

5.1 Introduction to Experiments, Results, and Discussion

In this section, we describe the details of the experiments that we performed to investigate the three goals that we defined: (a) Effect of batch size on the Performance of QD and DQD Algorithms, (b) Runtime of QD and DQD Algorithms on GPU compared to CPU and (c) Performance of DQD Algorithms vs QD Algorithms on GPUs.

5.2 Methodology & Design

5.2.1 Experiment Design

5.2.1.1 Independent Variables

The independent variables of our experimental design are:

1. The QD and DQD algorithms MAP-Elites (iso), MAP-Elites (line), CMA-ME (imp), CMA-ME (opt), CMA-ME (rd), OMG-MEGA (line), OG-MAP-Elites (line), CMA-MEGA (Gradient Ascent) and CMA-MEGA (Adam)
2. The domain or type of problem; i.e. Rastrigin with Simple Encoding, Rastrigin with Distorted Behavior Space and Arm Repertoire. We did not have the time to include results on more complex simulation domains.
3. The batch size used by the QD and DQD Algorithms
4. The problem size (also seen as Solutions Dimensions) for each type of problem

5. The device on which algorithms are executed (GPU or CPU)

5.2.1.2 Dependent Variables

For our experiments, we measure the diversity and the quality of the solutions returned by each algorithm via the Coverage, Best Fitness and QD Score metrics. We also measure the speed of execution of each algorithm via their runtime. These metrics are described below.

5.2.1.3 Methodology & Experimental Details

Our experiments considered the number of evaluations for the DQD algorithms, except CMA-MEGA variations to be twice as much as that used in QD algorithms for the same number of iterations and the same batch size. This decision is because, at each iteration, all the DQD algorithms perform two evaluations (one with the solutions derived from the container and one with the gradient amended solutions derived from those initially generated solutions). Thus, considering that we perform two evaluations at each iteration of a DQD algorithm, we chose to keep the number of epochs of the DQD algorithms in half so that the number of evaluations of the QD and DQD algorithms is the same and thus, their performances comparable.

All the algorithms were executed for 5 million evaluations on various batch sizes, types of problems, problem sizes, and devices. Their performance (metrics) was recorded on a per-iteration basis. More specifically, all the QD and DQD algorithms were executed on both GPUs and CPUs. The algorithms executed on CPUs were tested on the batch sizes: 512, 2048, 8192 and 16384, and problem sizes: 128, 256, 512, 1024 and 2048 for all the different types of problems. The algorithms executed on GPUs for types of problems (domains) were tested with batch sizes: 512, 2048, 8192, 16384, 32768 and 131072 and problem sizes: 128, 256, 512, 1024 and 2048. For each of those times, the algorithms were executed 10 times and the median was considered the representative of all the 10 executions (median absolute deviation was considered for showing any deviation).

For each problem type examined, the parameters of each algorithm selected (e.g. standard deviation of distributions used) were the best across a set of indicative parameters covering all the possible combinations of good parameters. For example, for the standard deviation of distributions used in the QD and DQD Algorithms, I tested all the values between 0 and 0.1 with step 0.01, all the values between 0.1 and 1 with a 0.1 step, and the values between 1 and 20 with a 5 step (i.e. 1, 5, 10, 15 and 20). By best parameters, we mean in terms of the Algorithm's accomplished QD Score when using those parameters. Those parameters can be found in Appendix B.

The code used for finding those best parameters and executing the algorithms for the experiments can be found in section A.8 of Appendix A.

5.2.2 Metrics

For our experiments, we will use the Coverage Score, the Best Fitness, the QD-Score and the Runtime in terms of seconds to measure the performance of each algorithm and have comparisons.

Given a collection of t bins $\{b_1, b_2, .., b_t\}$ of the container used in a QD Algorithm, each with performance scores (objective values) $\{O_1, O_2, .., O_t\}$ (where $O_i\ is$ 0, if there is no solution stored in b_i otherwise, it is nonzero) the coverage is defined as the number of bins that have a solution inside ($O_i\ not\ zero$). The Best Fitness is defined as the maximum objective value held by any solution in the container, i.e. max$\{O_1, O_2, .., O_t\}$ and QD-Score is the sum of all the solutions' objective values that are stored in the container; i.e. $\sum_{i=1}^{t} O_i$.

5.2.3 Hardware

The CPUs that we used for our experiments were the Intel(R) Xeon(R) Gold 6240 CPU @ 2.60GHz, and the GPUs that we used were the Tesla V100 32GB. All the algorithms executed with GPUs used a single GPU device (i.e. Tesla V100 32GB), and all the algorithms executed on CPUs used a single CPU with eight cores. Parallelization on the GPU and the CPU was managed via the JAX framework. We did not manually use commands like *pmap* to map operations to different cores or devices. The latter was done to avoid extra costs when moving data between cores/devices.

5.2.4 JAX Configurations

For our experiments, we used version 0.2.26 of JAX with the CUDA library cuda11_cudnn805. We also used XLA pre-allocation of 80% of the memory on GPUs to save time allocating memory during runtime.

5.3 Results & Analysis

Before we dive into the primary goal of this work, i.e. “Does GPU and Differentiability help speed up the performance of QD Algorithms?”, we shall try to examine whether QD and DQD Algorithms continue to perform well when we increase their batch size (also called population size denoting the number of solutions being handled simultaneously by a QD Algorithm). To our knowledge, examining the effects on QD Algorithms when increasing the batch size they work with hasn’t been investigated so far on all the QD and DQD algorithms. There is a small work on the effects of big batch sizes on two variations of MAP-Elites (MAP-Elites iso+lineDD and MAP-Elites iso) on GPUs, which showed that MAP-Elites with the iso+lineDD operator seems to be stable in terms of their performance (i.e. QD Score) on greater batch sizes. Here we will work with all the state-of-the-art QD and DQD Algorithms, and we will examine the effect on their performance not only when changing their batch size but also when increasing the size of the problem on which they operate.

Changes in the problem size and/or batch size are an essential matter that needs to be investigated to allow better exploitation of parallelization. We need to understand that it is a complex problem since there are a lot of direct and indirect factors that might come to play and affect the performance of QD and DQD Algorithms while examining them with bigger batch and problem sizes. For example, the state-of-the-art QD and DQD Algorithms generate a collection of solutions (based on the configured batch size) at each iteration, and they all use randomness in their selection and variation operations. Due to this randomness, there is a chance that the same solutions are generated either at the same iteration (i.e. two or more solutions from the collection of solutions generated are the same) or at a different iteration. When the batch size is increased, the number of solutions generated at each iteration increases and thus, it is more likely that the same solution will be generated at a specific iteration. Suppose the algorithm generates a significant number of already discovered solutions repeatedly. In that case, it doesn’t exploit the different evaluations efficiently (i.e. it wastes the time of evaluating the same solutions again and again) and thus becomes unattractive to be used.

In addition to the batch size, the problem size or the solutions’ dimension could appear to affect the performance of the QD and DQD algorithms. For example, increasing the problem size could make generating the same solutions at each iteration less likely to appear due to the many parameters of each solution being changed using some randomness (in the variation operations). At the same time, increasing the dimensions of a solution being used could make feature space exploration more difficult, especially in feature spaces that depend on the dimensions of their solution (i.e. problem size).

A different factor that might affect the performance of the QD Algorithms while examining them with bigger batch sizes is the type of problem or task that we use. More specifically, a QD Problem defines an objective function and a behavior function. They define the performance and the behavior descriptor of a solution, respectively. Changing them can provide completely different performances and behavior descriptors. For example, a high-performing solution in one task can be a low-performing one in a different one. Or a solution being mapped in the center of the behavior space for a specific task can be mapped at the edges of the feature space. Thus, QD Algorithms can perform better in specific QD Tasks and worse in others.

These and other factors can affect the performance of QD Algorithms when we change the batch size and problem size. Thus, in the first section, we will try to examine the performance of QD Algorithms on three different QD Tasks (Rastrigin with simple encoding, Arm Repertoire and Rastrigin with distorted behavior space) with different batch sizes and a different number of solutions' dimensions (=problem sizes). Our examination is not exhaustive but tries to identify some traits that can help further exploration of the benefits and drawbacks of using a greater number of batch sizes in QD Algorithms for speeding up the QD and DQD Algorithms.

5.3.1 Effect of batch sizes and problem sizes on the Performance of QD and DQD Algorithms

5.3.1.1 Rastrigin Function

The first task (=problem) we examined is the Rastrigin Function (Section 3.1.1) with simple mapping to behavioral space.

Initially, let's see how the problem size affects the performance of QD and DQD Algorithms. Below, in figure 5.1, you can see the QD Scores of all the QD and DQD Algorithms for the Rastrigin Problem when using batch size equal to 2048 for different problem sizes. Similarly, in figures 5.2 and 5.3, you can see the best fitness and coverage scores for the same case above.

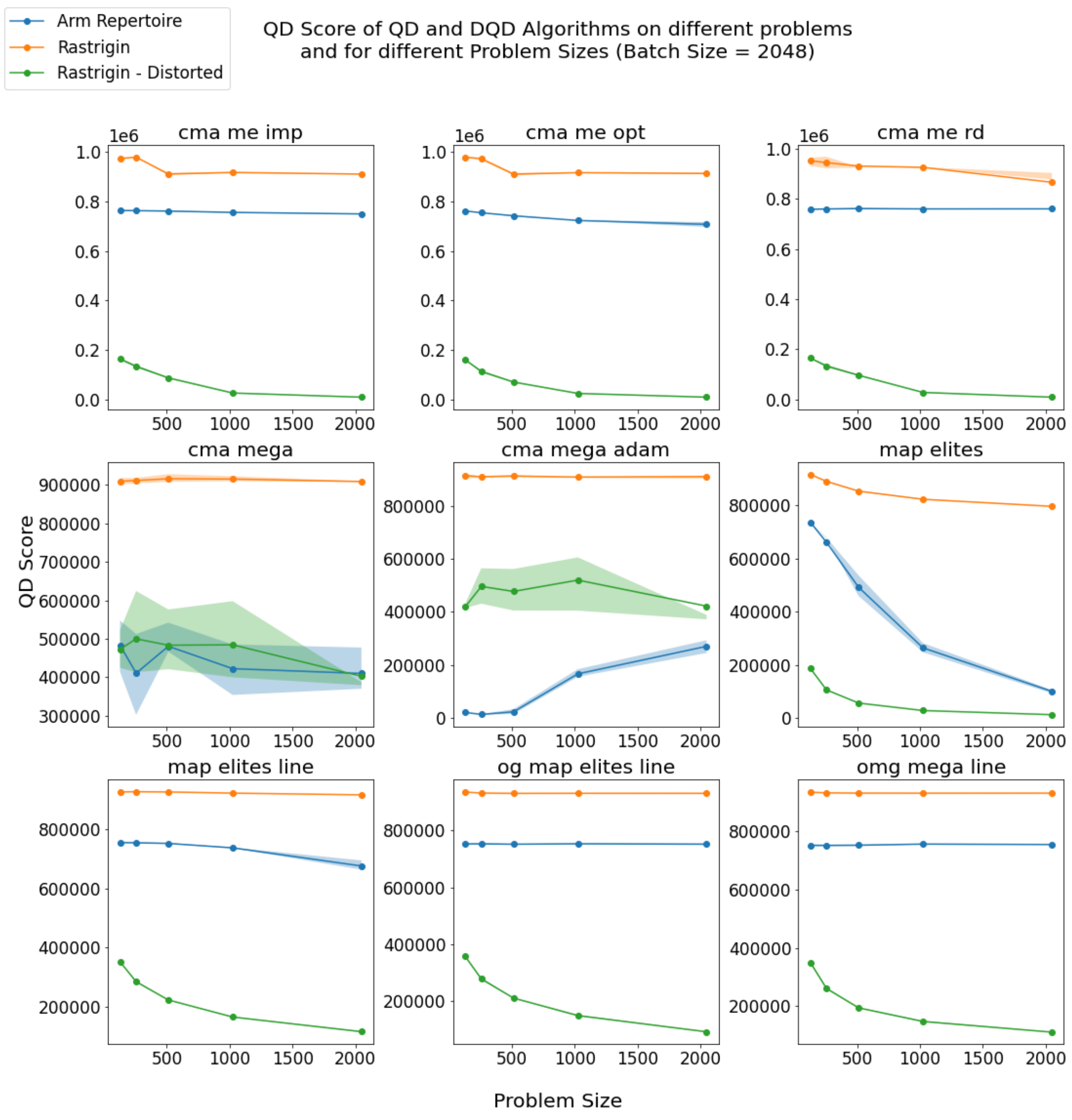


Fig. 5.1: QD Scores of QD and DQD Algorithms on different problems and for different problem sizes (Constant Batch size equal to 2048)

What we can see initially from the Coverage scores below (figure 5.3) for the Rastrigin problem is that all of the algorithms fill the Archive fully (≈10,000 coverage). Thus, the difference in the performances of the algorithms on this problem is on the quality (or fitness) of the solutions they store in their archive and not the number of cells they filled in the archive. Moving to the QD Scores (figure 3.1) above, **we can see that all of the QD and DQD Algorithms are affected negatively by the problem size increase**. Some of them are affected more (CMA-ME and MAP-Elites-iso), and some of them are affected less (CMA-MEGA, OMG-MEGA-line and OG-MAP-Elites-line, MAP-Elites (line)). By observing the changes in the best-fitness graphs (figure 3.2) for the Algorithms on the Rastrigin function, we can see that their best-fitness score decreases as the problem size increases. Therefore, the decrease of the QD Score as the problem size increases can be attributed to the decline in the quality of solutions stored in the archive when the problem size increases. Despite that the QD Score of all the QD and DQD Algorithms decreases, the algorithms whose performance decreases the least with the increase of the problem size for the Rastrigin problem are the CMA-ME, OMG-MEGA (line) and OG-MAP-Elites (line).

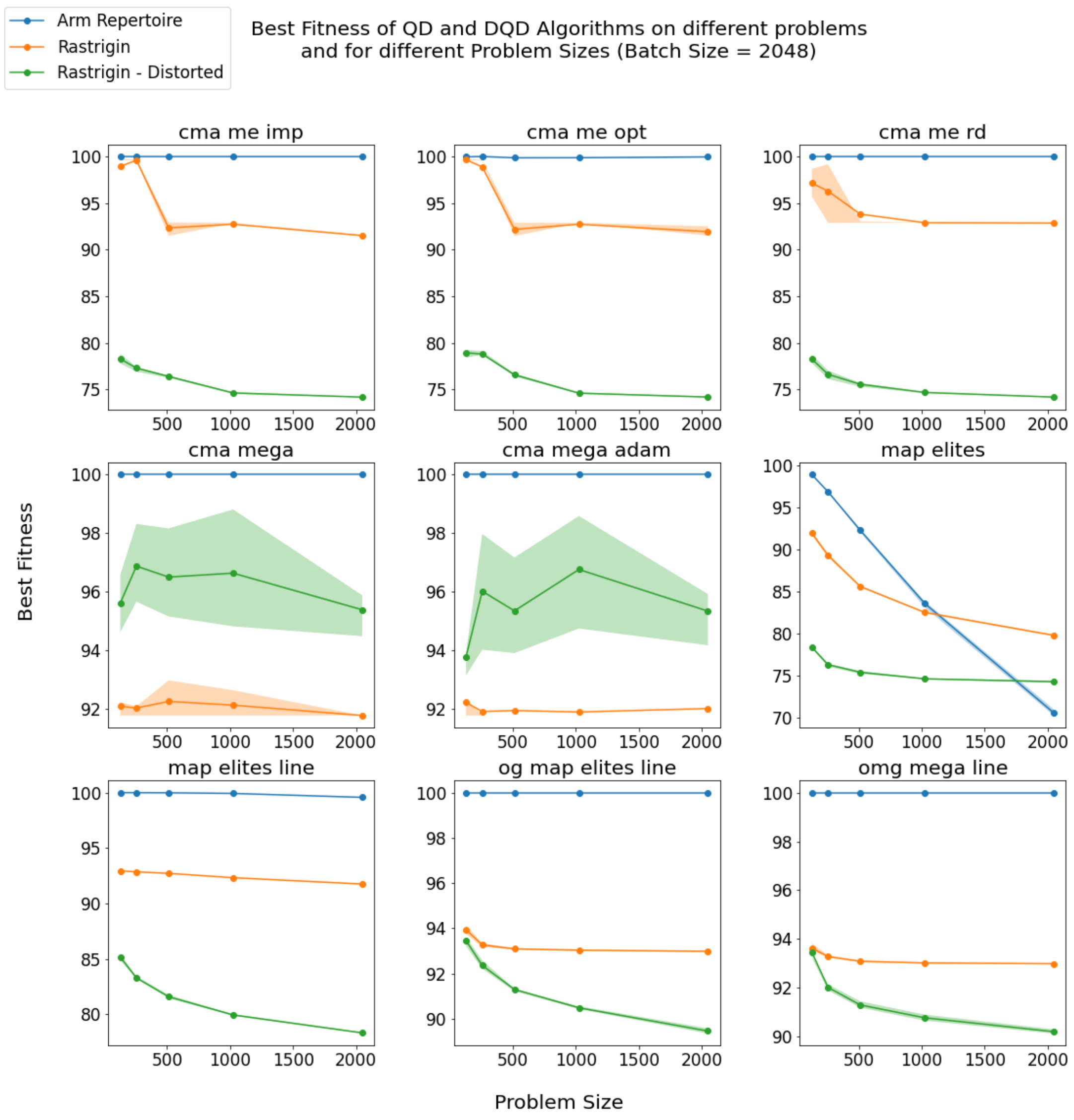


Fig. 5.2: Best Fitnesss of QD and DQD Algorithms on different problems and for different problem sizes (Constant Batch size equal to 2048)

From the figures below, 5.4 for QD Score, 5.6 for Coverage and 5.5 for Best Fitness, we examine the effect of increasing the batch size and keeping constant the problem size (=1024) for all the QD and DQD Algorithms examined on the Rastrigin function. From the QD Scores set of charts (figure 5.4), it is clear that **the increase in the batch size decreases the QD-Score performance of all the CMA-ME variations and MAP-Elites (iso) algorithms** on the Rastrigin problem significantly. **CMA-MEGA is affected slightly less than CMA-ME variations, but its performance still decreases with the decrease of the batch size**. Similarly, the QD Scores **of MAP-Elites (line) , OMG-MEGA (line) and OG-MAP-Elites (line)** algorithms decrease as the batch size increases, but the **decrease is minimal and the least compared to the reduction that appeared in the other mentioned algorithms**. This might seem counter-intuitive since someone might believe the statement we said before that greater batch sizes increase the number of times that we generate the same solutions. This could be true, but in fact, it is more unlikely for the MAP-Elites (line) to generate two or more solutions that are the same since the generation of each solution is done via a combination of two elite solutions, thus making it more unlikely to generate the same solutions twice (more unlikely, still possible though).

A few more observations that can be made are that CMA-ME algorithms perform really well for small problem sizes and small batch sizes. More specifically, for batch sizes equal to 2048 and problem sizes of 128 and 256, CMA-ME (imp) and CMA-ME (opt) had the best QD Scores found from any algorithm on the Rastrigin Function. But, as we keep the batch size constant and increase the problem size, their QD Scores are significantly reduced. On the other hand, increasing the batch size for all the CMA-ME variations decreases their performance significantly for all the problem sizes, but the interesting thing is that instead of performing better for smaller problem sizes in bigger batch sizes, just like how they performed in smaller batch sizes, they perform better in bigger problem sizes than in smaller problem sizes as the batch size increases.

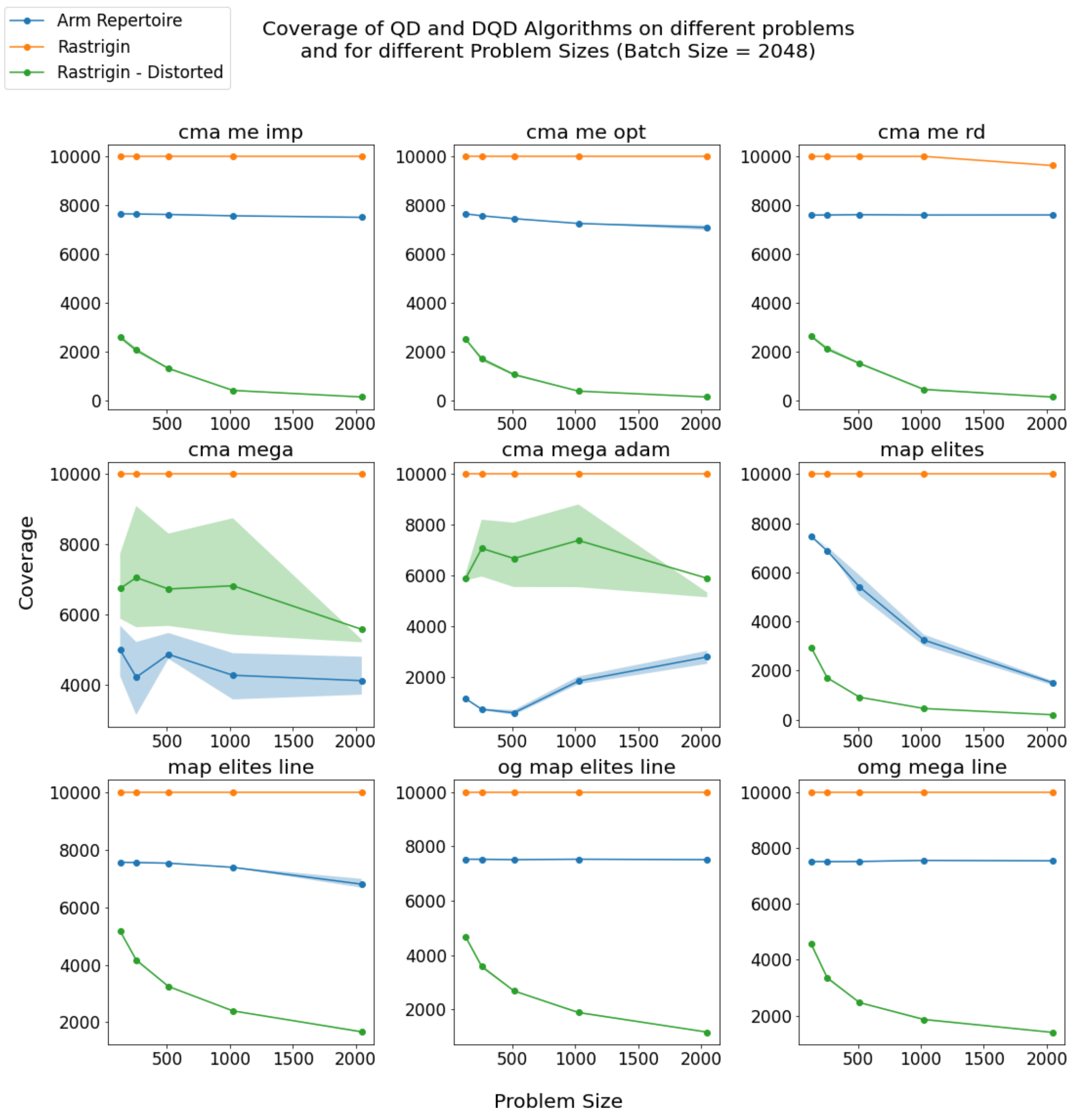


Fig. 5.3: Coverage Scores of QD and DQD Algorithms on different problems and for different problem sizes (Constant Batch size equal to 2048)

Last, the OMG-MEGA (line) and OG-MAP-Elites (line) have the highest QD Scores among all the other algorithms examined for almost all combinations of problem sizes and batch sizes examined. The latter two algorithms are affected the least among the QD and DQD Algorithms from the increase of the problem sizes on the Rastrigin problem. Following them, the next best QD Scores for most of the combinations of problem sizes and batch sizes examined are accomplished by MAP-Elites (line).

For the scores about combinations of batch sizes and problem sizes that we haven't included their charts here, there is no other difference than the things we mentioned. To see a complete list of those results, please refer to the Appendices.

In general, for the Rastrigin Task:

- Increasing the problem size decreases the performance of all the QD and DQD Algorithms in general (Best Fitness is primarily affected, which means that the quality of the solutions stored in the archive is decreased; less high-performant solutions are stored and more low-performant)
- Changes in problem and batch sizes affect coverage score the most (instead of the fitness of the solutions in the archive)
- MAP-Elites (iso) almost all of the time, gives worse results than all the other QD and DQD algorithms
- CMA-ME algorithms give really good results in small batch sizes, but the quality of their results becomes significantly worse with the increase in the batch size
- The quality of the results returned by CMA-MEGA variations decreases with the increase of the batch size, but that decrease is much smaller than that observed in CMA-ME algorithms
- The quality of the results returned by MAP-Elites (line), OMG-MEGA(line) and OG-MAP-Elites (line) remains approximately constant with the increase of the batch size (Stable with the increase of the batch size). Moreover, with the increase of the problem size their performance in terms of QD Score decreases the least.
- OMG-MEGA (line) and OG-MAP-Elites (line) perform better most of the time than all the other QD and DQD algorithms on both small and big batch sizes

Note that the quality of their results refers to the QD Score of the archive of solutions returned by an Algorithm at the end of their execution.

5.3.1.2 Arm Repertoire

In the task of Arm Repertoire, the difficulty increases compared to the Rastrigin function examined above since exploring the feature space is done by calculating the x and y positions of the arm’s end, which is a combination of all the arm’s link angles (i.e., all solution’s parameters).

Examining how the problem size affects the performance of QD and DQD Algorithms when keeping constant the batch size (here equal to 2048), from the charts of the figures 5.1 for QD Score, 5.2 for Best Fitness and 5.3 for the Coverage above on the Arm

Repertoire problem, **we can see that the QD Scores of all the QD and DQD Algorithms decreases as the problem size of the Arm Repertoire increases**.

From the figure of Best Fitness 5.2, we can see that the Best Fitness Score of all the algorithms except MAP-Elites (iso) remains close to the maximum (=100) in this problem. The Best-Fitness score for the MAP-Elites (line) though seems to be decreased more than the other QD and DQD Algorithms (except MAP-Elites-iso) as the problem size increases. The Coverage and Best-Fitness score (figure 5.3) significantly decreases for the MAP-Elites (iso) as the problem size increases (just below 8000 out of 10000 cells filled at problem size 128 becomes just over 2000 cells filled at problem size 2048 for Coverage and for the same problem sizes we have around 100 and then around 70 for Best Fitness respectively). This is the reason why its QD Score has the same decreasing trend. Despite that the increase of the problem size decreases the QD Score of each algorithm, this high degree of reduction in MAP-Elites (iso) cannot be seen in the other algorithms**. CMA-MEGA algorithms significantly underperform in all the problem sizes examined** compared to the other DQD Algorithms, the CMA-ME variations and the MAP-Elites (line).

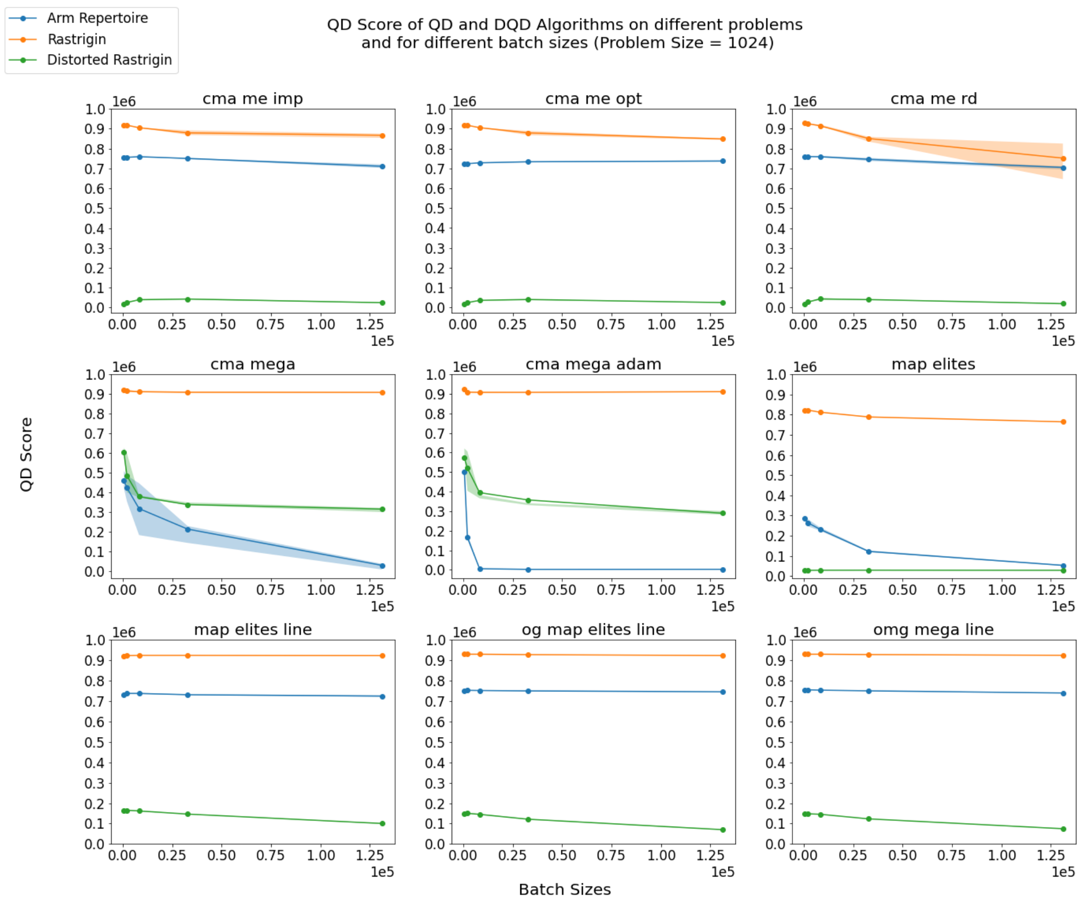


Fig. 5.4: QD Scores of QD and DQD Algorithms on different problems and for different batch sizes (Constant Problem size equal to 1024)

QD Scores of CMA-ME (imp), CMA-ME(rd), OMG-MEGA (line) and OG-MAP-Elites (line) are affected the least by the increase of the problem size on the Arm Repertoire when using batch size equal to 2048, but in greater batch sizes (e.g. 32,768) things change and only OMG-MEGA (line) and OG-MAP-Elites(line) are still stable with the increase of the problem size. This can be explained by the same trend in their Coverage (i.e., filling more cells in their archive even in bigger problem sizes). For the batch size equal to 2048, CMA-ME (rd) followed by CMA-ME (imp) were the algorithms that gave the highest QD Scores for all the problem sizes examined but as the batch size increases their QD Scores get smaller and smaller pushing them away from accomplishing the best QD Scores.

From the examination of the effect of increasing the batch size and keeping constant the problem size (=1024) for all the QD and DQD Algorithms on the Arm Repertoire (See figures 5.4 for QD Score, 5.6 for Coverage and 5.5 for Best Fitness), we observed that CMA-MEGA variations and MAP-Elites (iso) are affected the most from the batch size increase. For example, from figure 5.4 of QD Scores, CMA-MEGA (adam) on Arm Repertoire with problem size 1024 had QD Score just below 600,000 when using batch size equal to 512, whereas when using batch size equal to 32,768, its QD Score fell to below 100,000. CMA-MEGA and MAP-Elites (iso) algorithms give the worst QD Scores for all the combinations of batch sizes and problem sizes examined on Arm Repertoire.

QD Scores of CMA-ME variations also eventually decrease with the increase of the batch size, and this can be attributed to the reduction of the Coverage Score as well, which means that their exploration ability of new genotypes decreases with the increase of the batch size. More specifically, with batch sizes greater than 16384, CMA-ME(rd) and CMA-ME (imp) start to accomplish lower QD Scores than OMG-MEGA (line), OG-MAP-Elites (line) and MAP-Elites (line). For small batch sizes (i.e. 512 and 2048), CMA-ME (imp) and CMA-ME (rd) perform better in terms of QD Scores in smaller problem sizes (128, 256 and 512) and worse in bigger problem sizes (1024 and 2048), whereas in bigger batch sizes (i.e. 16384, 32768 and 131072) they perform better in bigger problem sizes (1024 and 2048) and worse in smaller ones (128, 256 and 512).

QD Scores of MAP-Elites (line), OMG-MEGA (line) and OG-MAP-Elites (line) remain approximately the same with the increase of the batch size.

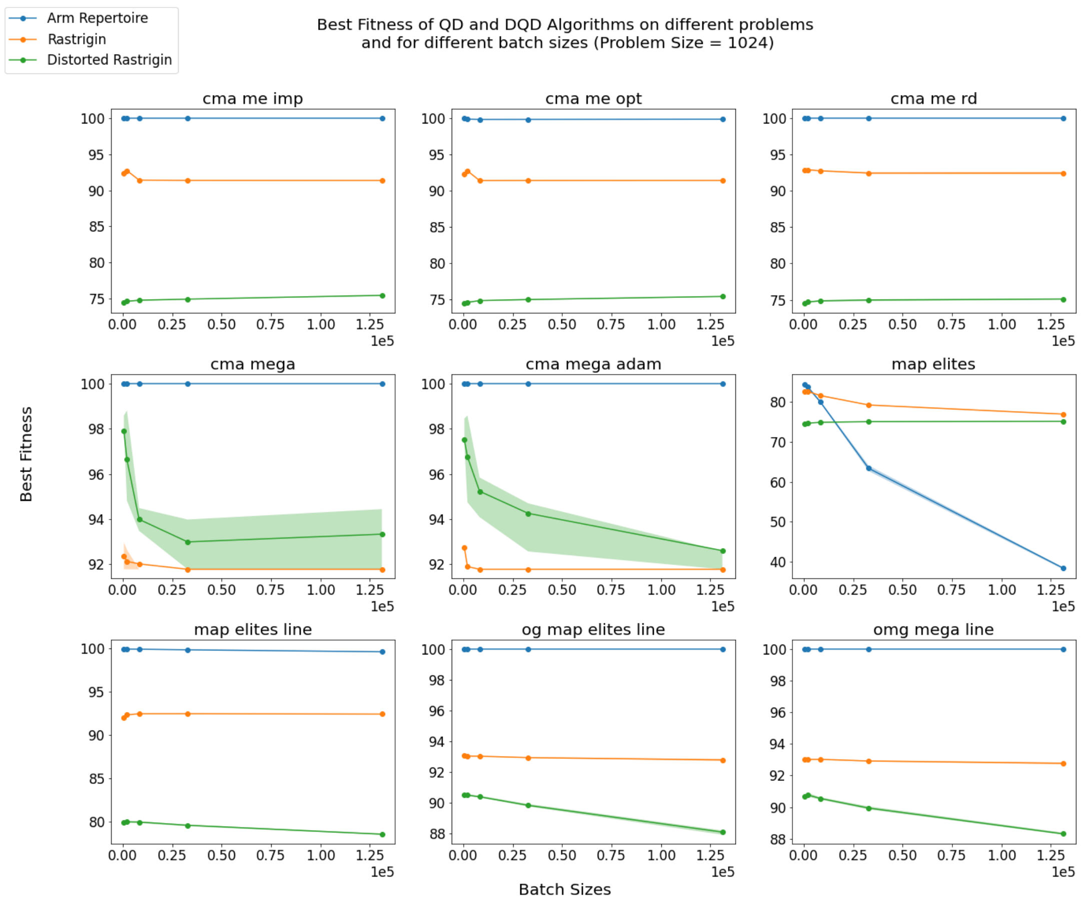


Fig. 5.5: Best Fitness of QD and DQD Algorithms on different problems and for different batch sizes (Constant Problem size equal to 1024)

In general, for the Arm Repertoire Task:

- Increasing the problem size decreases the performance of the QD and DQD Algorithms (Coverage is affected)
- Changes in problem and batch sizes affect coverage score the most (instead of the fitness of the solutions in the archive)
- The QD Scores of OMG-MEGA (line) and OG-MAP-Elites (line) are the only ones not significantly reduced with the increase of the problem size (Stable to problem size increase).
- MAP-Elites (line) is significantly affected by the increase of the problem size.
- CMA-MEGA and MAP-Elites (iso) algorithms perform really bad (low QD Scores) and their QD Scores become worse with the increase of the batch size

- MAP-Elites (line), OMG-MEGA (line) and OG-MAP-Elites (line) can be benefited by the increase of the batch size and thus accomplish better results (higher QD Scores) in bigger batch sizes.
- CMA-ME (imp) and CMA-ME (rd) algorithms are promising for small batch sizes (i.e. 512 and 2048) where they seem to accomplish the best QD Scores, but their accomplished QD Scores decrease as the batch size increases.

Note that quality of their results refers to QD Score of the archive of solutions returned by an Algorithm at the end of their execution.

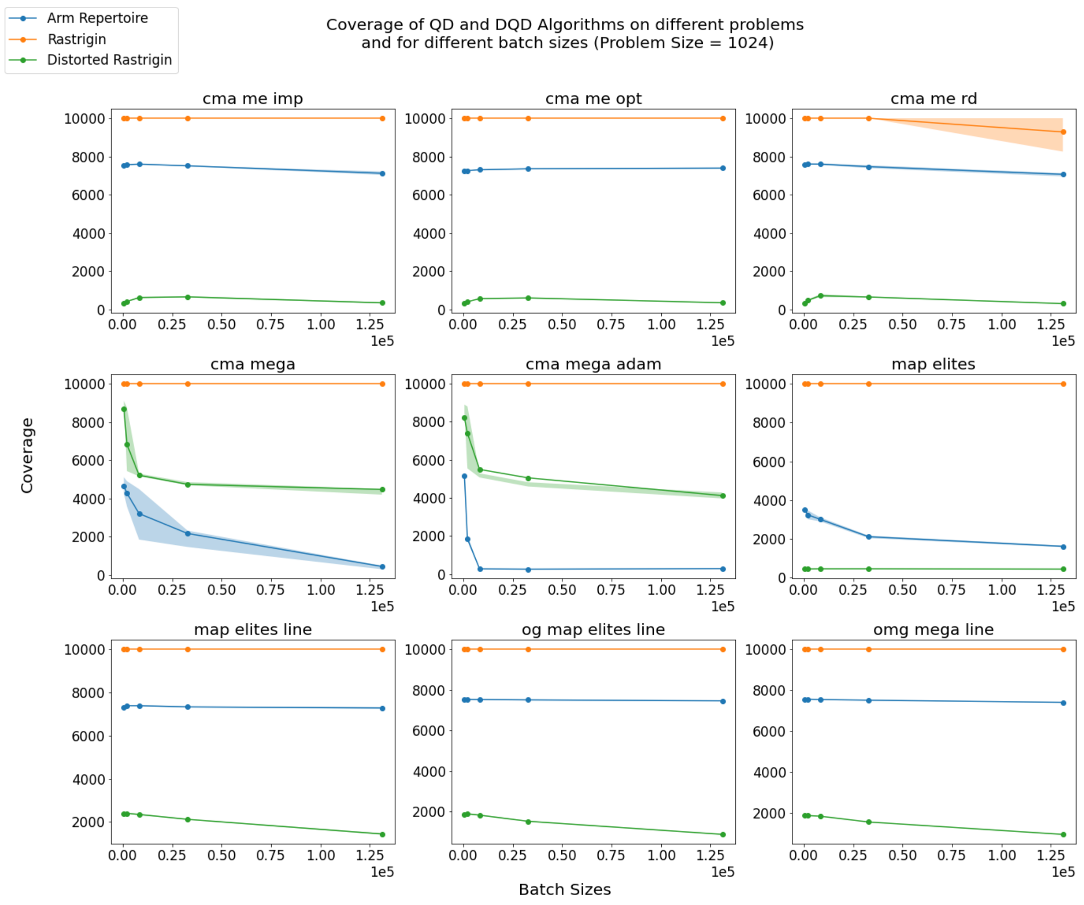


Fig. 5.6: Coverage of QD and DQD Algorithms on different problems and for different batch sizes (Constant Problem size equal to 1024)

5.3.1.3 Rastrigin Function with Distorted Behaviour Space

Moving to the Rastrigin Function with distorted Behavior Space, we can start again by examining the QD Scores of the QD and DQD Algorithms on different problem sizes and batch sizes equal to 2048 (figure 5.1 above). It is clear that all the algorithms' QD Scores significantly decrease with the increase of the problem size for the Distorted Rastrigin.

This decrease seems to depend on the decreasing trend appearing in the algorithms' Coverage scores. CMA-MEGA Algorithms significantly outperform all the other QD, and DQD Algorithms for all the problem sizes examined with batch sizes equal to 2048. CMA-MEGA and CMA-ME are affected the least by the increase of the problem size compared to the other QD and DQD Algorithms.

Examining now the effect of increasing the batch size on the QD and DQD Algorithms while we keep constant the problem size (=1024) (figure 5.4), we can see that increasing the batch size used by the CMA-ME and CMA-MEGA algorithms deteriorates their QD Scores the most. More specifically, for the CMA-ME and MAP-Elites (iso) algorithms, their performance increases until batch size 32,768 (not inclusive), from where their performance starts to decrease. For CMA-MEGA, with a problem size equal to 1024 and batch size equal to 512, CMA-MEGA variations (CMA-MEGA and CMA-MEGA-adam) accomplish around 600,000 QD Score, but with a batch size equal to 131,072 , they fall to QD Score equal to around 300,000. MAP-Elites (line), OMG-MEGA (line) and OG-MAP-Elites (line) accomplish better and better QD Scores until around batch size (2048), after which their QD Scores get smaller and smaller but in a slower trend than the other algorithms.

QD Scores of MAP-Elites(line), OMG-MEGA(line), and OG-MAP-Elites (line) are approximately the same. Despite that, OMG-MEGA(line) and OG-MAP-Elites (line) find fitter solutions than MAP-Elites(line) (can also be seen by higher Best Fitness Scores in OMG-MEGA-line and OG-MAP-Elites-line), whereas MAP-Elites(line) seems to have slightly better Coverage Scores.

Last, CMA-MEGA algorithms have the highest Best-Fitness Scores (See Figure 5.8), followed by OMG-MEGA(line) and OG-MAP-Elites(line) with slightly lower scores, which are decreased with the increase of the batch size. The Coverage Scores, of CMA-MEGA are significantly higher than of OMG-MEGA (line) and OG-MAP-Elites (line).

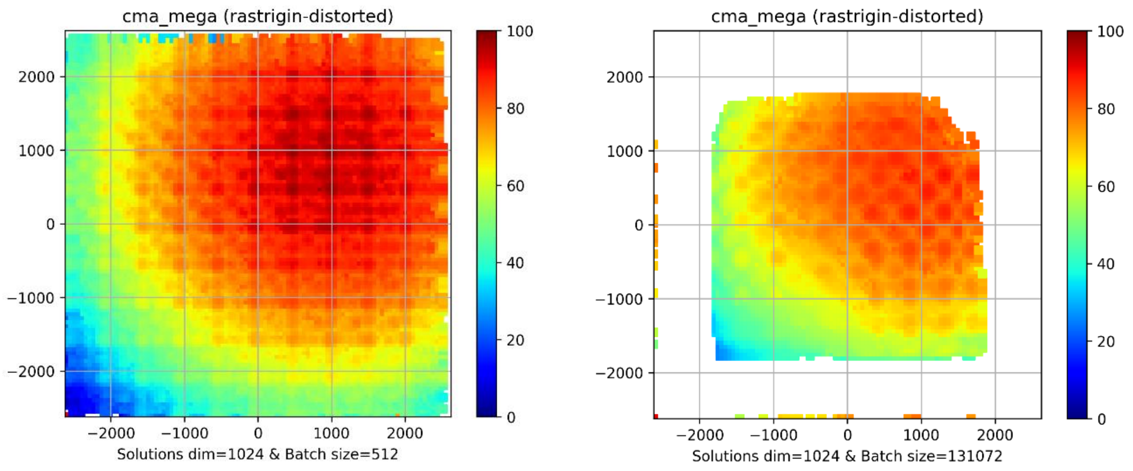


Fig. 5.8: Archives of CMA-MEGA(adam) for problem size 1024 and batch sizes 512 (left) and 131,072 (right) on the Rastrigin Function with distorted Behaviour Space

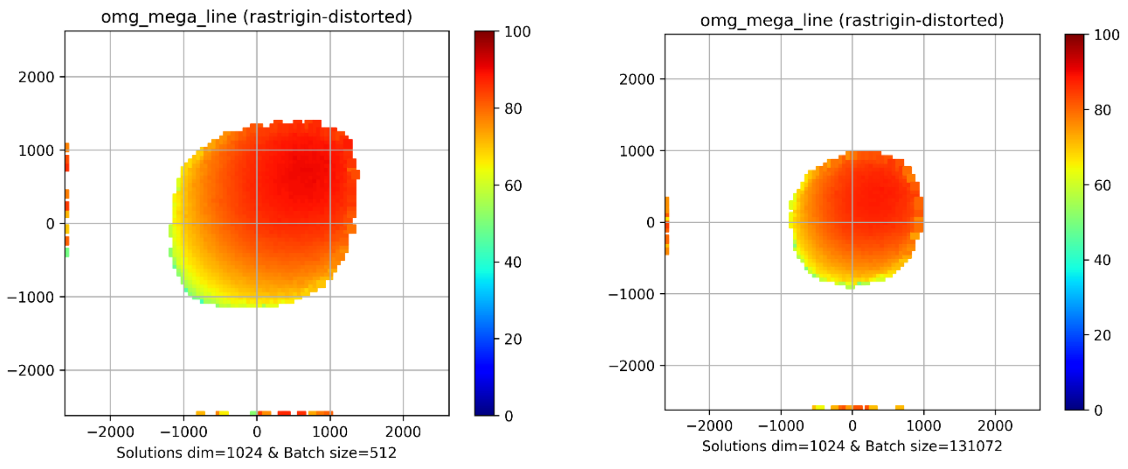


Fig. 5.9: Archives of OMG-MEGA(line) for problem size 1024 and batch sizes 512 (left) and 131,072 (right) on the Rastrigin Function with distorted Behaviour Space

From figure 5.8, we can see the archive of the CMA-MEGA (adam) for batch size 512 (left) and for batch size 131,072 (right) for the Rastrigin with Distorted Behavior Space problem with a problem size equal to 1024. Similarly, figures 5.9 and 5.10 show the same thing for the OMG-MEGA (line) and MAP-Elites (line) algorithms, respectively. From figure 5.8, we can see that CMA-MEGA fills most of its archive with solutions and also finds many local optima of the Rastrigin function with Distorted Behavior Space. The archive size, though, is decreased when moving from batch size 512 (left) to batch size 131,072 (right). The archives of OMG-MEGA (line) and MAP-Elites (line) are also negatively affected by the increase in their batch size, and OMG-MEGA (line) seems to find fitter solutions (more area with red colour) than MAP-Elites (line) (the red area in OMG-MEGA-line is orange in MAP-Elites-line), but MAP-Elites (line) seems to have slightly bigger size of an archive than OMG-MEGA (line).

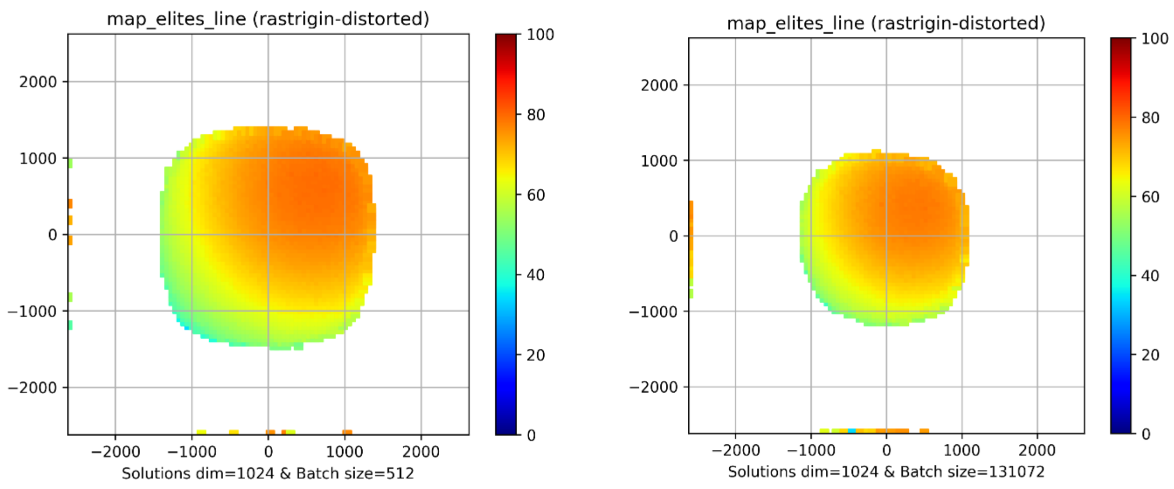


Fig. 5.10: Archives of MAP-Elites(line) for problem size 1024 and batch sizes 512 (left) and 131,072 (right) on the Rastrigin Function with distorted Behaviour Space

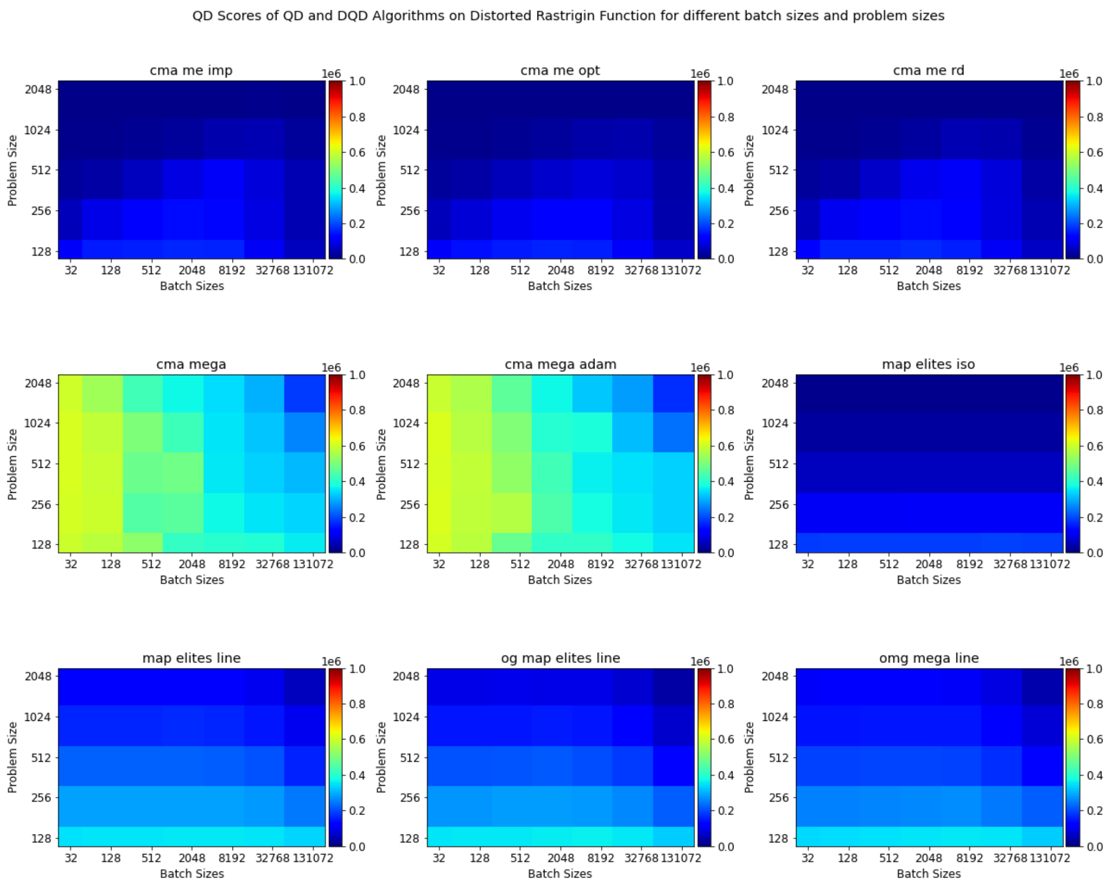


Fig. 5.11: QD Scores of QD and DQD algorithms on different problem sizes and batch sizes for the Rastrigin with Distorted Behaviour Space problem

From figure 5.11, we can see an overview of the QD Scores (represented by a color which can be interpreted to a value from the bar on the right) of all the QD and DQD Algorithms on all the combinations of batch size and problem size examined on the Rastrigin Function with Distorted Behavior Space. The interesting thing to note here is that the QD Scores of MAP-Elites (line), OG-MAP-Elites (line) and OMG-MEGA (line) (bottom three heat-maps) are significantly affected when we increase the problem size (change of the color vertically). In contrast, the batch size increase does not affect them as much as what happens with the increase of the problem size (change of the color horizontally is not as strong as how the color changes vertically). Moreover, from the same figure, CMA-MEGA variations can be seen to have significantly better QD Scores than the other QD and DQD Algorithms for small batch sizes (leftmost columns of heatmap with yellowish color). For greater batch sizes, their QD Scores decrease significantly (rightmost columns of heatmap where the color becomes bluer).

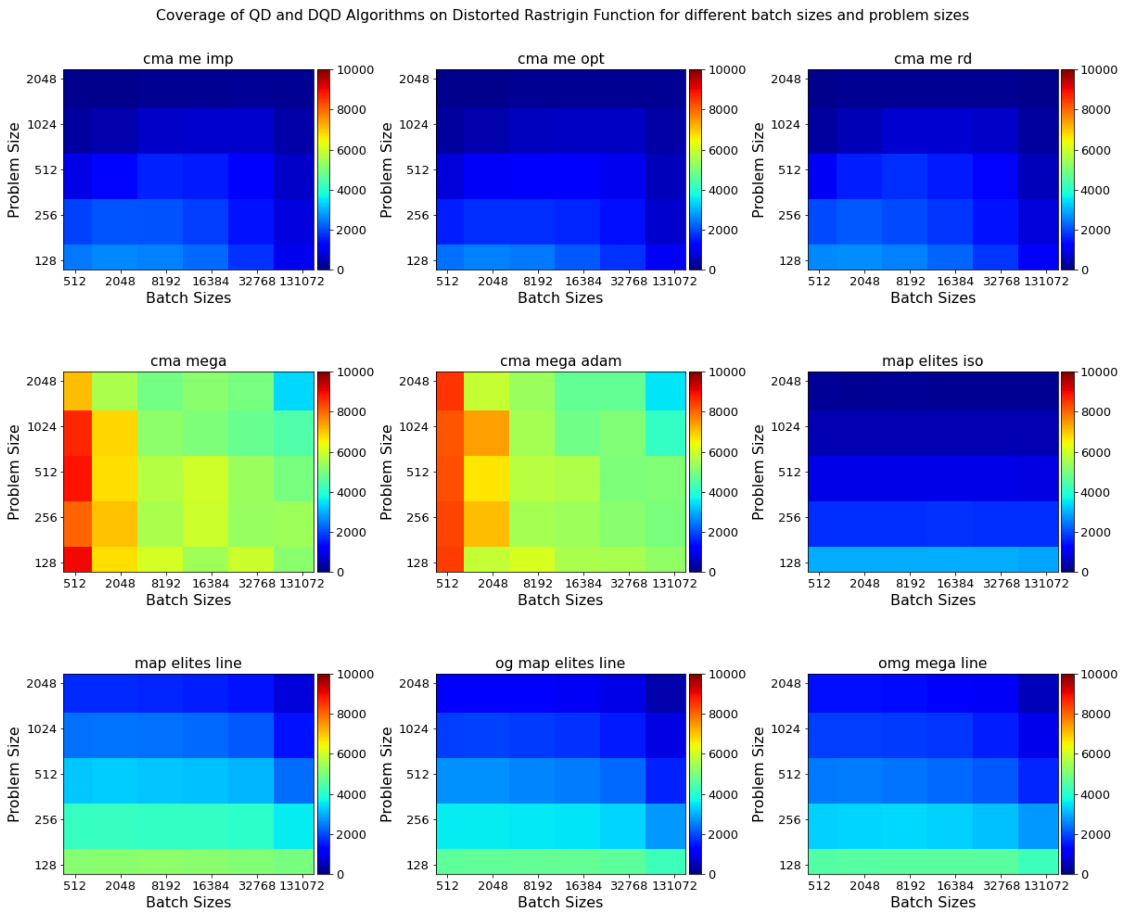


Fig. 5.12: Coverage Scores of QD and DQD algorithms on different problem sizes and batch sizes for the Rastrigin with Distorted Behaviour Space problem

From figure 5.12, we can see the Coverage scores of the of all the QD and DQD Algorithms on all the combinations of batch size and problem size examined on the Rastrigin Function with Distorted Behavior Space. We can see that the Coverage Scores of the algorithms represent their QD Scores (i.e., same patterns). Thus, the change of problem size and/or batch size affects the Coverage Score (the size of the archive) and then the QD Score.

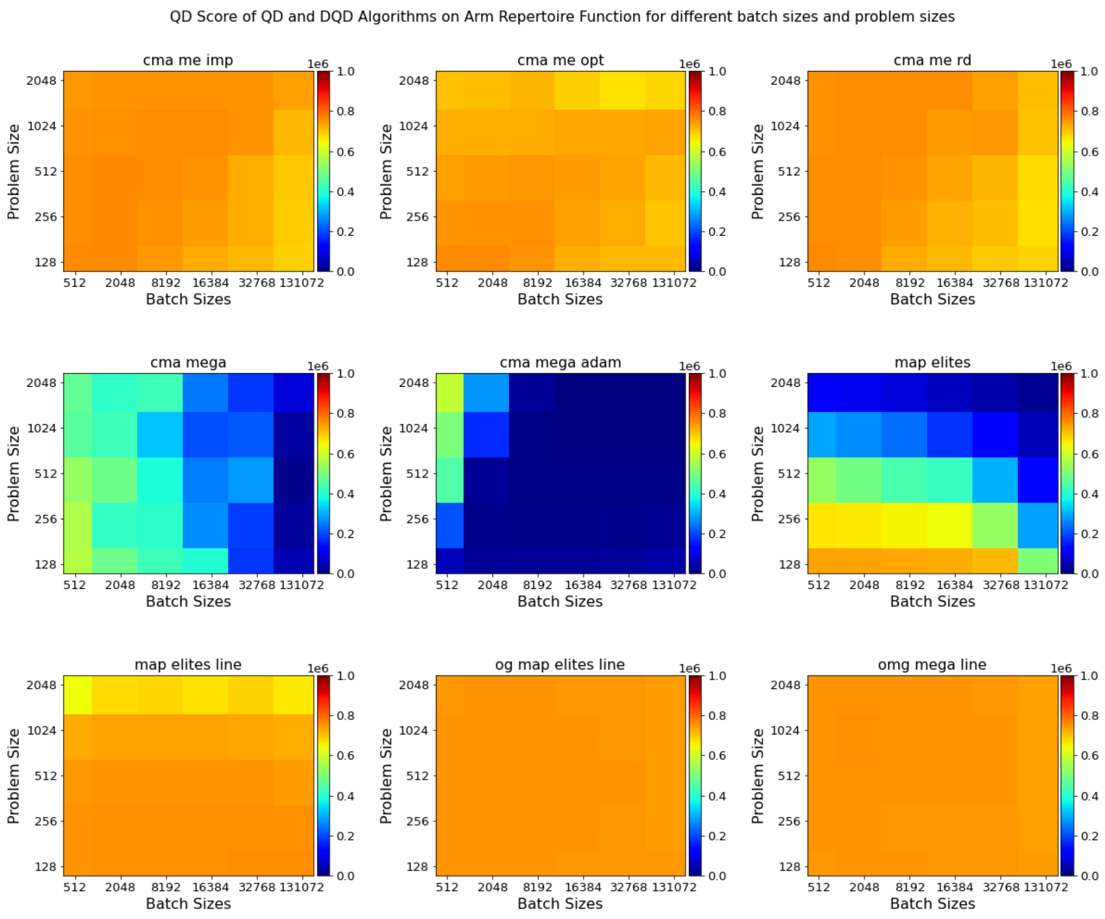


Fig. 5.13: QD Scores of QD and DQD algorithms on different problem sizes and batch sizes for the Arm Repertoire

From figure 5.13, we can see the QD scores of the of all the QD and DQD Algorithms on all the combinations of batch size and problem size examined on the Arm Repertoire. The interesting thing is that OMG-MEGA(line) and OG-MAP-Elites(line) are stable in the increase of both the batch size and problem size (retain same color), whereas the QD Scores of MAP-Elites(line) are deteriorated in bigger problem sizes.

In general, for the Rastrigin with distorted behavior space problem:

- Increasing the problem size decreases the QD Scores (QD Scores) of the QD and DQD Algorithms in general (Coverage seems to be affected)
- Increasing the batch size decreases the QD Scores of the QD Algorithms, especially when working on bigger problem sizes.
- Changes in problem and batch sizes affect coverage score the most (instead of the fitness of the solutions in the archive)
- CMA-ME algorithms and MAP-Elites (iso) accomplish the lowest QD Scores compared to the other QD and DQD Algorithms for all the combinations of problem size and batch size examined
- CMA-MEGA Algorithms outperform all the other QD and DQD Algorithms. They perform that well primarily because they fill more container cells (greater archive size) than the other QD and DQD Algorithms.
- MAP-Elites(line), OMG-MEGA (line) and OG-MAP-Elites (line) accomplish approximately the same QD Scores, but all three of them can be benefited from big batch sizes more than the other QD and DQD Algorithms since the increase of the batch size decreases their QD Scores less than what happens with the other Algorithms (128 and 256).

In conclusion we can identify a few general conclusions:

- MAP-Elites(line), OMG-MEGA (line) and OG-MAP-Elites (line) are benefited from batch size increase (until 131,072 which was tested)
- CMA-ME and CMA-MEGA algorithms cannot be used with big batch sizes, and they can perform better than all the other QD and DQD algorithms when used with small problem size in problems with difficult mapping of genotypes to the feature space
  → Should only be used with small batch sizes (<512)
- OMG-MEGA (line) and OG-MAP-Elites (line) accomplish higher QD Scores than MAP-Elites(line)
  → Use OMG-MEGA (line) and OG-MAP-Elites (line) instead of MAP-Elites(line) when the runtime of OMG-MEGA (line) and OG-MAP-Elites (line)

is not prohibitively higher (e.g., when calculating derivatives leads to much higher runtimes) than of MAP-Elites (line)

5.3.2 The runtime of QD + DQD Algorithms on GPU compared to CPU

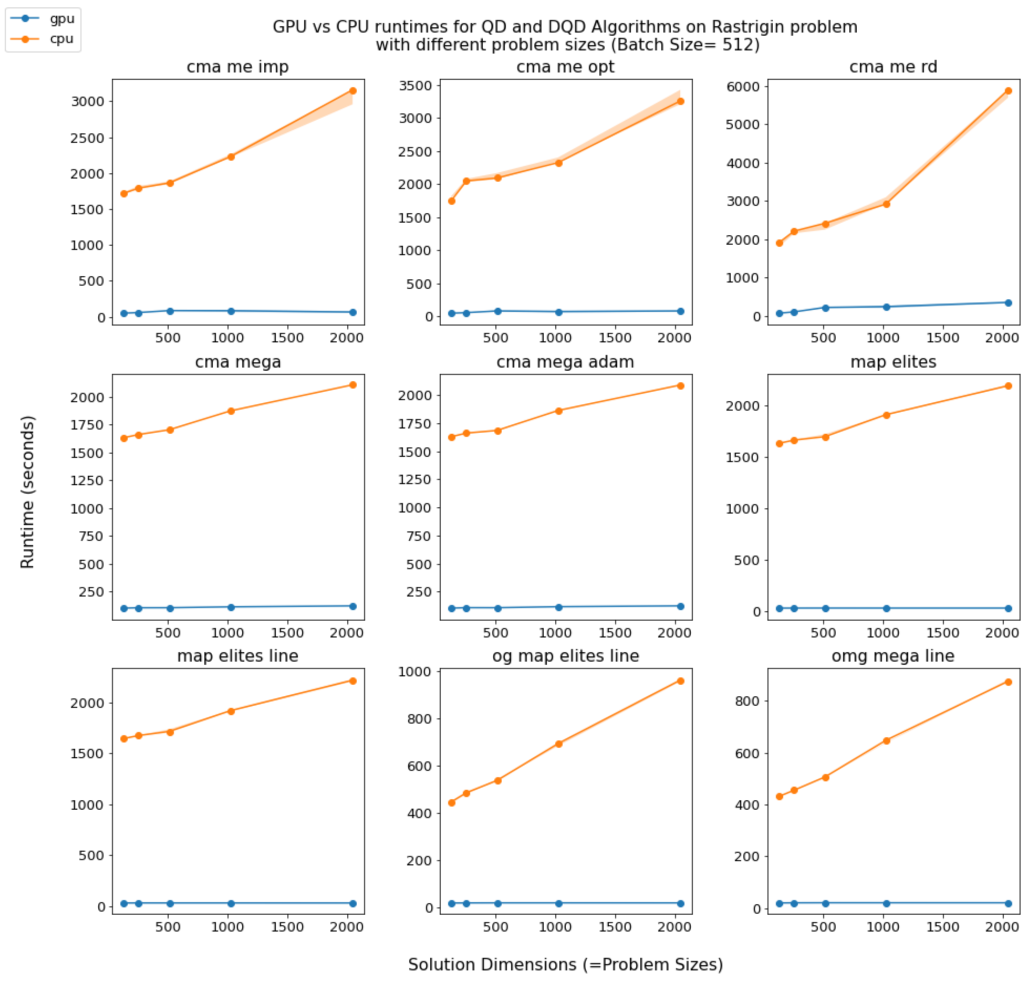


Fig 5.14: Runtime of QD Emitters on Rastrigin Function with simple encoding for batch size equal to 512 and increasing problem sizes

In the previous section, we have investigated the performance (in terms of the metrics QD Scores, Best Fitness and Archive Size/Coverage) of QD and DQD Algorithms for bigger batch sizes and bigger problem sizes.

We saw that the batch size can negatively affect the performance of QD and DQD Algorithms in some tasks/problems but at the same time it can have no significant negative effect (e.g. OMG-MEGA(line) and OG-MAP-Elites (line) performing well in Arm Repertoire and Rastrigin with simple encoding but not well in Rastrigin with distorted Behavior Space). Thus, there is potential for some QD and DQD Algorithms to exploit greater sizes of batch sizes and improve their runtime performance when executed on GPUs. Consequently, our second experimental goal is to investigate whether there is any significant speedup in runtime when using GPUs instead of CPUs.

Before we investigate the potential of speeding up the algorithms while we increase the batch size, let us investigate what happens to the runtime of QD and DQD algorithms on CPU and GPUs, when we increase the problem size that they use. Below, at the chart 5.15 we can see the runtimes of QD and DQD Algorithms with batch size 512 on CPU and GPU, with different problem sizes on the Rastrigin Function with simple encoding.

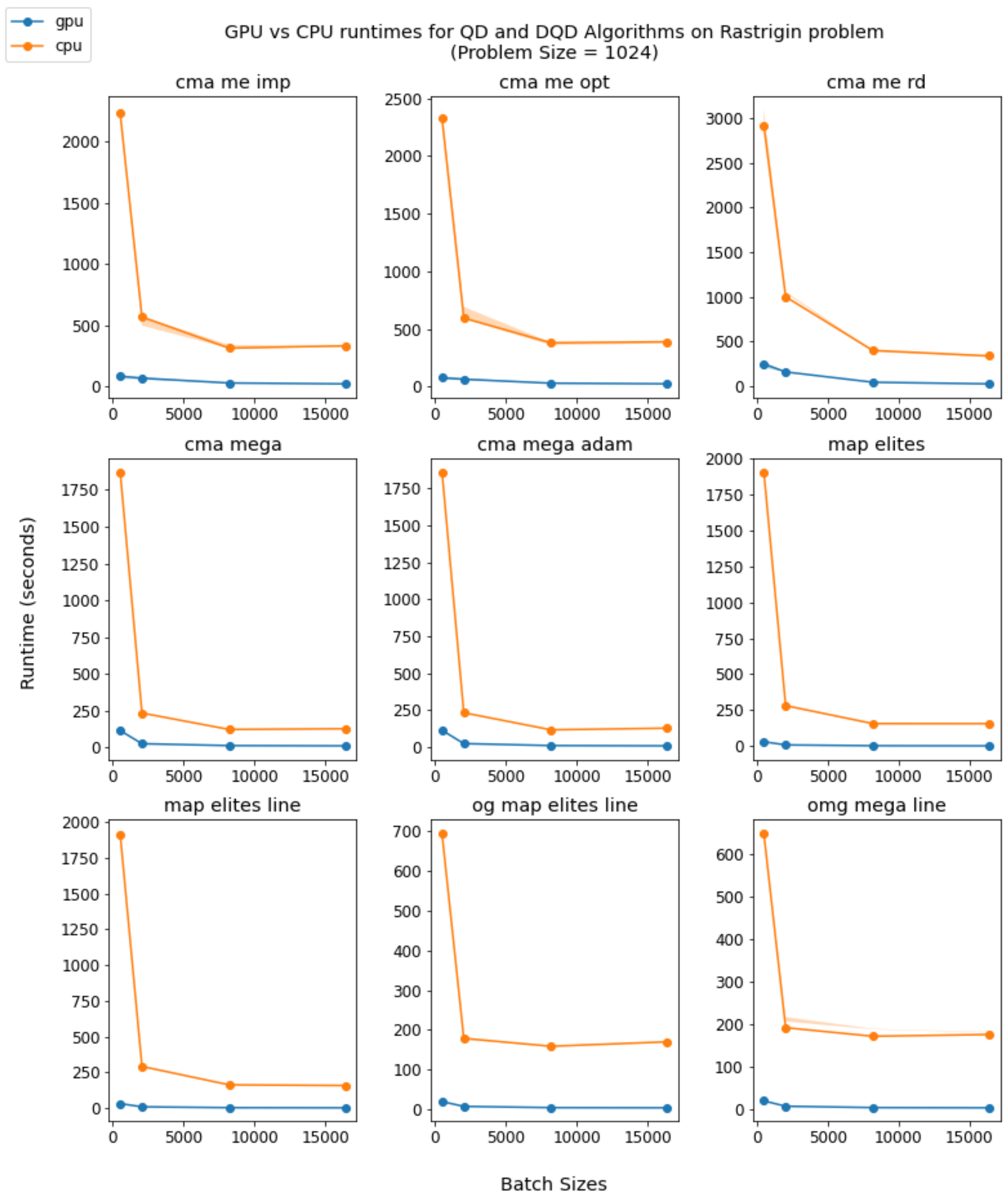


Fig 5.15: Runtime of QD Emitters on Rastrigin Function with simple encoding for problem size equal to 512 and increasing batch size

From this set of charts, it is obvious that increasing the problem size increases the runtime of all the QD and DQD algorithms on both the GPUs and CPUs. Despite that, the runtimes on GPUs increase much less with the increase of the problem size than the runtimes of CPUs. The difference in runtimes between CPUs and GPUs is significant. For example, MAP-Elites (line) with 512 batch size and 2048 problem size takes 20 seconds on GPUs whereas it takes 2106 (≈36 minutes) on CPUs.

Now, moving to the examination of the effect on the runtimes of the QD and DQD algorithms when increasing the batch size, we can clearly see that there is a significant acceleration in the runtimes of QD and DQD Algorithms when being executed on GPUs, instead of CPUs. This applies in all the three Domains (Rastrigin, Arm Repertoire and Rastrigin with Distorted Behavior Space) examined, but we will show the results for the Rastrigin function with simple encoding.

Above, in figure 5.15, you can see the Runtimes of all the QD and DQD Algorithms for increasing batch sizes and for a constant problem size (dimensions of a solution) equal to 1024. The runtimes of the QD + DQD algorithms on CPU and GPU significantly decrease with the increase of the batch size. The runtimes from the GPUs are considerably smaller than the runtimes from CPUs for all the combinations of batch sizes and problem sizes examined. More specifically, for batch size equal to 512 and problem size equal to 1024, all the QD Algorithms (excluding the DQD) executed on CPUs needed at least around 30 minutes (>≈1800 seconds). On the other hand, QD Algorithms executed on GPUs needed around a minute for problem size equal to 1024 and batch size equal to 512, except the CMA-ME (rd), which needed more time (248 seconds). For the DQD algorithms, we can see something similar for batch size 512 and problem size 1024 but with shorter runtimes. Concisely, all the DQD algorithms executed on GPU took less than a minute to complete for problem size equal to 1024 and batch size 512, whereas the runtime of the same algorithms on CPUs for the same problem size and batch size needed not less than 10 minutes.

Despite that increasing the batch size decreases the runtime of both QD and DQD algorithms, this reduction in runtime depends on the number of evaluations we want to perform since, from the figure 5.15, we can see that the improvement in runtime becomes smaller and smaller with the increase of the batch size. Reducing the number of epochs by increasing the batch size improves the runtime, but there is for sure a natural limitation on how big the batch size can be.

The CMA-ME algorithms take more time to execute than the MAP-Elites iso and line algorithms. This is true for both CPU and GPU executions (Table 1 and Table 2 below). The same can be observed for the DQD Algorithms between the CMA-MEGA Algorithms (gradient variant of CMA-ME improvement) and the OMG-MEGA or OG-MAP-Elites (gradient variants of MAP-Elites).

Runtime (Seconds)	*MAP-Elites (iso)*	*MAP-Elites (line)*	*CMA-ME (imp)*	*CMA-ME (opt)*	*CMA-ME (rd)*	*OMG-MEGA (line)*	*OG-MAP-Elites (line)*	*CMA-MEGA*	*CMA-MEGA (ADAM)*
CPU	156	160	332	380	332	182	169	126	129
GPU	2	3	22	23	25	5	5	11	11

Fig 5.1: Runtime of QD Emitters on Rastrigin Function with simple encoding for problem size equal to 1024 and batch size 2048

Runtime (Seconds)	*MAP-Elites (iso)*	*MAP-Elites (line)*	*CMA-ME (imp)*	*CMA-ME (opt)*	*CMA-ME (rd)*	*OMG-MEGA (line)*	*OG-MAP-Elites (line)*	*CMA-MEGA*	*CMA-MEGA (ADAM)*
CPU	283	291	525	620	1048	215	177	233	234
GPU	9	9	69	62	159	8	8	25	26

Table 5.2: Runtime of QD Emitters on Rastrigin Function with simple encoding for problem size equal to 1024 and batch size 16,384

OMG-MEGA (line) and OG-MAP-Elites (line) take approximately the same runtime to finish with MAP-Elites (line).

Last, the CMA-ME algorithms are the slowest algorithms from all the other QD and DQD algorithms (Table 1 and Table 2 above). CMA-MEGA though are faster than CMA-ME algorithms, but slower than all the rest QD and DQD algorithms.

All of the observations mentioned for the Runtimes of the algorithms on the Rastrigin Function with simple encoding problem can be generalized since the same things appear for the other tasks as well. That is, all these observations apply for the Rastrigin Function with Distorted Behavior Space and the Arm Repertoire problems as well.

To see the runtimes of all the QD and DQD Algorithms for 5 million evaluations on the Rastrigin Function with simple encoding, Rastrigin Function with Distorted Behavior Space, and the Arm Repertoire problems, please refer to the Appendices.

In general:

- CMA-ME algorithms are the slowest algorithms from all the other QD and DQD algorithms examined.
- Increasing the problem size increases the runtime of the QD and DQD Algorithms.
- Increasing the batch size of each algorithm reduces their runtime, but the improvement becomes less and less with each batch size increase.
- The use of GPUs is beneficial compared to using CPUs since the runtime on CPUs is significantly greater than on GPUs

5.3.3 Performance of DQD Algorithms vs QD Algorithms on GPUs

Now that we have seen the performances of all the Algorithms on two different dimensions (QD Scores, Coverage, Best Fitness vs Runtime) separately, it is important to combine the results we have identified and try to understand which algorithms can provide sufficiently good results fast and also whether DQD Algorithms can be beneficial for accomplishing that (i.e. good results in a short amount of time). That is, which algorithms are able to exploit parallelization in order to be executed fast and at the same time give good results comparable to not using parallelization (not significantly degraded performance) and also whether DQD can be beneficial in accomplishing that.

First of all, the runtimes of CMA-ME algorithms compared to the other QD Algorithms, and the runtimes of CMA-MEGA algorithms compared to the other DQD Algorithms is significantly higher. This could be a problem in the case of using the algorithms with more complex tasks (e.g. learning robot tasks via simulations). But at the end of the day, the final outcome of the algorithms might matter more (i.e. getting many high-performing solutions that vary sufficiently across different features). More specifically, we have seen that the CMA-ME Algorithms take more time to be executed than the other QD algorithms (they require, at minimum, twice the amount of time that the other QD Algorithms require, and the runtime difference between the other QD and CMA-ME algorithms becomes bigger as the problem size increases) and do not do well in both of the Rastrigin problems examined. Only for the task of Arm-Repertoire, their QD Scores were, in general, better than the other Algorithms' QD Scores, but that was true for small batch sizes. Thus, the current implementation of CMA-ME is not beneficial with big batch sizes (>=512). The potential speedup through when increasing the batch size is clearly big; the only problem is that the current implementation of CMA-ME gives worse results.

CMA-MEGA is faster than its CMA-ME variants but slightly slower than all the other QD and DQD Algorithms. It performs really bad in terms of QD Scores in Arm-Repertoire, and generally, it performs worse than the CMA-ME variants in terms of QD Scores in Arm-Repertoire and Rastrigin with simple encoding. The thing that makes it really promising though is its performance on the task of Rastrigin with Distorted Behavior Space, where it significantly outperformed all the other QD and DQD algorithms for all the problem sizes and batch sizes examined. But the bigger the batch size, the lower its accomplished QD Score. Thus, for the CMA-MEGA algorithms using big batch sizes does not help the quality and size of the collection of solutions they return.

OMG-MEGA (line) and OG-MAP-Elites (line) showed really good QD Scores in the Rastrigin with simple encoding and Arm-Repertoire problems. More specifically, in the Rastrigin with simple encoding, they had the best QD scores for all the combinations of problem sizes and batch sizes examined, whereas, in the Arm-Repertoire, their QD scores were close to the best ones. They are generally not affected by the increase of the batch size and problem size (Stable). But in some cases, like in Rastrigin with distorted behavior space, the increase of the batch size and problem size made clearer their decrease in their

QD Scores. These algorithms performed better than their QD original MAP-Elites (line). In terms of Runtime, OMG-MEGA (line) and OG-MAP-Elites (line) seem to be as fast as MAP-Elites (line). More specifically, OMG-MEGA (line) and OG-MAP-Elites (line) are faster with smaller batch sizes (e.g., 512 MAP-Elites-line takes around 30 seconds, whereas the OMG-MEGA-line and OG-MAP-Elites-line take around 20 seconds for all the different problem sizes examined), but slightly slower with bigger batch sizes (e.g., 16384 MAP-Elites-line takes around 3 seconds whereas the OMG-MEGA-line and OG-MAP-Elites-line take between 5-7 seconds for all the different problem sizes examined).

MAP-Elites (line) performed really well in Rastrigin with simple encoding and Arm Repertoire. Still, in those two problems we could see that it was affected significantly by the problem size (a bigger problem size led to lower QD Scores). In contrast, its DQD variants OMG-MEGA (line) and OG-MAP-Elites (line), weren't significantly affected by the problem size. In terms of their runtime, they are faster than all of the CMA-ME and CMA-MEGA algorithms and are as fast as the OMG-MEGA and OG-MAP-Elites Algorithms.

MAP-Elites (iso) are the fastest but the worst in terms of quality of results (i.e., lowest QD Scores) in all the problems examined.

Thus, in conclusion, we can see that the performance of each algorithm depends on the task examined. Experiments on different tasks, can show different Algorithms performing better. Despite the fact that there does not seem to be a single global ideal algorithm, we can note a few general observations:

- MAP-Elites (line), OMG-MEGA (line) and OG-MAP-Elites (line) are generally faster than the CMA-ME and CMA-MEGA Algorithms, and in general, the batch size can help them speed up their execution without losing a significant quality of their results. Depending on the task and its difficulty exploring the feature space, they are capable of performing really good and even better than the CMA-ME and CMA-MEGA algorithms.
- Depending on the task and its difficulty exploring the feature space OMG-MEGA (line) and OG-MAP-Elites (line) can perform better than MAP-Elites (line) (e.g. Rastrigin with simple encoding) because they can usually find fitter solutions, and

they seem to be slightly more stable than MAP-Elites (line) when increasing the problem size (e.g. Arm Repertoire).

- CMA-MEGA algorithms do not do well with the increase of the batch size they use (i.e. the size and the quality of the collection of solutions they return are degraded), and they seem to be very promising when they use small batch size and when they are used in particular types of problems, that introduce a significant difficulty in exploring the feature space from solutions generated. In those problems, they seem to explore better the feature space and, thus, fill more empty cells in their archive.
- CMA-ME algorithms seem to require the most time to be executed, and in general, they perform better in smaller batch sizes. Despite that, they also seem promising for particular types of problems when they use small batch sizes.

These lead us to the following general conclusions:

- OMG-MEGA (line) and OG-MAP-Elites (line)
    - **can provide a speedup when used with GPUs and with bigger batch sizes** (just like MAP-Elites-line but requiring around the same runtime; thus can replace MAP-Elites-line)
    - **can provide better results than all the other QD and DQD algorithms in Simple Problems (i.e., Rastrigin with simple encoding)**
- CMA-ME and CMA-MEGA
    - can be benefited from optimized code on GPUs but not from the increase of their batch size (**No possible further speedup**)
    - can get better results than all the other QD and DQD algorithms when using small batch size and especially in difficult problems (Arm Repertoire and Rastrigin with distorted Behavioral Space) where the mapping of solutions to their feature space is not simple like in Rastrigin with simple encoding

Concisely, OMG-MEGA(line) and OG-MAP-Elites (line) **can be sped up without losing the quality of their results,** whereas the **CMA-ME and CMA-MEGA cannot be sped up**. Despite that CMA-ME and CMA-MEGA cannot be sped up, they continue to outperform the other QD and DQD algorithms in terms of the quality of their solutions in the archive and the size of their archive, but they require more runtime because they accomplish that with smaller batch sizes.

Chapter 6

Conclusion & Future Work

6.1 Lessons Learnt

There are different lessons that we learned when working with JAX, which we include in this section.

1. Avoid the use of loops and conditions: We have seen that in JAX, loops and conditions, like if-statements, are some of the things that cannot be used as they are used in conventional python. These limitations are introduced to make tracing your code easier but, at the same time, can improve the performance of your code. For example, a for-loop that can be statically defined (the number of iterations is known before the code with the for-loop is compiled) by the JAX-JIT compiler is unrolled in the final optimized code. The unrolled loop creates more code for compilation and execution, leading to greater compilation and potentially execution times. Thus, avoiding loops when this can be done is a good step to make your code JAX-Compatible and also make it more efficient.
2. Think Vectorizing your code: A better alternative to using loops in implementations of ML, DL and RL algorithms is the use of vectorization. That is, organize your data that you need to perform operations on vectors and matrices and the operations on those data as vector and matrix operations. The benefit of vectorization is that many libraries execute these operations efficiently, and many times they automatically take advantage of available hardware that can improve the speed of the operations.
3. Optimizing your code for efficiency on specific hardware pays off: There are many libraries (e.g. Numba and JAX) that can optimize your code for a

specific device you want to run it for in order to speed up different operations (e.g. linear algebra operations). We used JAX, and via the JAX-JIT compiler, we were able to compile the same code for either CPUs, CUDA GPUs or TPUs. Compiled code for CPUs was much faster than conventional Python code. For example, the CMA-ME-imp algorithm we implemented needed more than twice as much time when being run without being optimized than when being optimized with JAX-JIT.

4. Implementing algorithms in JAX does not include replacing NumPy with JAX-NumPy: One of the misinterpretations of JAX that we have seen people have is that you can take an algorithm that uses the NumPy library and change the references to NumPy to be JAX-NumPy. Despite that this can work, in the general case, it is not that simple and not all the NumPy references are always needed to be replaced with JAX-NumPy. As we mentioned again, writing code in JAX includes replacing loops with method-like structures, removing if-statement, eliminating side-effects in methods etc.
5. Avoid parallelizing work on CPU's cores manually using JAX's *pmap*: The command pmap defined by the JAX framework is used to map a function on data on multiple devices in parallel. It is primarily used for different devices (e.g. different GPU devices), but it can also be used with different CPUs. The different CPUs can be different cores on the same or different chips. When using pmap, the data are copied to the other device so that the other device can work with them. Thus, using the pmap for cores of the same CPU that use the same memory will cause the data to be copied as many times as it is your degree of parallelization with pmap. Thus, using pmap with cores of the same chip is unnecessary since JAX can manage them automatically and also, the use of pmap on the cores of the same chip can create performance degradation if not careful.

6.2 Conclusions

The exploration of the question of whether GPUs can help speed up QD and DQD Algorithms led us to the exploration of the technologies and frameworks available to support executing Machine, Deep and Reinforcement Learning algorithms on GPUs. In reality, we have seen that there are many of them, like JAX and PyTorch, that make the development of algorithms on specialized hardware like GPUs and TPUs easier.

Choosing the JAX framework allowed us to see the ease that a framework like JAX offers in developing algorithms for GPUs and TPUs, but we have also seen the constraints that are being introduced with them. These constraints are enforced for optimizing code to be executed on specialized hardware, including but not limited to not using conventional Python Objects and not using Python if-statements in the code. To address that, we created a framework that makes the development of QD Algorithms optimized on JAX for specialized hardware easier and more scalar. Based on that framework, we have implemented all the state-of-the-art QD and DQD Algorithms and used them to help us answer our question on whether GPUs can help speed up QD and DQD Algorithms.

Then we tested those algorithms on a set of different tasks, different batch sizes and for different problem sizes (Rastrigin with distorted Behavior Space, Rastrigin with simple encoding and Arm Repertoire) on both GPU and CPU. We investigated three things: (1) how these algorithms are affected by the increase of the batch size, (2) whether running the algorithms on GPUs can help speed up the algorithms, and (3) whether it is beneficial to use both GPUs and Differentiability in the QD Algorithms (i.e. use DQD algorithms on GPUs) to get better results and faster.

Based on our experiments we made on MAP-Elites(line), OMG-MEGA (line) and OG-MAP-Elites (line) we have seen that they (1) are **benefited from batch size increase** (until 131,072 which was tested), (2) are **faster than the family of CMA-MEGA and CMA-ME** algorithms, (3) perform **as good or even better than CMA-MEGA and CMA-ME in simple problems** where the mapping of genotypes to the feature space is simple (e.g., Rastrigin with simple encoding), (4) that OMG-MEGA (line) and OG-MAP-Elites (line) perform better than MAP-Elites(line) and are as fast as MAP-Elites(line). These led us to the conclusion that OMG-MEGA (line) and OG-MAP-Elites (line) in general are stable in batch size and problem size changes and thus **can provide a beneficial speedup when used with GPUs and with bigger batch sizes replacing the MAP-Elites (line) algorithm.**

On the other hand, from our work with CMA-ME and CMA-MEGA algorithms we have seen that they (1) cannot be used with big batch sizes, (2) they usually perform better than all the other QD and DQD algorithms when used with small batch size (i.e. 32 batch size in Fontaine & Nikolaidis [4] and in our experiments with batch size 512 CMA-ME were the best in Arm Repertoire and CMA-MEGA were the best in Rastrigin with Distorted

Behavior Space), (3) their performance (QD Scores) power is clear in problems where the mapping of solutions to their feature space is not simple like in Rastrigin with simple encoding but more difficult (e.g., CMA-MEGA in Rastrigin function and CMA-ME in Arm Repertoire), (4) that the CMA-ME are the slowest algorithms, (5) CMA-MEGA are faster than CMA-ME. Thus, we concluded that CMA-ME and CMA-MEGA still outperform the other QD and DQD algorithms in terms of archive size and quality of solutions in archive in small batch sizes, especially in more difficult problems where the mapping of solutions to the feature space is not simple(e.g. Rastrigin with Distorted Behavioral Space and Arm Repertoire). But, despite that they can be benefited from optimized code on GPUs, the increase of the batch size significantly decreases their performance, which means that there is no beneficial further speedup for them requiring them to be executed more time to get better results than OMG-MEGA (line) and OG-MAP-Elites (line).

Overall, with this final section we conclude that we found a significant speedup for all the QD and DQD algorithms when being executed on GPUs compared to CPUs, but only MAP-Elites(line), OMG-MEGA (line) and OG-MAP-Elites (line) are accomplishing beneficial further speedup with the batch size increase (>512). In this opportunity for speedup, OMG-MEGA (line) and OG-MAP-Elites (line) are proven more attractive to be used in any kind of problem than MAP-Elites (line) since they require approximately the same time as MAP-Elites (line) and they accomplish most of the time better QD Scores. CMA-MEGA and CMA-ME algorithms continue to outperform in terms of QD Scores the other QD and DQD algorithms in problems where the mapping of solutions to their feature space is not simple, but their current implementations do not benefit them for a speed up by increasing their batch size (i.e., they work well with small batch sizes e.g. 32 as tested by Fontaine and Nikolaidis [4]).

Through this work, we hope to see the community start using our ideas, code and even implementations to accelerate QD and DQD algorithms on specialized hardware like GPUs. We also hope to encourage the community to work on the limitations and issues of the QD and DQD algorithms that arise when trying to exploit parallelisation (i.e. using greater sizes of batch sizes), and we hope to see new algorithmic ideas that could address those limitations and leverage the massive parallelism offered by specialized hardware to improve performance of QD and DQD algorithms.

6.3 Future Work

Our work focused on examining the performance of QD and DQD Algorithms on Toy Domains (Types of QD Problems) on GPUs and CPUs. But our final goal is to do the same examination on more complex Domains, including learning robotic skills (e.g., allowing a hexapod to learn to move in each direction of the space) via simulations (i.e. BRAX Simulator). Thus, in the future, the performance of all the QD and DQD Algorithms on Complex Domains, on GPUs and CPUs should be examined to investigate the three questions that we investigated. That is, (1) What happens to the performance of the algorithms when the batch size and the problem size change, (2) What is the speedup acquired by executing QD and DQD algorithms on those more complex environments on GPUs compared to CPUs and (3) "Can DQD be proven to help get high-performing solutions quickly on GPU for those more complex Domains?". There are more things to be considered in more complex QD Domains. For example, in more complex environments, the size of the solutions can be significantly bigger than in Toy Problems since a solution there can be more complex things like the parameters of a Neural Network. In addition to that, DQD Algorithms depend on calculating the derivatives of the solutions with respect to the objective and behavioural functions' outputs. In more complex Domains, the derivatives are not usually computed analytically, just in our case, where hardcoded equations were calculating the derivatives, but auto-differentiation is usually used. The auto-differentiation calculates the derivatives of complex functions by repeatedly applying the chain rule. This thing costs more in terms of computational time, and it would probably change the runtimes of the DQD Algorithms in those complex QD Domains.

In addition to that, more investigation should be done on why the QD and DQD algorithms are affected by the big batch sizes and whether using multiple individual Emitters (instances of QD and DQD Algorithms) with smaller batch sizes can be more beneficial than using a single Emitter with huge batch size. This investigation should also include more information about the limits of the performance of QD and DQD algorithms using big batch sizes (e.g., how big batch sizes can MAP-Elites(line) be used before its performance starts to decrease?).

References

[1] Mouret, J.-B., & Clune, J. (2015). Illuminating search spaces by mapping elites. arXiv [cs.AI].

[2] Chatzilygeroudis, K., Cully, A., Vassiliades, V., & Mouret, J.-B. (2020). Quality-Diversity Optimization: a novel branch of stochastic optimization. arXiv [cs.NE].

[3] Cully, A., Clune, J., Tarapore, D., & Mouret, J.-B. (2015). Robots that can adapt like animals. Nature, 521(7553), 503–507. doi:10.1038/nature14422

[4] Fontaine, M. C., & Nikolaidis, S. (2021). Differentiable Quality Diversity. arXiv [cs.AI].

[5] Lehman, J., & Stanley, K. O. (2011). Evolving a Diversity of Virtual Creatures through Novelty Search and Local Competition. Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation, 211–218.

[6] Fontaine, M. C., Togelius, J., Nikolaidis, S., & Hoover, A. K. (2019). Covariance Matrix Adaptation for the Rapid Illumination of Behavior Space. CoRR, abs/1912.02400.

[7] Vassiliades, V., Chatzilygeroudis, K., & Mouret, J.-B. (2018). Using Centroidal Voronoi Tessellations to Scale Up the Multidimensional Archive of Phenotypic Elites Algorithm. IEEE Transactions on Evolutionary Computation, 22(4), 623–630.

[8] Grillotti, L., & Cully, A. (2021). Unsupervised Behaviour Discovery with Quality-Diversity Optimisation. CoRR, abs/2106.05648.

[9] Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. In: Z. Ghahramani and M. Welling and C. Cortes and N. Lawrence and K.Q. Weinberger (Eds.), Advances in Neural Information Processing systems 27(NIPS 2014), MIT Press, Cambridge MA, pp. 3104-3112.

[10] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is All you Need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, & R. Garnett (Eds.), Advances in Neural Information Processing Systems (Vol. 30). Curran Associates, Inc, Red Hook NY , pp. 6000-6010.

[11] Schmidhuber, J., 2000, October. Evolutionary computation versus reinforcement learning. In 2000 26th Annual Conference of the IEEE Industrial Electronics Society. IECON 2000. 2000 IEEE International Conference on Industrial Electronics, Control and Instrumentation. 21st Century Technologies (Vol. 4, pp. 2992-2997). IEEE.

[12] Sutton, R. S., & Barto, A. G. (2018). Reinforcement learning: An introduction. MIT press, Cambridge MA

[13] Freeman, C. D., Frey, E., Raichuk, A., Girgin, S., Mordatch, I., & Bachem, O. (2021). Brax - A Differentiable Physics Engine for Large Scale Rigid Body Simulation. CoRR, abs/2106.13281.

[14] Whitley, L. D. (1991). Fundamental Principles of Deception in Genetic Search (G. J. E. Rawlins, Red). In (bll 221–241). doi:10.1016/B978-0-08-050684-5.50017-3

[15] Chatzilygeroudis, K., Vassiliades, V., & Mouret, J.-B. (2018). Reset-free Trial-and-Error Learning for Robot Damage Recovery. Robotics and Autonomous Systems, 100, 236–250. doi:10.1016/j.robot.2017.11.010

[16] M. Duarte, J. Gomes, S. M. Oliveira and A. L. Christensen, "Evolution of Repertoire-Based Control for Robots With Complex Locomotor Systems," in IEEE Transactions on Evolutionary Computation, vol. 22, no. 2, pp. 314-328, April 2018, doi: 10.1109/TEVC.2017.2722101.

[17] Cully, A., & Mouret, J.-B. (2013). Behavioral Repertoire Learning in Robotics. Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation, 175–182.

[18] Vassiliades, V., Chatzilygeroudis, K., & Mouret, J.-B. (2017). A Comparison of Illumination Algorithms in Unbounded Spaces. Proceedings of the Genetic and Evolutionary Computation Conference Companion, 1578–1581. Presented at the Berlin, Germany. doi:10.1145/3067695.3082531

[19] Vassiliades, V., & Mouret, J.-B. (2018). Discovering the Elite Hypervolume by Leveraging Interspecies Correlation. CoRR, abs/1804.03906.

[20] Quality-Diversity Optimisation algorithms: https://quality-diversity.github.io/

[21] Rudin, N., Hoeller, D., Reist, P., & Hutter, M. (2021). Learning to Walk in Minutes Using Massively Parallel Deep Reinforcement Learning. arXiv preprint arXiv:2109.11978.

[22] Heiden, E., Millard, D., Coumans, E., Sheng, Y., & Sukhatme, G. S. (2020). NeuralSim: Augmenting differentiable simulators with neural networks. arXiv preprint arXiv:2011.04217.

[23] Differentiable Quality Diversity Algorithms, Code and Experiments: https://github.com/icaros-usc/dqd

[24] Makoviychuk, V., Wawrzyniak, L., Guo, Y., Lu, M., Storey, K., Macklin, M., ... & State, G. (2021). Isaac Gym: High

[26] J. Clune, J-B. Mouret, and H. Lipson. The evolutionary origins of modularity. Proceedings of the Royal Society B, 280(20122863), 2013.

[27] Deb K. 2001 Multi-objective optimization using evolutionary algorithms. New York, NY: Wiley.

[28] Jerome H Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. ACM Transactions on Mathematical Software (TOMS), 3(3):209– 226, 1977.

[29] Antoine Cully and Yiannis Demiris. Quality and diversity optimization: A unifying modular framework. IEEE Transactions on Evolutionary Computation, 22(2):245– 259, 2018

[30] Adam Gaier, Alexander Asteroth, and Jean-Baptiste Mouret. Aerodynamic design exploration through surrogate-assisted illumination. In 18th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, page 3330, 2017.

[31] Niels Justesen, Sebastian Risi, and Jean-Baptiste Mouret. Map-elites for noisy domains by adaptive sampling. In Proceedings of the Genetic and Evolutionary Computation Conference Companion, pages 121–122. ACM, 2019.

[32] Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O Stanley, and Jeff Clune. Go-explore: a new approach for hard-exploration problems. arXiv preprint arXiv:1901.10995, 2019.

[33] Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O Stanley, and Jeff Clune. First return then explore. arXiv preprint arXiv:2004.12919, 2020.

[34]Nikolaus Hansen, Anne Auger, Raymond Ros, Steffen Finck, and Petr PoÅąÃŋk. 2010. Comparing Results of 31 Algorithms from the Black-Box Optimization Benchmarking BBOB-2009. Proceedings of the 12th Annual Genetic and Evolutionary Computation Conference, GECCO '10 - Companion Publication, 1689–1696.

[35] Hansen, N. (2016). The CMA Evolution Strategy: A Tutorial. doi:10.48550/ARXIV.1604.00772

[36] D. Floreano and C. Mattiussi. Bio-inspired artificial intelligence: theories, methods, and technologies. The MIT Press, 2008.

[37] J. Clune, K.O. Stanley, R.T. Pennock, and C. Ofria. On the performance of indirect encoding across the continuum of regularity. IEEE Transactions on Evolutionary Computation, 15(4):346–367, 2011.

[38] K.O. Stanley and R. Miikkulainen. A taxonomy for artificial embryogeny. Artificial Life, 9(2):93–130, 2003.

[39] N. Cheney, R. MacCurdy, J. Clune, and H. Lipson. Unshackling evolution: Evolving soft robots with multiple materials and a powerful generative encoding. In Proceedings of the Genetic and Evolutionary Computation Conference, pages 167–174, 2013.

[40] J. Yosinski, J. Clune, D. Hidalgo, S. Nguyen, J.C. Zagal, and H. Lipson. Evolving robot gaits in hardware: the hyperneat generative encoding vs. parameter optimization. In Proceedings of the European Conference on Artificial Life, pages 890–897, 2011.

[41] S. Lee, J. Yosinski, K. Glette, H. Lipson, and J. Clune. Evolving gaits for physical robots with the hyperneat generative encoding: the benefits of simulation. In Applications of Evolutionary Computing. Springer, 2013.

[42] J. Clune, B.E. Beckmann, C. Ofria, and R.T. Pennock. Evolving coordinated quadruped gaits with the HyperNEAT generative encoding. In Proceedings of the IEEE Congress Evolutionary Computation, pages 2764–2771, 2009.

[43] G.S. Hornby, H. Lipson, and J.B. Pollack. Generative representations for the automated design of modular physical robots. IEEE Transactions on Robotics and Automation, 19(4):703–719, 2003.

[44] G.S. Hornby and J.B. Pollack. Creating high-level components with a generative representation for body-brain evolution. Artificial Life, 8(3):223–246, 2002.

[45] G.S. Hornby. Functional scalability through generative representations: the evolution of table designs. Environment and Planning B, 31(4):569–588, 2004.

[46] J. Lehman and K. O. Stanley. 2008. Exploiting Open-Endedness to Solve Problems Through the Search for Novelty. In ALIFE. 329–336.

[47] N. Hansen and A. Ostermeier. 2001. Completely derandomized self-adaptation in evolution strategies. Evol. Comput. 9, 2 (2001), 159–195.

[48] Yoeng-Jin Chu. On the shortest arborescence of a directed graph. Scientia Sinica, 14:1396–1400, 1965.

[49] Youhei Akimoto, Yuichi Nagata, Isao Ono, and Shigenobu Kobayashi. Bidirectional relation between cma evolution strategies and natural evolution strategies. In International Conference on Parallel Problem Solving from Nature, pages 154–163. Springer, 2010.

[50] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings, 2015.

[51] Rumelhart, D. E., Hinton, G. E. & Williams, R. J. (1986). Learning Representations by Back-propagating Errors. Nature, 323, 533--536. doi: 10.1038/323533a0

[52] Zeiler, M. D. (2012). ADADELTA: An Adaptive Learning Rate Method. CoRR, abs/1212.5701.

[53] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. page 14, 2012.

[54] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018).

[55] Antoine Cully and Yiannis Demiris. 2017. Quality and diversity optimization: A unifying modular framework. IEEE Transactions on Evolutionary Computation 22, 2 (2017), 245–259.

[56] Justin K Pugh, Lisa B Soros, and Kenneth O Stanley. 2016. Quality diversity: A new frontier for evolutionary computation. Frontiers in Robotics and AI 3 (2016), 40.

[57] Konstantinos Chatzilygeroudis, Vassilis Vassiliades, and Jean-Baptiste Mouret. 2018. Reset-free trial-and-error learning for robot damage recovery. Robotics and Autonomous Systems 100 (2018), 236–250.

[58] Rituraj Kaushik, Pierre Desreumaux, and Jean-Baptiste Mouret. 2020. Adaptive prior selection for repertoire-based online adaptation in robotics. Frontiers in Robotics and AI 6 (2020), 151.

[59] Dave Steinkrau, Patrice Y. Simard, and Ian Buck. 2005. Using GPUs for Machine Learning Algorithms. In Proceedings of the Eighth International Conference on Document Analysis and Recognition (ICDAR '05). IEEE Computer Society, USA, 1115–1119. https://doi.org/10.1109/ICDAR.2005.251

[60] Dan C. Ciresan, Ueli Meier, and Jürgen Schmidhuber. 2012. Multi-column Deep Neural Networks for Image Classification. CoRR abs/1202.2745 (2012). arXiv:1202.2745 http://arxiv.org/abs/1202.2745

[61] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). 265–283. https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[62] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. 2011. Torch7: A Matlab-like Environment for Machine Learning. http://infoscience.epfl.ch/record/192376

[63] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zach DeVito, Martin Raison, Alykhan

Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. CoRR abs/1912.01703 (2019). arXiv:1912.01703 http://arxiv.org/abs/1912. 01703

[64] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. JAX: composable transformations of Python+NumPy programs. http://github.com/google/jax

[65] Tjanaka, B., Fontaine, M. C., & Nikolaidis, S. (2021). Learning a Repertoire of Robot Arm Configurations. https://docs.pyribs.org/en/stable/tutorials/arm_repertoire.html

[66] Nikolaus Hansen, Youhei Akimoto, and Petr Baudis. CMA-ES/pycma on Github. Zenodo, DOI:10.5281/zenodo.2559634, February 2019. URL https://doi.org/10.5281/zenodo. 2559634.

[67] C. Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. 2021. Brax - A Differentiable Physics Engine for Large Scale Rigid Body Simulation. http://github.com/google/brax

[68] Viktor Makoviychuk, Lukasz Wawrzyniak, Yunrong Guo, Michelle Lu, Kier Storey, Miles Macklin, David Hoeller, Nikita Rudin, Arthur Allshire, Ankur Handa, and Gavriel State. 2021. Isaac Gym: High Performance GPU-Based Physics Simulation For Robot Learning. CoRR abs/2108.10470 (2021). arXiv:2108.10470 https://arxiv.org/abs/2108.10470

[69] Lim, B., Allard, M., Grillotti, L., & Cully, A. (2022). Accelerated Quality-Diversity for Robotics through Massive Parallelism. doi:10.48550/ARXIV.2202.01258

Appendix A - Implementations

A.1 QD Functions

A.1.1 Rastrigin Objective Function

```python
import jax
from jax import lax
import jax.numpy as jnp

def calc_rastrigin(sol):
    C = 5.12
    A = 10.0
    dim = sol.shape[1]

    # Shift the Rastrigin function so that the optimal value is at x_i = 2.048.
    # This is to avoid having the lowest point of the rastrigin functionat
    # position x = [0,0,0,...,0], i.e. f(0,0,0,...,0) = 0 because the position
    # x = [0,0,0,...,0] is usually used as an initial point for the search
    # That is, an initial solution
    target_shift = C * 0.4
    # sol = 2*C*sol - C
    best_obj = jnp.zeros(len(sol))
    displacement = -C * jnp.ones(sol.shape) - target_shift
    sum_terms = jnp.square(displacement) - A * jnp.cos(2 * jnp.pi * displacement)
    worst_obj = A * dim + jnp.sum(sum_terms, axis=1)

    displacement = sol - target_shift
    sum_terms = jnp.square(displacement) - A * jnp.cos(2 * jnp.pi * displacement)
    raw_obj = A * dim + jnp.sum(sum_terms, axis=1)

    # Normalize the objective to the range [0, 100] where 100 is optimal.
    # Approximate 0 by the bottom-left corner.
    objs = (raw_obj - worst_obj) / (best_obj - worst_obj) * 100

    derivatives = -(2 * displacement + 2 * jnp.pi * A * jnp.sin(2 * jnp.pi * displacement))

    return objs, derivatives
```

Code Snippet A.1: Objective Function Rastrigin(x) defined by the shifted and flipped Rastrigin Function

A.1.2 Simple Behavioural Function – simple_b(x)

```
import jax
from jax import lax
import jax.numpy as jnp

def calc_bds_rastrigin_simple(sol):
    mask_1 = jnp.zeros(sol.shape).at[:,0].set(1)
    mask_2 = jnp.zeros(sol.shape).at[:,1].set(1)
    return ( jnp.stack((sol[:,0], sol[:,1]), axis=-1),
                jnp.stack((mask_1, mask_2), axis=1)
           )
```

Code Snippet A.2: Simple Behavioural Function b_simple(x)

A.1.3 Behavioural Function distorted_b(x)

```
import jax
from jax import lax
import jax.numpy as jnp

def calc_bds_rastrigin(sol):
    C = 5.12
    dim = sol.shape[1]

    mask_greater = jnp.where(sol > C, 1, 0)
    mask_less = jnp.where(sol < -C, 1, 0)

    mask_range=jnp.invert((mask_greater+mask_less).astype(dtype=bool)).astype(dtype=jnp.int32)

    clipped = (mask_greater + mask_less) * (C / sol) + sol * mask_range
    measures = jnp.concatenate(
        (
            jnp.sum(clipped[:, :dim // 2], axis=1, keepdims=True),
            jnp.sum(clipped[:, dim // 2:], axis=1, keepdims=True),
        ),
        axis=1,
    )

    derivatives = (mask_greater + mask_less) * ( -C / jnp.square(sol)) + mask_range
```

```python
    mask_0 = jnp.concatenate((jnp.ones(dim//2), jnp.zeros(dim-dim//2)))
    mask_1 = jnp.concatenate((jnp.zeros(dim//2), jnp.ones(dim-dim//2)))

    d_measure0 = jnp.multiply(derivatives, mask_0)
    d_measure1 = jnp.multiply(derivatives, mask_1)

    jacobian = jnp.stack((d_measure0, d_measure1), axis=1)
    return measures, jacobian
```

Code Snippet A.3: Behavioural Function distorted_b(x)

A.1.4 Objective Function grasp_obj(x)

```python
import jax
from jax import lax
import jax.numpy as jnp

def calc_grasp_objs(joint_angles, link_lengths, calc_jacobians=True):

    n_dim = link_lengths.shape[0]
    objs = -jnp.var(joint_angles, axis=1)

    # Remap the objective from [-1, 0] to [0, 100]
    objs = (objs+1.0)*100.0

    if calc_jacobians:
        means = jnp.mean(joint_angles, axis=1)
        means = jnp.expand_dims(means, axis=1)
        base = n_dim * jnp.ones(n_dim)
        obj_derivatives = -2 * (joint_angles - means) / base

        return objs, obj_derivatives
    return objs, None
```

Code Snippet A.4: Objective Function grasp(x)

A.1.5 Behavioural Function grasp_b(x)

```python
import jax
from jax import lax
import jax.numpy as jnp

def _step_calc_bd_gradients(data):
    i, bds_derivatives, link_lengths, cum_theta, sum_0, sum_1 = data

    sum_0 += -link_lengths[i] * jnp.sin(cum_theta[:, i])
    sum_1 += link_lengths[i] * jnp.cos(cum_theta[:, i])
    bds_derivatives.at[:, 0, i].set(sum_0)
    bds_derivatives.at[:, 1, i].set(sum_1)

    return (i - 1, bds_derivatives, link_lengths, cum_theta, sum_0, sum_1 )

def calc_grasp_bds(joint_angles, link_lengths, calc_jacobians=True):
    # max_val = jnp.sum(link_lengths)
    # joint_angles = joint_angles * 2*max_val - max_val
    n_dim = link_lengths.shape[0]

    # theta_1, theta_1 + theta_2, ...
    cum_theta = jnp.cumsum(joint_angles, axis=1)
    # l_1 * cos(theta_1), l_2 * cos(theta_1 + theta_2), ...
    x_pos = link_lengths[None] * jnp.cos(cum_theta)
    # l_1 * sin(theta_1), l_2 * sin(theta_1 + theta_2), ...
    y_pos = link_lengths[None] * jnp.sin(cum_theta)

    bds = jnp.concatenate(
        (
            jnp.sum(x_pos, axis=1, keepdims=True),
            jnp.sum(y_pos, axis=1, keepdims=True),
        ),
        axis=1
    )

    if calc_jacobians:
        sum_0 = jnp.zeros(joint_angles.shape[0])
        sum_1 = jnp.zeros(joint_angles.shape[0])

        bds_derivatives = jnp.zeros((joint_angles.shape[0], 2, n_dim))
        data = (n_dim-1, bds_derivatives, link_lengths, cum_theta, sum_0, sum_1)
        lax.while_loop(lambda d: d[0] >= 0,
            _step_calc_bd_gradients,
            data
        )

        return bds, bds_derivatives
    return bds, None
```

Code Snippet A.5: Behavioural Function grasp_b(x)

A.2 Optimizers

A.2.1 Covariance Matrix Adaptation Evolution Strategy (CMA-ES)

```python
"""Implementation of CMA-ES that can be used across various emitters.
Adapted from Nikolaus Hansen's pycma:
https://github.com/CMA-ES/pycma/blob/master/cma/purecma.py
"""
from typing import Any, Callable, Dict, Optional, Tuple, List
import jax.numpy as jnp
import flax
import jax
from jax import lax
import functools

D_TYPE = jnp.float32

Array = jnp.ndarray

class WeightRules:
    TRUNCATION = 1
    ACTIVE = 2

@flax.struct.dataclass
class DecompMatrix:
    """Maintains a covariance matrix and its eigendecomposition.
    CMA-ES requires the inverse square root of the covariance matrix in order to
    sample new solutions from a multivariate normal distribution. However,
    calculating the inverse square root is an O(n^3) operation because an
    eigendecomposition is involved. (n is the dimensionality of the search
    space). To amortize the operation to O(n^2) and avoid recomputing, this
    class maintains the inverse square root and waits several evals before
    recomputing the inverse square root.
    """

    cov: Array
    eigenbasis: Array
    eigenvalues: Array
    condition_number: jnp.float32
    invsqrt: Array
    updated_eval: jnp.int32

    @classmethod
    def create(cls, dimension):
        cov = jnp.eye(dimension, dtype=D_TYPE)
        eigenbasis = jnp.eye(dimension, dtype=D_TYPE)
        eigenvalues = jnp.ones((dimension,), dtype=D_TYPE)
        condition_number = 1.0
        invsqrt = jnp.eye(dimension, dtype=D_TYPE)  # C^(-1/2)
        # The last evaluation on which the eigensystem was updated.
        updated_eval = 0
        return cls(cov, eigenbasis, eigenvalues, condition_number, invsqrt, updated_eval)

    @staticmethod
    def check_update_eigensystem(decompMatrix, current_eval, lazy_gap_evals):
        return lax.cond(current_eval <= decompMatrix.updated_eval + lazy_gap_evals,
                lambda x: decompMatrix,
                lambda x: DecompMatrix._update_eigensystem(decompMatrix, current_eval),
                    current_eval
            )

    @staticmethod
    def _update_eigensystem(decompMatrix, current_eval):
        """Updates the covariance matrix.
        """
        # Force symmetry.
        cov = jnp.maximum(decompMatrix.cov, jnp.transpose(decompMatrix.cov))

        # Note: eigh returns float64, so we must cast it.
        eigenvalues, eigenbasis = jnp.linalg.eigh(cov)
        eigenvalues = eigenvalues.real.astype(D_TYPE)
        eigenbasis = eigenbasis.real.astype(D_TYPE)
        condition_number = (jnp.max(eigenvalues) /
                            jnp.min(eigenvalues))
        invsqrt = jnp.matmul( (eigenbasis *
                    (1 / jnp.sqrt(eigenvalues))), jnp.transpose(eigenbasis))

        # Force symmetry.
        invsqrt = jnp.maximum(invsqrt, jnp.transpose(invsqrt))

        updated_eval = current_eval
        return decompMatrix.replace(cov = cov,
                          eigenbasis = eigenbasis,
                          eigenvalues = eigenvalues,
                          condition_number = condition_number,
                          invsqrt = invsqrt,
                          updated_eval = updated_eval)

@flax.struct.dataclass
class CMAEvolutionStrategy:
    """CMA-ES optimizer for use with emitters.
    The basic usage is:
    - Initialize the optimizer and reset it.
    - Repeatedly:
      - Request new solutions with ask()
      - Rank the solutions in the emitter (better solutions come first) and pass
        them back with tell().
      - Use check_stop() to see if the optimizer has reached a stopping
        condition, and if so, call reset().
    """
    batch_size: jnp.int32
    sigma0: jnp.float32
    solution_dim: jnp.int32
    lazy_gap_evals: jnp.float32
    current_eval: jnp.int32
    mean: jnp.float32
    sigma: jnp.float32
    pc: Array
    ps: Array
    cov: DecompMatrix
    weight_rule: str

    @classmethod
    def create(cls, sigma0, batch_size, solution_dim, weight_rule):
        batch_size = (4 + int(3 * jnp.log(solution_dim))
                      if batch_size is None else batch_size)
        sigma0 = float(sigma0)

        if weight_rule not in [WeightRules.TRUNCATION, WeightRules.ACTIVE]:
            raise ValueError(f"Invalid weight_rule {weight_rule}")



        calc_strat_params_fn = jax.jit(functools.partial(
        CMAEvolutionStrategy._calc_strat_params, weight_rule, batch_size, solution_dim))
        static_settings = dict()
        static_settings['calc_strat_params_fn'] = calc_strat_params_fn
        static_settings['solution_dim'] = solution_dim
        static_settings['batch_size'] = batch_size
        static_settings['should_update_eigensystem'] = False

        # Calculate gap between covariance matrix updates.
        num_parents = batch_size // 2
        temp_indices = jnp.arange(0, batch_size, 1)
        parents_mask = jnp.where(temp_indices < num_parents, 1, 0)
        *_, c1, cmu = calc_strat_params_fn(num_parents, parents_mask)
        lazy_gap_evals = (0.5 * solution_dim * batch_size *
                          (c1 + cmu)**-1 / solution_dim**2)

        # Strategy-specific params -> initialized in reset().
        current_eval = None
        mean = None
        sigma = None
        pc = None
        ps = None
        cov = None

        return cls(batch_size, sigma0, solution_dim, lazy_gap_evals, current_eval,
                   mean, sigma, pc, ps, cov, weight_rule), static_settings

    @staticmethod
    def _get_batch_size(static_settings):
        return static_settings['batch_size']

    @staticmethod
    def _get_solution_dim(static_settings):
        return static_settings['solution_dim']

    @staticmethod
    def _get_calc_strat_params_fn(static_settings):
        return static_settings['calc_strat_params_fn']

    @staticmethod
    def reset(static_settings, cmaEvolStrategy, x0):
        """Resets the optimizer to start at x0.
        Args:
            x0 (jnp.ndarray): Initial mean.
        """
        solution_dim = cmaEvolStrategy._get_solution_dim(static_settings)
        current_eval = 0
        sigma = cmaEvolStrategy.sigma0
        mean = jnp.array(x0, D_TYPE)

        # Setup evolution path variables.
        pc = jnp.zeros(solution_dim, dtype=D_TYPE)
        ps = jnp.zeros(solution_dim, dtype=D_TYPE)

        # Setup the covariance matrix.
        cov = DecompMatrix.create(solution_dim)

        return cmaEvolStrategy.replace(cov = cov, pc = pc, ps = ps, sigma = sigma,
                                       mean = mean, current_eval = current_eval)

    @staticmethod
    def check_stop(cmaEvolStrategy, ranking_values, new_sols):
        """Checks if the optimization should stop and be reset.
        Tolerances come from CMA-ES.
        """
        area = cmaEvolStrategy.sigma * jnp.sqrt(jnp.asarray(
                                           cmaEvolStrategy.cov.eigenvalues).max())

        a = cmaEvolStrategy.cov.condition_number > 1e14
        b = area < 1e-11
        c = new_sols >= 2
        d = jnp.abs(ranking_values[0] - ranking_values[-1]) < 1e-12

        # condition number > 1e14 or
        # Area of distribution too small or
        # Fitness is too flat (only applies if there are at least 2 parents).
        # return cmaEvolStrategy.cov.condition_number > 1e14 or area < 1e-11 or (new_sols >= 2
and
        #           jnp.abs(ranking_values[0] - ranking_values[-1]) < 1e-12)
        return jnp.logical_or(jnp.logical_or(a,b),jnp.logical_and(c,d))


    @staticmethod
    def _transform_and_check_sol( transform_mat, mean, lower_bounds, upper_bounds,
                                  unscaled_params):
        """Helper for transforming parameters to the solution space."""
        solutions = (jnp.transpose(jnp.matmul(transform_mat,
                        jnp.transpose(unscaled_params))) +
                     jnp.expand_dims(mean, axis=0))
        out_of_bounds = jnp.logical_or(
            solutions < jnp.expand_dims(lower_bounds, axis=0),
            solutions > jnp.expand_dims(upper_bounds, axis=0),
        )
        return solutions, out_of_bounds

    @staticmethod
    def _get_remaining_out_of_bounds(batch_size, solution_dim, sigma,
                                           _transform_and_check_sol_fn, data):
        # jax.jit requires that not working with code that generate output array shape that is
data-dependent
        # (e.g. use jnp.where to find indices of an array that satisfy a condition)
        # Thus, we generate deterministically each time as many solutions as it is the number
of batch sizes
        # and we exploit only a fraction of it. Should be investigated whether it can be
optimised further
        out_of_bounds_all, solutions, key = data
        tempkey, key2 = jax.random.split(key, 2)
        unscaled_params = sigma * jax.random.normal(key2,
                                    shape=solutions.shape,
                                    dtype=D_TYPE)

        new_solutions, out_of_bounds = _transform_and_check_sol_fn(unscaled_params)
        # Change the solutions that previously were out of bounds
        mask = jnp.repeat(jnp.expand_dims(out_of_bounds_all,axis=-1), solution_dim, axis=-1)
        solutions = jnp.where(mask, new_solutions, solutions)

        # update which solutions are still out of bounds
        out_of_bounds_all = jnp.logical_and(
                out_of_bounds_all,
                jnp.any(out_of_bounds, axis=1),
            )
        return (out_of_bounds_all, solutions , tempkey)

    @staticmethod
    def _more_out_of_bounds(data):
        return jnp.sum(data[0].astype(jnp.int32)) > 0

    @staticmethod
    def ask(static_settings, cmaEvolStrategy, lower_bounds, upper_bounds, seed):
        """Samples new solutions from the Gaussian distribution.
        """
        solution_dim = cmaEvolStrategy._get_solution_dim(static_settings)
        batch_size = cmaEvolStrategy._get_batch_size(static_settings)
        decompMatrix = DecompMatrix.check_update_eigensystem(cmaEvolStrategy.cov,
                       cmaEvolStrategy.current_eval, cmaEvolStrategy.lazy_gap_evals)

        solutions = jnp.empty((batch_size, solution_dim),dtype=D_TYPE)
        transform_mat = decompMatrix.eigenbasis * jnp.sqrt(decompMatrix.eigenvalues)
        cmaEvolStrategy = cmaEvolStrategy.replace(cov = decompMatrix)

        # keeps a flag (= True or False) for each solution whether any of its parameter's
value is out of bounds
        out_of_bounds_all = jnp.full(batch_size,True)
        tempkey = seed
        # to_ones = jnp.vectorize(lambda x: lax.cond(x, lambda x: 1, lambda x: 0, x))
        _transform_and_check_sol_partial = functools.partial(
                             CMAEvolutionStrategy._transform_and_check_sol,
                             transform_mat, cmaEvolStrategy.mean, lower_bounds, upper_bounds)

        get_remaining_out_of_bounds_fn =
functools.partial(CMAEvolutionStrategy._get_remaining_out_of_bounds,
                                                            batch_size,
                                                            solution_dim,
                                                            cmaEvolStrategy.sigma,
                                                            _transform_and_check_sol_partial)

        return cmaEvolStrategy, jnp.asarray(
lax.while_loop(CMAEvolutionStrategy._more_out_of_bounds,
                                                get_remaining_out_of_bounds_fn,
                                            (out_of_bounds_all, solutions, tempkey))[1]
                                        )

        # Resampling method for bound constraints -> sample new solutions until
        # all solutions are within bounds.
        while jnp.sum(out_of_bounds_all.astype(jnp.int32)) > 0:
            # jax.jit requires that not working with code that generate output array shape
that is data-dependent
            # (e.g. use jnp.where to find indices of an array that satisfy a condition)
            # Thus, we generate deterministically each time as many solutions as it is the
number of batch sizes
            # and we exploit only a fraction of it. Should be investigated whether it can be
optimised further
            tempkey, key2 = jax.random.split(tempkey, 2)
            unscaled_params = cmaEvolStrategy.sigma * jax.random.normal(key2,
                                    shape=(cmaEvolStrategy.batch_size,
                                    cmaEvolStrategy.solution_dim),
                                    dtype=cmaEvolStrategy.dtype)

            new_solutions, out_of_bounds = _transform_and_check_sol_partial(unscaled_params)
            # Change the solutions that previously were out of bounds
            mask = jnp.repeat(jnp.expand_dims(out_of_bounds_all,axis=-1),
                                   cmaEvolStrategy.solution_dim,axis=-1)
            solutions = jnp.where(mask,new_solutions,solutions)

            # update which solutions are still out of bounds
            out_of_bounds_all = jnp.logical_and(
                    out_of_bounds_all,
                    jnp.any(out_of_bounds, axis=1),
                )

        return jnp.asarray(solutions)

    @staticmethod
    def _calc_strat_params(weight_rule, batch_size, solution_dim, num_parents, parents_mask):
        """Calculates weights, mueff, and learning rates for CMA-ES."""
        # Create fresh weights for the number of parents found.
        if weight_rule == WeightRules.TRUNCATION:
            # The first num_parents weights are used that depend on the number of parents
            # but the array has size equal to solution_dim to allow it to be jit compiled
            weights = (jnp.log(num_parents + 0.5) -
                       jnp.log(jnp.arange(1, batch_size + 1)))
            # make the non-parent entries to zero arrays so that the sum is not affected
            # by the extra entries in the array (elements that are not parents)
            filtered_weights = jnp.multiply(weights, parents_mask)
            total_weights = jnp.sum(filtered_weights)
            weights = filtered_weights / total_weights
            # mueff = jnp.sum(weights)**2 / jnp.sum(weights**2)
            mueff = jnp.square(total_weights) / jnp.sum(jnp.square(filtered_weights))
        elif weight_rule == WeightRules.ACTIVE:
            weights = None

        # Dynamically update these strategy-specific parameters.
        cc = ((4 + mueff / solution_dim) /
              (solution_dim + 4 + 2 * mueff / solution_dim))
        cs = (mueff + 2) / (solution_dim + mueff + 5)
        c1 = 2 / ((solution_dim + 1.3)**2 + mueff)
        cmu = jnp.minimum(
            1 - c1,
            2 * (mueff - 2 + 1 / mueff) / ((solution_dim + 2)**2 + mueff),
        )
        return weights, mueff, cc, cs, c1, cmu


    @staticmethod
    def _calc_mean(solutions, parents_mask, weights):
        """Helper for calculating the new mean."""
        masked_solutions = jnp.multiply(solutions, jnp.expand_dims(parents_mask, axis=-1))
        return jnp.sum(jnp.multiply(masked_solutions, jnp.expand_dims(weights, axis=1)),
                                   axis=0)

    @staticmethod
    def _calc_weighted_ys(solutions, parents_mask, old_mean, weights):
        """Calculates y's for use in rank-mu update."""
        ys = solutions - jnp.expand_dims(old_mean, axis=0)

        expanded_parents_mask = jnp.expand_dims(parents_mask,axis=-1)
        masked_ys = jnp.multiply(ys, expanded_parents_mask)
        masked_weighted_ys = jnp.multiply(masked_ys, jnp.expand_dims(weights, axis=1))

        return masked_weighted_ys, masked_ys

    @staticmethod
    def _calc_cov_update(cov, c1a, cmu, c1, pc, sigma, rank_mu_update):
        """Calculates covariance matrix update."""
        rank_one_update = c1 * jnp.outer(pc, pc)
        return (cov * (1 - c1a - cmu) + rank_one_update * c1 +
                rank_mu_update * cmu / (sigma**2))

    @staticmethod
    def tell(static_settings, cmaEvolStrategy, solutions, num_parents):
        """Passes the solutions back to the optimizer.
        """
        current_eval = cmaEvolStrategy.current_eval + len(solutions)
        def pass_solutions(static_settings, cmaEvolStrategy, solutions, num_parents,
                            current_eval):
            # parents = solutions[:num_parents]
            calc_strat_params_fn = cmaEvolStrategy._get_calc_strat_params_fn(static_settings)
            solution_dim = cmaEvolStrategy._get_solution_dim(static_settings)
            batch_size = cmaEvolStrategy._get_batch_size(static_settings)

            temp_indices = jnp.arange(0,batch_size, 1)
            parents_mask = jnp.where(temp_indices < num_parents, 1, 0)
            weights, mueff, cc, cs, c1, cmu = calc_strat_params_fn(num_parents, parents_mask)

            damps = (1 + 2 * jnp.maximum(
                0,
                jnp.sqrt((mueff - 1) / (solution_dim + 1)) - 1,
            ) + cs)

            # Recombination of the new mean.
            old_mean = cmaEvolStrategy.mean
            mean = CMAEvolutionStrategy._calc_mean(solutions, parents_mask, weights)

            # Update the evolution path.
            y = mean - old_mean
            z = jnp.matmul(cmaEvolStrategy.cov.invsqrt, y)
            ps = ((1 - cs) * cmaEvolStrategy.ps +
                    (jnp.sqrt(cs * (2 - cs) * mueff) / cmaEvolStrategy.sigma) * z)
            left = (jnp.sum(jnp.square(ps)) / solution_dim /
                    (1 - (1 - cs)**(2 * current_eval / batch_size)))
            right = 2 + 4. / (solution_dim + 1)

            hsig = lax.cond(left < right, lambda x: 1, lambda x: 0, None)

            pc = ((1 - cc) * cmaEvolStrategy.pc + hsig * jnp.sqrt(cc * (2 - cc) * mueff) *
y)

            # Adapt the covariance matrix.
            weighted_ys, ys = CMAEvolutionStrategy._calc_weighted_ys(solutions, parents_mask,
                                                                      old_mean, weights)
            # Equivalent to calculating the outer product of each ys[i] with itself
            # and taking a weighted sum of the outer products. Unfortunately, numba
            # does not support einsum.
            rank_mu_update = jnp.einsum("ki,kj", jnp.asarray(weighted_ys, dtype=D_TYPE),
                                        jnp.asarray(ys, dtype=D_TYPE))

            c1a = c1 * (1 - (1 - hsig**2) * cc * (2 - cc))
            cov = CMAEvolutionStrategy._calc_cov_update(cmaEvolStrategy.cov.cov, c1a, cmu, c1,
                                                        pc, cmaEvolStrategy.sigma,
                                                        rank_mu_update)
            cov = cmaEvolStrategy.cov.replace(cov = cov)
            # Update sigma.
            cn, sum_square_ps = cs / damps, jnp.sum(jnp.square(ps))
            sigma = cmaEvolStrategy.sigma * jnp.exp(
                jnp.minimum(1,cn * (sum_square_ps / solution_dim - 1) / 2)
                )

            return cmaEvolStrategy.replace(cov=cov, current_eval = current_eval, mean = mean,
ps = ps, pc = pc, sigma = sigma)

        # Examine whether num_parents == 0 is required as a check
        return lax.cond(num_parents == 0,
                        lambda x: cmaEvolStrategy.replace(current_eval = current_eval),
                        lambda x: pass_solutions(static_settings, cmaEvolStrategy, solutions,
num_parents, current_eval),
                        None
                    )
```

Code Snippet A.6: Implementation of the CMA-ES in JAX (_cma_es.py)

A.2.2 ADAM

```
# Adapted from: https://github.com/icaros-usc/dqd/blob/main/ribs/emitters/opt/_adam.py
from typing import Any
import jax.numpy as jnp
import jax.numpy as jnp
import flax
import jax
from jax import lax
import functools

D_TYPE = jnp.float32
Array = jnp.ndarray

# Adam Gradient Ascent
@flax.struct.dataclass
class AdamOpt:
    beta1: D_TYPE
    beta2: D_TYPE
    stepsize: jnp.int32
    dim: jnp.int32
    t: D_TYPE
    epsilon: D_TYPE
    theta: Array
    m: Array
    v: Array
```

```
@classmethod
def create(cls, theta0, stepsize, betas=(0.9, 0.999), epsilon=1e-8):
    t = 0
    dim = len(theta0)
    beta1 = betas[0]
    beta2 = betas[1]
    # The following attributes will be initialised with the use of reset
    theta = jnp.array([])
    m = jnp.array([])
    v = jnp.array([])
    static_settings = dict()
    static_settings['dim'] = dim
    return cls.reset(static_settings, cls(beta1, beta2, stepsize,
        dim, t, epsilon, theta, m, v), jnp.array(theta0)), static_settings

@staticmethod
def _get_dim(static_settings):
    return static_settings['dim']

@staticmethod
def reset(static_settings, adamOpt, theta0):
    dim = adamOpt._get_dim(static_settings)
    theta = theta0
    m = jnp.zeros(dim, dtype=D_TYPE)
    v = jnp.zeros(dim, dtype=D_TYPE)
    return adamOpt.replace(m = m, v = v, theta = theta)

@staticmethod
def _compute_step(adamOpt, grad):
    a = adamOpt.stepsize * jnp.sqrt(1 - jnp.power(adamOpt.beta2,
                    adamOpt.t)) / (1 - jnp.power(adamOpt.beta1, adamOpt.t))
    m = adamOpt.beta1 * adamOpt.m + (1 - adamOpt.beta1) * grad
    v = adamOpt.beta2 * adamOpt.v + (1 - adamOpt.beta2) * (grad * grad)
    step = a * m / (jnp.sqrt(v) + adamOpt.epsilon)
    return adamOpt.replace(m = m, v = v), step

@staticmethod
def step(static_settings, adamOpt, grad):
    t = adamOpt.t + 1
    adamOpt, step = adamOpt._compute_step(adamOpt.replace(t = t), grad)
    theta = adamOpt.theta + step
    return adamOpt.replace(theta = theta)
```

Code Snippet A.7: Implementation of the Adap Optimiser in JAX (_adam.py)

A.2.3 Gradient Ascent

```
# Adapted from: https://github.com/icaros-usc/dqd/blob/main/ribs/emitters/opt/_adam.py
from typing import Any
import jax.numpy as jnp
import jax.numpy as jnp
import flax
import jax
from jax import lax
import functools

D_TYPE = jnp.float32
Array = jnp.ndarray

# Adam Gradient Ascent
@flax.struct.dataclass
class AdamOpt:
    beta1: D_TYPE
    beta2: D_TYPE
    stepsize: jnp.int32
    dim: jnp.int32
    t: D_TYPE
    epsilon: D_TYPE
    theta: Array
    m: Array
    v: Array

    @classmethod
    def create(cls, theta0, stepsize, betas=(0.9, 0.999), epsilon=1e-8):
        t = 0
        dim = len(theta0)
        beta1 = betas[0]
        beta2 = betas[1]
        # The following attributes will be initialised with the use of reset
        theta = jnp.array([])
        m = jnp.array([])
        v = jnp.array([])
        static_settings = dict()
        static_settings['dim'] = dim
        return cls.reset(static_settings, cls(beta1, beta2, stepsize,
            dim, t, epsilon, theta, m, v), jnp.array(theta0)), static_settings

    @staticmethod
    def _get_dim(static_settings):
        return static_settings['dim']

    @staticmethod
    def reset(static_settings, adamOpt, theta0):
        dim = adamOpt._get_dim(static_settings)
        theta = theta0
        m = jnp.zeros(dim, dtype=D_TYPE)
        v = jnp.zeros(dim, dtype=D_TYPE)
        return adamOpt.replace(m = m, v = v, theta = theta)

    @staticmethod
    def _compute_step(adamOpt, grad):
        a = adamOpt.stepsize * jnp.sqrt(1 - jnp.power(adamOpt.beta2,
                        adamOpt.t)) / (1 - jnp.power(adamOpt.beta1, adamOpt.t))
        m = adamOpt.beta1 * adamOpt.m + (1 - adamOpt.beta1) * grad
        v = adamOpt.beta2 * adamOpt.v + (1 - adamOpt.beta2) * (grad * grad)
        step = a * m / (jnp.sqrt(v) + adamOpt.epsilon)
        return adamOpt.replace(m = m, v = v), step

    @staticmethod
    def step(static_settings, adamOpt, grad):
        t = adamOpt.t + 1
        adamOpt, step = adamOpt._compute_step(adamOpt.replace(t = t), grad)
        theta = adamOpt.theta + step
        return adamOpt.replace(theta = theta)
```

Code Snippet A.8: Implementation of the Gradient Ascent Optimizer in JAX (_adam.py)

A.3 QD Emitters (Array Version)

A.3.1 MAP-Elites (Isotropic Gaussian)

```
"""Provides the GaussianEmitter.
Adapted from https://github.com/icaros-usc/dqd/blob/main/ribs/emitters/_gaussian_emitter.py
"""

from typing import Any
import jax.numpy as jnp
import flax
import jax
from qd_utils.grid_archive import Repertoire
from training.emitters.emitters_utils import EmitterBase
from collections import namedtuple

D_TYPE = jnp.float32
Array = jnp.ndarray
```

```python
@flax.struct.dataclass
class GaussianEmitter:
    """Emits solutions by adding Gaussian noise to existing archive solutions.

    If the archive is empty, calls to :meth:`ask` will generate solutions from a
    user-specified Gaussian distribution with mean ``x0`` and standard deviation
    ``sigma0``. Otherwise, this emitter selects solutions from the archive and
    generates solutions from a Gaussian distribution centered around each
    solution with standard deviation ``sigma0``.

    This is the classic variation operator presented in `Mouret 2015
    <https://arxiv.org/pdf/1504.04909.pdf>`_.

    """
    x0: Array
    sigma0: D_TYPE
    lower_bounds: Array
    upper_bounds: Array
    batch_size: jnp.int32
    solution_dim: jnp.int32


    @classmethod
    def create(cls,
               x0,
               sigma0,
               batch_size,
               bounds=None):
        solution_dim = len(x0)
        x0 = jnp.array(x0, dtype=D_TYPE)
        lower_bounds, upper_bounds = EmitterBase.process_bounds(bounds, solution_dim)
        batch_size = batch_size
        sigma0 = float(sigma0)
        static_settings = dict()
        static_settings['batch_size'] = batch_size
        static_settings['solution_dim'] = solution_dim
        StaticSettings = namedtuple('StaticSettings', static_settings)

        return (cls(x0, sigma0, lower_bounds, upper_bounds, batch_size, solution_dim),
                            StaticSettings(**static_settings))

    @staticmethod
    def _get_batch_size(static_settings):
        return static_settings.batch_size

    @staticmethod
```

```python
def _get_solution_dim(static_settings):
    return static_settings.solution_dim


@staticmethod
def _ask_clip(parents, lower_bounds, upper_bounds):
    return jnp.minimum(jnp.maximum(parents, lower_bounds), upper_bounds)

@staticmethod
def ask(static_settings, gaussian_emitter, repertoire, key):
    """Creates solutions by adding Gaussian noise to elites in the archive.
    """
    batch_size = gaussian_emitter._get_batch_size(static_settings)
    solution_dim = gaussian_emitter._get_solution_dim(static_settings)
    key_selection, key_variation = jax.random.split(key, 2)

    # SELECTION #
    idx_p1 = jax.random.randint(key_selection, shape=(batch_size,), minval=0,
                                    maxval=repertoire.num_indivs)
    tot_indivs = repertoire.fitness.ravel().shape[0]
    indexes = jnp.argwhere(jnp.logical_not(jnp.isnan(repertoire.fitness)), size =
                           tot_indivs)
    indexes = jnp.transpose(indexes, axes=(1, 0))
    indiv_indices = jnp.array(jnp.ravel_multi_index(indexes, repertoire.fitness.shape,
                               mode='clip')).astype(int)

    idx_p1 = indiv_indices.at[idx_p1].get()
    sols = jax.tree_map(lambda x: x.at[idx_p1].get(),repertoire.archive)

    # # VARIATION - MUTATION #
    # # Better approach since it operates directly on the tree
    # # structure of the solutions
    # num_vars = len(jax.tree_leaves(sols))
    # treedef = jax.tree_structure(sols)
    # all_keys = jax.random.split(key_variation, num=num_vars)

    # # Gaussian noise
    # noise = jax.tree_multimap(
    #     lambda g, k: jax.random.normal(k, shape=g.shape, dtype=g.dtype), sols,
    #     jax.tree_unflatten(treedef, all_keys))

    # # Added noise in positive direction
    # mutated_sols = jax.tree_multimap(lambda g, n: g + n * gaussian_emitter.sigma0, sols,
    #                                    noise)
    # # Added noise in negative direction
    # anit_mutated_sols = jax.tree_multimap(lambda g, n: g - n * gaussian_emitter.sigma0,
    #                                         sols, noise)
```

```
        # return gaussian_emitter, mutated_sols

        noise = jax.random.normal(key_variation, shape=(batch_size, solution_dim),
                                         dtype=D_TYPE) * gaussian_emitter.sigma0

        return gaussian_emitter, gaussian_emitter._ask_clip(sols + noise,
                                     gaussian_emitter.lower_bounds,
                                     gaussian_emitter.upper_bounds)

    @staticmethod
    def tell(static_settings, gaussian_emitter, solutions, objective_values, behavior_values,
             dead, repertoire, key):
        """Inserts entries into the archive.
        """
        repertoire = repertoire.add_to_archive(repertoire = repertoire,
                                         pop_p = solutions,
                                         bds = behavior_values,
                                         eval_scores = objective_values,
                                         dead = dead)
        return gaussian_emitter, repertoire
```

Code Snippet A.9: Implementation of the MAP-Elites with Isotropic Gaussian in JAX (_gaussian_emitter.py)

A.3.2 MAP-Elites (Iso + LineDD)

```
"""Provides the IsoLineEmitter.
Adapted from https://github.com/icaros-usc/dqd/blob/main/ribs/emitters/_iso_line_emitter.py
"""

import jax.numpy as jnp
import flax
import jax
from training.emitters.emitters_utils import EmitterBase
from collections import namedtuple

D_TYPE = jnp.float32
Array = jnp.ndarray

@flax.struct.dataclass
class IsoLineEmitter:
    """Emits solutions that are nudged towards other archive solutions.
    If the archive is empty, calls to :meth:`ask` will generate solutions from
    an isotropic Gaussian distribution with mean ``x0`` and standard deviation
    ``iso_sigma``. Otherwise, to generate each new solution, the emitter selects
    a pair of elites :math:`x_i` and :math:`x_j` and samples from
    .. math::
        x_i + \\sigma_{iso} \\mathcal{N}(0,\\mathcal{I}) +
            \\sigma_{line}(x_j - x_i)\\mathcal{N}(0,1)
    This emitter is based on the Iso+LineDD operator presented in `Vassiliades
    2018 <https://arxiv.org/abs/1804.03906>`_.
    """
    x0: Array
    iso_sigma: D_TYPE
    line_sigma: D_TYPE
    lower_bounds: Array
    upper_bounds: Array
    batch_size: jnp.int32
    solution_dim: jnp.int32


    @classmethod
    def create(cls,
                 x0,
                 iso_sigma,
                 line_sigma,
                 batch_size,
                 bounds=None):
        solution_dim = len(x0)
        x0 = jnp.array(x0, dtype=D_TYPE)
        lower_bounds, upper_bounds = EmitterBase.process_bounds(bounds, solution_dim)
        batch_size = batch_size
        iso_sigma = float(iso_sigma)
        line_sigma = float(line_sigma)
        static_settings = dict()
        static_settings['batch_size'] = batch_size
        static_settings['solution_dim'] = solution_dim
        StaticSettings = namedtuple('StaticSettings', static_settings)

        return (cls(x0, iso_sigma, line_sigma, lower_bounds, upper_bounds, batch_size,
                     solution_dim), StaticSettings(**static_settings))

    @staticmethod
    def _get_batch_size(static_settings):
        return static_settings.batch_size

    @staticmethod
    def _get_solution_dim(static_settings):
        return static_settings.solution_dim

    @staticmethod
    def _ask_clip(parents, lower_bounds, upper_bounds):
        return jnp.minimum(jnp.maximum(parents, lower_bounds), upper_bounds)

    @staticmethod
```

```python
def ask(static_settings, iso_emitter, repertoire, key):
    """Generates ``batch_size`` solutions.
    """
    batch_size = iso_emitter._get_batch_size(static_settings)
    solution_dim = iso_emitter._get_solution_dim(static_settings)
    key_selection, key_variation = jax.random.split(key, 2)

    # SELECTION #
    key_select_p1, key_select_p2 = jax.random.split(key_selection, 2)
    idx_s1 = jax.random.randint(key_select_p1, shape=(batch_size,),
                  minval=0, maxval=repertoire.num_indivs)
    idx_s2 = jax.random.randint(key_select_p2, shape=(batch_size,),
                  minval=0, maxval=repertoire.num_indivs)
    tot_indivs = repertoire.fitness.ravel().shape[0]
    indices = jnp.argwhere(jnp.logical_not(jnp.isnan(repertoire.fitness)),
                      size = tot_indivs)
    indices = jnp.transpose(indices, axes=(1, 0))
    indiv_indices = jnp.array(jnp.ravel_multi_index(indices,
                      repertoire.fitness.shape, mode='clip')).astype(int)

    idx_s1 = indiv_indices.at[idx_s1].get()
    idx_s2 = indiv_indices.at[idx_s2].get()
    sols_1 = jax.tree_map(lambda x: x.at[idx_s1].get(),
            repertoire.archive)
    sols_2 = jax.tree_map(lambda x: x.at[idx_s2].get(),
            repertoire.archive)

    # # VARIATION #
    # # Better approach since it operates directly on the tree
    # # structure of the solutions
    # num_vars = len(jax.tree_leaves(sols_1))
    # treedef = jax.tree_structure(sols_1)
    # key_a, key_b = jax.random.split(key_variation, 2)
    # all_keys_a = jax.random.split(key_a, num_vars)
    # all_keys_b = jax.random.split(key_b, num_vars)

    # noise_a = jax.tree_multimap(
    #     lambda g, k: jax.random.normal(k, shape=g.shape, dtype=g.dtype), sols_1,
    #     jax.tree_unflatten(treedef, all_keys_a))
    # noise_b = jax.tree_multimap(
    #     lambda g, k: jax.random.normal(k, shape=g.shape, dtype=g.dtype), sols_2,
    #     jax.tree_unflatten(treedef, all_keys_b))

    # new_sols = jax.tree_multimap(lambda x, y, a, b:
    #                             x + a * iso_emitter.iso_sigma +
    #                             b * iso_emitter.line_sigma * (x - y),
    #                                 sols_1, sols_2, noise_a, noise_b)
```

```
        # return iso_emitter, new_sols

        key_a, key_b = jax.random.split(key_variation, 2)
        iso_gaussian = jax.random.normal(key_a,
                            shape=(batch_size, solution_dim),
                            dtype=D_TYPE) * iso_emitter.iso_sigma

        # expanded last dimension used for multiplication later
        line_gaussian = jax.random.normal(key_b,
                            shape=(batch_size, 1),
                            dtype=D_TYPE) * iso_emitter.line_sigma

        directions = (sols_1 - sols_2).astype(D_TYPE)

        new_sols = sols_2 + iso_gaussian + jnp.multiply(
                        jnp.array(line_gaussian), directions)

        return iso_emitter, iso_emitter._ask_clip(new_sols,
                        iso_emitter.lower_bounds,
                        iso_emitter.upper_bounds)

    @staticmethod
    def tell(static_settings, iso_emitter, solutions, objective_values,
                behavior_values, dead, repertoire, key):
        """Inserts entries into the archive.
        """
        repertoire = repertoire.add_to_archive(repertoire = repertoire,
                                    pop_p = solutions,
                                    bds = behavior_values,
                                    eval_scores = objective_values,
                                    dead = dead)
        return iso_emitter, repertoire
```

Code Snippet A.10: Implementation of the MAP-Elites (Iso + LineDD) in JAX (_iso_line_emitter.py)

A.3.3 Covariance Matrix Adaptation MAP-Elites (CMA-ME) - Improvement

```
"""Provides the ImprovementEmitter.
Adapted from https://github.com/icaros-usc/dqd/blob/main/ribs/emitters/_improvement_emitter.py
"""

from typing import Any
import jax.numpy as jnp
import flax
import jax
from jax import lax
import functools
```

```
from qd_utils.grid_archive import Repertoire
from training.emitters.emitters_utils import EmitterBase
from training.opt._cma_es import CMAEvolutionStrategy, WeightRules
from collections import namedtuple

Array = jnp.ndarray

class SelectionRules:
    MU = 1
    FILTER = 2

class RestartRules:
    BASIC = 1
    NO_IMPROVEMENT = 2

D_TYPE = jnp.float32

@flax.struct.dataclass
class ImprovementEmitter:
    """Adapts a covariance matrix towards changes in the archive.

    This emitter originates in `Fontaine 2020
    <https://arxiv.org/abs/1912.02400>`_. Initially, it starts at ``x0`` and
    uses CMA-ES to search for solutions that improve the archive, i.e. solutions
    that add new entries to the archive or improve existing entries. Once CMA-ES
    restarts (see ``restart_rule``), the emitter starts from a randomly chosen
    elite in the archive and continues searching for solutions that improve the
    archive.
    """

    x0: Array
    sigma0: D_TYPE
    lower_bounds: Array
    upper_bounds: Array
    batch_size: jnp.int32
    opt: CMAEvolutionStrategy
    solution_dim: jnp.int32
    num_parents: jnp.int32
    restarts: jnp.int32
    restart_rule: jnp.int32
    selection_rule: jnp.int32


    @classmethod
    def create(cls,
               x0,
               sigma0,
                 selection_rule=SelectionRules.FILTER,
                 restart_rule=RestartRules.NO_IMPROVEMENT,
                 weight_rule=WeightRules.TRUNCATION,
                 bounds=None,
                 batch_size=None):
        solution_dim = len(x0)
        x0 = jnp.array(x0, dtype=D_TYPE)
        sigma0 = sigma0
        lower_bounds, upper_bounds = EmitterBase.process_bounds(bounds, solution_dim)
        batch_size = batch_size

        if selection_rule not in [SelectionRules.MU, SelectionRules.FILTER]:
            raise ValueError(f"Invalid selection_rule {selection_rule}")

        if restart_rule not in [RestartRules.BASIC, RestartRules.NO_IMPROVEMENT]:
            raise ValueError(f"Invalid restart_rule {restart_rule}")

        opt, static_settings_opt = CMAEvolutionStrategy.create(sigma0,
                        batch_size, solution_dim, weight_rule)
        opt = opt.reset(static_settings_opt, opt, x0)
        get_num_of_parents_fn = jax.jit(functools.partial(cls._get_num_of_parents,
                            selection_rule))
        num_parents = (opt.batch_size // 2
                  if selection_rule == SelectionRules.MU else None)
        batch_size = opt.batch_size

        static_settings = dict()
        static_settings['get_num_of_parents_fn'] = get_num_of_parents_fn
        static_settings['solution_dim'] = solution_dim
        static_settings['batch_size'] = batch_size
        static_settings['restart_rule'] = restart_rule
        static_settings['opt_settings'] = static_settings_opt
        StaticSettings = namedtuple('StaticSettings', static_settings)

        restarts = 0
        return (cls(x0, sigma0, lower_bounds, upper_bounds ,
                  batch_size, opt,solution_dim, num_parents,
                  restarts, restart_rule, selection_rule),
                  StaticSettings(**static_settings))

    @staticmethod
    def _get_batch_size(static_settings):
        return static_settings.batch_size

    @staticmethod
    def _get_restart_rule(static_settings):
        return static_settings.restart_rule
```

```
@staticmethod
def _get_solution_dim(static_settings):
    return static_settings.solution_dim

@staticmethod
def _get_num_of_parents_fn(static_settings):
    return static_settings.get_num_of_parents_fn

@staticmethod
def _get_optimiser_settings(static_settings):
    return static_settings.opt_settings

@staticmethod
def _get_num_of_parents(selection_rule, new_sols, num_parents):
    return (new_sols if selection_rule == SelectionRules.FILTER
            else num_parents)

@staticmethod
def ask(static_settings, imp_emitter, repertoire, key):
    """Samples new solutions from a multivariate Gaussian.

    The multivariate Gaussian is parameterized by the CMA-ES optimizer.
    """
    opt, solutions = imp_emitter.opt.ask(imp_emitter._get_optimiser_settings(
                    static_settings),
                    imp_emitter.opt, imp_emitter.lower_bounds,
                    imp_emitter.upper_bounds, key)

    return imp_emitter.replace(opt = opt), solutions

@staticmethod
def _check_restart(restart_rule, num_parents):
    """Emitter-side checks for restarting the optimizer.

    The optimizer also has its own checks.
    """
    return restart_rule == RestartRules.NO_IMPROVEMENT and num_parents == 0

@staticmethod
def _reset_opt(static_settings, imp_emitter, repertoire, elite_key):
    new_x0 = Repertoire.get_random_elite(repertoire, elite_key)
    opt = imp_emitter.opt.reset(
                imp_emitter._get_optimiser_settings(static_settings),
                imp_emitter.opt, new_x0)
    restarts = imp_emitter.restarts + 1
    return imp_emitter.replace(opt = opt, restarts =
                    restarts), repertoire

    @staticmethod
    def tell(static_settings, imp_emitter, solutions, objective_values,
                        behavior_values, dead, repertoire, key):
        """Gives the emitter results from evaluating solutions.

        As solutions are inserted into the archive, we record their "improvement
        value" -- conveniently, this is the ``value`` returned by
        :meth:`ribs.archives.ArchiveBase.add`. We then rank the solutions
        according to their add status (new solutions rank in front of
        solutions that improved existing entries in the archive, which rank
        ahead of solutions that were not added), followed by their improvement
        value.  We then pass the ranked solutions to the underlying CMA-ES
        optimizer to update the search parameters.
        """
        get_num_of_parents_fn = ImprovementEmitter._get_num_of_parents_fn(static_settings)
        solution_dim = ImprovementEmitter._get_solution_dim(static_settings)
        restart_rule = ImprovementEmitter._get_restart_rule(static_settings)
        # new_sols = repertoire.num_indivs
        repertoire, sols_entries, new_sols = repertoire.add_to_archive_extended(
                                        repertoire = repertoire,
                                        pop_p = solutions,
                                        bds = behavior_values,
                                        eval_scores = objective_values,
                                        dead = dead)
        # Find the indices of the sorted array on status codes and then objective/evaluation
scores
        r_indices = jnp.lexsort((sols_entries[:,2], sols_entries[:,1], sols_entries[:,0]))
        ranked_sols_entries = jnp.flip(sols_entries[r_indices], axis=0)
        # new_sols = repertoire.num_indivs - new_sols
        # we want the descending order of the rankings and not the ascending
        indices = ranked_sols_entries[:,2].astype(jnp.int32)
        num_parents = get_num_of_parents_fn(new_sols, imp_emitter.num_parents)

        opt = CMAEvolutionStrategy.tell(imp_emitter._get_optimiser_settings(static_settings),
                                            imp_emitter.opt, solutions[indices], num_parents)

        key, elite_key = jax.random.split(key, 2)

        imp_emitter = imp_emitter.replace(opt = opt)

        should_reset_opt = jnp.logical_or(
                                    CMAEvolutionStrategy.check_stop(opt,
                                            ranked_sols_entries[:,1], num_parents),
                                    ImprovementEmitter._check_restart(restart_rule,
                                            new_sols)
                                    )

        return lax.cond(should_reset_opt,
                lambda x: ImprovementEmitter._reset_opt(static_settings,
                            imp_emitter, repertoire, elite_key),
                lambda x: (imp_emitter, repertoire),
                None
            )
```

Code Snippet A.11: Implementation of CMA-ME Improvement the in JAX (_improvement_emitter.py)

A.3.4 Covariance Matrix Adaptation MAP-Elites (CMA-ME) - Optimizing

```
"""Provides the OptimizingEmitter.
Adapted from https://github.com/icaros-usc/dqd/blob/main/ribs/emitters/_optimizing_emitter.py
"""
from typing import Any
import jax.numpy as jnp
import flax
import jax
from jax import lax
from training.emitters.emitters_utils import EmitterBase
from training.opt._cma_es import CMAEvolutionStrategy, WeightRules
from collections import namedtuple

Array = jnp.ndarray

class SelectionRules:
    MU = 1
    FILTER = 2

class RestartRules:
    BASIC = 1
    NO_IMPROVEMENT = 2

D_TYPE = jnp.float32

@flax.struct.dataclass
class OptimizingEmitter:
    """Adapts a covariance matrix towards the objective.

    This emitter originates in `Fontaine 2020
    <https://arxiv.org/abs/1912.02400>`_. Initially, it starts at ``x0`` and
    uses CMA-ES to optimize for objective values. After CMA-ES converges, the
    emitter restarts the optimizer. It picks a random elite in the archive and
    begins optimizing from there.
    """
    x0: Array
    sigma0: D_TYPE
    lower_bounds: Array
    upper_bounds: Array
    batch_size: jnp.int32
    opt: CMAEvolutionStrategy
    solution_dim: jnp.int32
    num_parents: jnp.int32
    restarts: jnp.int32
    restart_rule: jnp.int32
    selection_rule: jnp.int32

    @classmethod
    def create(cls,
               x0,
               sigma0,
               selection_rule=SelectionRules.FILTER,
               restart_rule=RestartRules.NO_IMPROVEMENT,
               weight_rule=WeightRules.TRUNCATION,
               bounds=None,
               batch_size=None):
        solution_dim = len(x0)
        x0 = jnp.array(x0, dtype=D_TYPE)
        sigma0 = sigma0
        lower_bounds, upper_bounds = EmitterBase.process_bounds(bounds, solution_dim)
        batch_size = batch_size

        if selection_rule not in [SelectionRules.MU, SelectionRules.FILTER]:
            raise ValueError(f"Invalid selection_rule {selection_rule}")

        if restart_rule not in [RestartRules.BASIC, RestartRules.NO_IMPROVEMENT]:
            raise ValueError(f"Invalid restart_rule {restart_rule}")

        opt, static_settings_opt = CMAEvolutionStrategy.create(sigma0,
                        batch_size, solution_dim, weight_rule)
        opt = opt.reset(static_settings_opt, opt, x0)
        # get_num_of_parents_fn = jax.jit(functools.partial(cls._get_num_of_parents,
selection_rule))
        num_parents = (opt.batch_size // 2 if selection_rule ==
                        SelectionRules.MU else None)
        batch_size = opt.batch_size

        static_settings = dict()
        # static_settings['get_num_of_parents_fn'] = get_num_of_parents_fn
        static_settings['solution_dim'] = solution_dim
        static_settings['batch_size'] = batch_size
        static_settings['restart_rule'] = restart_rule
        static_settings['selection_rule'] = selection_rule
        static_settings['opt_settings'] = static_settings_opt
        StaticSettings = namedtuple('StaticSettings', static_settings)

        restarts = 0
        return (cls(x0, sigma0, lower_bounds, upper_bounds ,
                    batch_size, opt,
                    solution_dim, num_parents, restarts,
                    restart_rule, selection_rule),
                    StaticSettings(**static_settings))

    @staticmethod
    def _get_batch_size(static_settings):
        return static_settings.batch_size

    @staticmethod
    def _get_restart_rule(static_settings):
        return static_settings.restart_rule

    @staticmethod
    def _get_solution_dim(static_settings):
        return static_settings.solution_dim

    @staticmethod
    def _get_selection_rule(static_settings):
        return static_settings.selection_rule

    @staticmethod
    def _get_optimiser_settings(static_settings):
        return static_settings.opt_settings

    @staticmethod
    def ask(static_settings, otEmitter, repertoire, key):
        """Samples new solutions from a multivariate Gaussian.

        The multivariate Gaussian is parameterized by the CMA-ES optimizer.
        """
        opt, solutions = otEmitter.opt.ask(otEmitter._get_optimiser_settings(
                        static_settings),
                        otEmitter.opt, otEmitter.lower_bounds,
                        otEmitter.upper_bounds, key)
        return otEmitter.replace(opt = opt), solutions

    @staticmethod
    def _check_restart(restart_rule, num_parents):
        """Emitter-side checks for restarting the optimizer.
        The optimizer also has its own checks.
        """
        return restart_rule == RestartRules.NO_IMPROVEMENT and num_parents == 0

    @staticmethod
    def _reset_opt(static_settings, optEmitter, repertoire, elite_key):
        new_x0 = repertoire.get_random_elite(repertoire, elite_key)
        opt = optEmitter.opt.reset(optEmitter._get_optimiser_settings(static_settings),
                                optEmitter.opt, new_x0)
        restarts = optEmitter.restarts + 1
        return optEmitter.replace(opt = opt, restarts = restarts), repertoire

    @staticmethod
    def tell(static_settings, optEmitter, solutions, objective_values,
                behavior_values, dead, repertoire, key):
        """Gives the emitter results from evaluating solutions.

        As solutions are inserted into the archive, we record their "improvement
        value" -- conveniently, this is the ``value`` returned by
        :meth:`ribs.archives.ArchiveBase.add`. We then rank the solutions
        according to their add status (new solutions rank in front of
        solutions that improved existing entries in the archive, which rank
        ahead of solutions that were not added), followed by their improvement
        value.  We then pass the ranked solutions to the underlying CMA-ES
        optimizer to update the search parameters.
        """
        solution_dim = optEmitter._get_solution_dim(static_settings)
        restart_rule = optEmitter._get_restart_rule(static_settings)
        selection_rule = optEmitter._get_selection_rule(static_settings)

        repertoire, sols_entries, new_sols = repertoire.add_to_archive_extended(
                                        repertoire = repertoire,
                                        pop_p = solutions,
                                        bds = behavior_values,
                                        eval_scores = objective_values,
                                        dead = dead)

        if selection_rule == SelectionRules.FILTER:
            # Sort by whether the solution was added into the archive, followed
            # by objective value.
            sort_elements = (objective_values[sols_entries[:,2].astype(jnp.int32)],
                                    sols_entries[:,0])
        elif selection_rule == SelectionRules.MU:
            # Sort only by objective value.
            sort_elements = (objective_values[sols_entries[:,2].astype(jnp.int32)])
        # Find the indices of the sorted array on status codes and then objective/evaluation
scores
        r_indices = jnp.lexsort(sort_elements)
        ranked_sols_entries = jnp.flip(sols_entries[r_indices], axis=0)
        # we want the descending order of the rankings and not the ascending
        indices = ranked_sols_entries[:,2].astype(jnp.int32)

        num_parents = (new_sols if selection_rule == SelectionRules.FILTER
                        else optEmitter.num_parents)

        opt = optEmitter.opt.tell(optEmitter._get_optimiser_settings(static_settings),
                        optEmitter.opt, solutions[indices], num_parents)

        key, elite_key = jax.random.split(key, 2)

        optEmitter = optEmitter.replace(opt = opt)

        should_reset_opt = jnp.logical_or(
                            CMAEvolutionStrategy.check_stop(opt,
                            ranked_sols_entries[:,1], num_parents),
                            optEmitter._check_restart(restart_rule, new_sols)
                        )

        return lax.cond(should_reset_opt,
                lambda x: optEmitter._reset_opt(static_settings, optEmitter,
                        repertoire, elite_key),
                lambda x: (optEmitter, repertoire),
                None
            )
```

Code Snippet A.12: Implementation of the CMA-ME - Optimizing in JAX (_optimizing_emitter.py)

A.3.5 Covariance Matrix Adaptation MAP-Elites (CMA-ME) – Random Direction

```
"""Provides the RandomDirectionEmitter.
Adapted from https://github.com/icaros-
usc/dqd/blob/main/ribs/emitters/_random_direction_emitter.py
"""
from typing import Any
import jax.numpy as jnp
import flax
import jax
from jax import lax
from qd_utils.grid_archive import Repertoire
from training.emitters.emitters_utils import EmitterBase
from training.opt._cma_es import CMAEvolutionStrategy, WeightRules
from collections import namedtuple
```

```
Array = jnp.ndarray

class SelectionRules:
    MU = 1
    FILTER = 2

class RestartRules:
    BASIC = 1
    NO_IMPROVEMENT = 2

D_TYPE = jnp.float32

@flax.struct.dataclass
class RandomDirectionEmitter(EmitterBase):
    """Performs a random walk in behavior space by pursuing randomly chosen
    behavior space directions.

    This emitter originates in `Fontaine 2020
    <https://arxiv.org/abs/1912.02400>`_. Initially, it starts at ``x0`` and
    uses CMA-ES to search for solutions along a randomly chosen direction. Once
    CMA-ES restarts (see ``restart_rule``), the emitter starts from a randomly
    chosen elite in the archive and pursues a new random direction.
    """

    x0: Array
    sigma0: D_TYPE
    lower_bounds: Array
    upper_bounds: Array
    batch_size: jnp.int32
    opt: CMAEvolutionStrategy
    solution_dim: jnp.int32
    num_parents: jnp.int32
    restarts: jnp.int32
    restart_rule: jnp.int32
    selection_rule: jnp.int32
    archive_bounds: Array
    target_behavior_dir:Array

    @classmethod
    def create(cls,
                 x0,
                 sigma0,
                 archive_bounds,
                 key,
                 selection_rule=SelectionRules.FILTER,
                 restart_rule=RestartRules.NO_IMPROVEMENT,
                weight_rule=WeightRules.TRUNCATION,
                bounds=None,
                batch_size=None):

        solution_dim = len(x0)
        x0 = jnp.array(x0, dtype=D_TYPE)
        sigma0 = sigma0
        lower_bounds, upper_bounds = EmitterBase.process_bounds(
                    bounds, solution_dim)
        batch_size = batch_size

        if selection_rule not in [SelectionRules.MU, SelectionRules.FILTER]:
            raise ValueError(f"Invalid selection_rule {selection_rule}")

        if restart_rule not in [RestartRules.BASIC, RestartRules.NO_IMPROVEMENT]:
            raise ValueError(f"Invalid restart_rule {restart_rule}")

        opt, static_settings_opt = CMAEvolutionStrategy.create(sigma0,
                        batch_size, solution_dim, weight_rule)
        opt = opt.reset(static_settings_opt, opt, x0)

        num_parents = (opt.batch_size // 2 if selection_rule ==
                    SelectionRules.MU else None)
        batch_size = opt.batch_size

        static_settings = dict()
        static_settings['solution_dim'] = solution_dim
        static_settings['solution_dim'] = solution_dim
        static_settings['batch_size'] = batch_size
        static_settings['restart_rule'] = restart_rule
        static_settings['selection_rule'] = selection_rule
        static_settings['archive_bounds'] = archive_bounds
        static_settings['opt_settings'] = static_settings_opt
        StaticSettings = namedtuple('StaticSettings', static_settings)

        target_behavior_dir = cls._generate_random_direction(
                        static_settings, archive_bounds, key)

        restarts = 0
        return cls(x0, sigma0, lower_bounds, upper_bounds ,
                    batch_size, opt, solution_dim,
                    num_parents, restarts, restart_rule,
                    selection_rule, archive_bounds,
                    target_behavior_dir), StaticSettings(**static_settings)

    @staticmethod
    def _get_batch_size(static_settings):
        return static_settings.batch_size

    @staticmethod
    def _get_restart_rule(static_settings):
        return static_settings.restart_rule

    @staticmethod
    def _get_solution_dim(static_settings):
        return static_settings.solution_dim

    @staticmethod
    def _get_selection_rule(static_settings):
        return static_settings.selection_rule

    @staticmethod
    def _get_archive_bounds(static_settings):
        return static_settings.archive_bounds

    @staticmethod
    def _get_optimiser_settings(static_settings):
        return static_settings.opt_settings


    @staticmethod
    def ask(static_settings, randomDirEmitter, repertoire, key):
        """Samples new solutions from a multivariate Gaussian.

        The multivariate Gaussian is parameterized by the CMA-ES optimizer.

        Returns:
            ``(batch_size, solution_dim)`` array -- contains ``batch_size`` new
            solutions to evaluate.
        """
        opt, solutions = randomDirEmitter.opt.ask(
                randomDirEmitter._get_optimiser_settings(static_settings),
                randomDirEmitter.opt, randomDirEmitter.lower_bounds,
                randomDirEmitter.upper_bounds, key)
        return randomDirEmitter.replace(opt = opt), solutions

    @staticmethod
    def _generate_random_direction(static_settings, archive_bounds, key):
        """Generates a new random direction in the behavior space.

        The direction is sampled from a standard Gaussian -- since the standard
        Gaussian is isotropic, there is equal probability for any direction. The
        direction is then scaled to the behavior space bounds.
        """
        ranges = archive_bounds[:,1] - archive_bounds[:,0]
        behavior_dim = archive_bounds.shape[0]
        unscaled_dir = jax.random.normal(key,
                                shape=(behavior_dim,),
                                dtype=D_TYPE)
        return unscaled_dir * ranges

    @staticmethod
    def _check_restart(restart_rule, num_parents):
        """Emitter-side checks for restarting the optimizer.

        The optimizer also has its own checks.
        """
        return restart_rule == RestartRules.NO_IMPROVEMENT and num_parents == 0

    @staticmethod
    def _reset_opt(static_settings, randomDirEmitter, repertoire, key):
        dir_key, elite_key = jax.random.split(key, 2)
        new_x0 = repertoire.get_random_elite(repertoire, elite_key)
        opt = randomDirEmitter.opt.reset(
            randomDirEmitter._get_optimiser_settings(static_settings),
            randomDirEmitter.opt, new_x0)
        archive_bounds = randomDirEmitter._get_archive_bounds(static_settings)
        target_behavior_dir = randomDirEmitter._generate_random_direction(
                    static_settings,
                    archive_bounds, dir_key)
        restarts = randomDirEmitter.restarts + 1
        return randomDirEmitter.replace(opt = opt, restarts = restarts,
                        target_behavior_dir = target_behavior_dir), repertoire

    @staticmethod
    def tell(static_settings, randomDirEmitter, solutions, objective_values,
                        behavior_values, dead, repertoire, key):
        """Gives the emitter results from evaluating solutions.

        As solutions are inserted into the archive, we record their "improvement
        value" -- conveniently, this is the ``value`` returned by
        :meth:`ribs.archives.ArchiveBase.add`. We then rank the solutions
        according to their add status (new solutions rank in front of
        solutions that improved existing entries in the archive, which rank
        ahead of solutions that were not added), followed by their improvement
        value.  We then pass the ranked solutions to the underlying CMA-ES
        optimizer to update the search parameters.
        """
        solution_dim = randomDirEmitter._get_solution_dim(static_settings)
        restart_rule = randomDirEmitter._get_restart_rule(static_settings)
        selection_rule = randomDirEmitter._get_selection_rule(static_settings)
        repertoire, sols_entries, new_sols = repertoire.add_to_archive_extended(
                                        repertoire = repertoire,
                                        pop_p = solutions,
                                        bds = behavior_values,
                                        eval_scores = objective_values,
                                        dead = dead)
        projection = jnp.dot(behavior_values, randomDirEmitter.target_behavior_dir)
        # Rearrange based on their initial indices to make it parallel to sols_entries
        projection = projection[sols_entries[:,2].astype(jnp.int32)]
        if selection_rule == SelectionRules.FILTER:
            # Sort by whether the solution was added into the archive, followed
            # by projection.
            sort_elements = (sols_entries[:,2], projection, sols_entries[:,0])
        elif selection_rule == SelectionRules.MU:
            # Sort only by projection.
            sort_elements = (projection)

        # Find the indices of the sorted array on status codes and then objective/evaluation
scores
        r_indices = jnp.lexsort(sort_elements)
        ranked_sols_entries = jnp.flip(sols_entries[r_indices], axis=0)
        # we want the descending order of the rankings and not the ascending
        indices = ranked_sols_entries[:,2].astype(jnp.int32)

        num_parents = (new_sols if selection_rule == SelectionRules.FILTER
                            else randomDirEmitter.num_parents)

        opt = randomDirEmitter.opt.tell(randomDirEmitter._get_optimiser_settings(
                            static_settings),
                            randomDirEmitter.opt,
                            solutions[indices], num_parents)

        key, reset_key = jax.random.split(key, 2)

        randomDirEmitter = randomDirEmitter.replace(opt = opt)

        should_reset_opt = jnp.logical_or(
                                CMAEvolutionStrategy.check_stop(opt,
                                        projection, num_parents),
                                randomDirEmitter._check_restart(restart_rule,
                                        new_sols)
                            )

        return lax.cond(should_reset_opt,
                lambda x: randomDirEmitter._reset_opt(static_settings, randomDirEmitter,
                                                    repertoire, reset_key),
                lambda x: (randomDirEmitter, repertoire),
                None
            )
```

Code Snippet A.13: Implementation of the CMA-ME Random Direction in JAX (_random_direction_emitter.py)

A.4 DQD Emitters (Array Version)

A.4.1 Gradient Emitter (OMG-MEGA iso & line, OG-MAP-Elites iso & line)

```
"""Provides the GradientImprovementEmitter.
Adapted from: https://github.com/icaros-usc/dqd/blob/main/ribs/emitters/_gradient_emitter.py
"""
import jax.numpy as jnp
import flax
import jax
from collections import namedtuple
from training.emitters.emitters_utils import EmitterBase

Array = jnp.ndarray
D_TYPE = jnp.float32

class OperatorTypes:
    ISOTROPIC = 1
    ISO_LINE_DD = 2

@flax.struct.dataclass
class GradientEmitter:
    """Generates new solutions based on the gradient of the objective and measures.
    """

    x0: Array
    sigma0: D_TYPE
    sigma_g: D_TYPE
    line_sigma: D_TYPE
    num_coefficients: jnp.int32
    measure_gradients: jnp.int32
    normalize_gradients: jnp.int32
    operator_type: jnp.int32
    batch_size: jnp.int32
    solution_dim: jnp.int32
    lower_bounds: Array
    upper_bounds: Array
    jacobian: Array
    parents: Array

    @classmethod
    def create(cls,
                 x0,
                 sigma0,
                 sigma_g,
                 line_sigma,
                 behavior_dim,
                 measure_gradients=1,    # 0 --> false 1 --> true
                 normalize_gradients=1,  # 0 --> false 1 --> true
                 operator_type = OperatorTypes.ISOTROPIC,
                 bounds=None,
                 batch_size=None):

        # static normalize_gradients solution_dim batch_size selection_rule restart_rule
        # self._rng = np.random.default_rng(key)
        x0 = jnp.array(x0, dtype=D_TYPE)
        sigma0 = float(sigma0)
        sigma_g = float(sigma_g)
        line_sigma = float(line_sigma)
        solution_dim = len(x0)
        # What's the difference with the manual creation of bounds in ask?
        lower_bounds, upper_bounds = EmitterBase.process_bounds(bounds, solution_dim)
        num_coefficients = behavior_dim + 1
        static_settings = dict()
        static_settings['solution_dim'] = solution_dim
        static_settings['batch_size'] = batch_size
        static_settings['measure_gradients'] = measure_gradients
        static_settings['normalize_gradients'] = normalize_gradients
        static_settings['operator_type'] = operator_type
        static_settings['num_coefficients'] = num_coefficients
        StaticSettings = namedtuple('StaticSettings', static_settings)

        jacobian = jnp.zeros((batch_size, num_coefficients, solution_dim))
        parents = jnp.zeros((batch_size, solution_dim))
        return (cls(x0, sigma0, sigma_g, line_sigma, num_coefficients,
                    measure_gradients,
                    normalize_gradients, operator_type,
                    batch_size, solution_dim,
                    lower_bounds, upper_bounds, jacobian,
                    parents),
                     StaticSettings(**static_settings))

    @staticmethod
    def _get_batch_size(static_settings):
        return static_settings.batch_size

    @staticmethod
    def _get_solution_dim(static_settings):
        return static_settings.solution_dim

    @staticmethod
    def _get_measure_gradients(static_settings):
        return static_settings.measure_gradients

    @staticmethod
    def _get_normalize_gradients(static_settings):
        return static_settings.normalize_gradients

    @staticmethod
    def _get_operator_type(static_settings):
        return static_settings.operator_type

    @staticmethod
    def _get_num_coefficients(static_settings):
        return static_settings.num_coefficients

    @staticmethod
    def _ask_clip(parents, lower_bounds, upper_bounds):
        return jnp.minimum(jnp.maximum(parents, lower_bounds),
                upper_bounds)

    @staticmethod
    def _ask_as_gaussian_emitter(static_settings, grad_emitter, repertoire, key):
        """Creates solutions by adding Gaussian noise to elites in the archive.
        """
        batch_size = grad_emitter._get_batch_size(static_settings)
        solution_dim = grad_emitter._get_solution_dim(static_settings)
        key_selection, key_variation = jax.random.split(key, 2)

        # SELECTION #
        idx_p1 = jax.random.randint(key_selection, shape=(batch_size,),
                    minval=0, maxval=repertoire.num_indivs)
        tot_indivs = repertoire.fitness.ravel().shape[0]
        indexes = jnp.argwhere(jnp.logical_not(jnp.isnan(repertoire.fitness)),
                    size = tot_indivs)
        indexes = jnp.transpose(indexes, axes=(1, 0))
        indiv_indices = jnp.array(jnp.ravel_multi_index(indexes,
                    repertoire.fitness.shape, mode='clip')).astype(int)

        idx_p1 = indiv_indices.at[idx_p1].get()
        sols = jax.tree_map(lambda x: x.at[idx_p1].get(),repertoire.archive)

        # # VARIATION - MUTATION #
        # # Better approach since it operates directly on the tree
        # # structure of the solutions
        # num_vars = len(jax.tree_leaves(sols))
        # treedef = jax.tree_structure(sols)
        # all_keys = jax.random.split(key_variation, num=num_vars)

        # # Gaussian noise
        # noise = jax.tree_multimap(
        #     lambda g, k: jax.random.normal(k, shape=g.shape, dtype=g.dtype), sols,
        #     jax.tree_unflatten(treedef, all_keys))

        # # Added noise in positive direction
        # mutated_sols = jax.tree_multimap(lambda g, n: g + n * grad_emitter.sigma0, sols,
noise)
        # # Added noise in negative direction
        # anit_mutated_sols = jax.tree_multimap(lambda g, n: g - n * grad_emitter.sigma0,
sols, noise)

        # return grad_emitter, mutated_sols

        noise = jax.random.normal(key_variation, shape=(batch_size,
                        solution_dim), dtype=D_TYPE) * grad_emitter.sigma0

        return grad_emitter, grad_emitter._ask_clip(sols + noise,
                                     grad_emitter.lower_bounds,
                                     grad_emitter.upper_bounds)

    @staticmethod
    def _ask_as_iso_line_emitter(static_settings, grad_emitter, repertoire, key):
        """Generates ``batch_size`` solutions.
        """
        batch_size = grad_emitter._get_batch_size(static_settings)
        solution_dim = grad_emitter._get_solution_dim(static_settings)
        key_selection, key_variation = jax.random.split(key, 2)

        # SELECTION #
        key_select_p1, key_select_p2 = jax.random.split(key_selection, 2)
        idx_s1 = jax.random.randint(key_select_p1, shape=(batch_size,),
                    minval=0, maxval=repertoire.num_indivs)
        idx_s2 = jax.random.randint(key_select_p2, shape=(batch_size,),
                    minval=0, maxval=repertoire.num_indivs)
        tot_indivs = repertoire.fitness.ravel().shape[0]
        indices = jnp.argwhere(jnp.logical_not(jnp.isnan(repertoire.fitness)),
                    size = tot_indivs)
        indices = jnp.transpose(indices, axes=(1, 0))
        indiv_indices = jnp.array(jnp.ravel_multi_index(indices,
                    repertoire.fitness.shape, mode='clip')).astype(int)

        idx_s1 = indiv_indices.at[idx_s1].get()
        idx_s2 = indiv_indices.at[idx_s2].get()
        sols_1 = jax.tree_map(lambda x: x.at[idx_s1].get(),repertoire.archive)
        sols_2 = jax.tree_map(lambda x: x.at[idx_s2].get(),repertoire.archive)

        # # VARIATION #
        # # Better approach since it operates directly on the tree
        # # structure of the solutions
        # num_vars = len(jax.tree_leaves(sols_1))
        # treedef = jax.tree_structure(sols_1)
        # key_a, key_b = jax.random.split(key_variation, 2)
        # all_keys_a = jax.random.split(key_a, num_vars)
        # all_keys_b = jax.random.split(key_b, num_vars)

        # noise_a = jax.tree_multimap(
        #     lambda g, k: jax.random.normal(k, shape=g.shape, dtype=g.dtype), sols_1,
        #     jax.tree_unflatten(treedef, all_keys_a))
        # noise_b = jax.tree_multimap(
        #     lambda g, k: jax.random.normal(k, shape=g.shape, dtype=g.dtype), sols_2,
        #     jax.tree_unflatten(treedef, all_keys_b))

        # new_sols = jax.tree_multimap(lambda x, y, a, b:
        #                              x + a * grad_emitter.sigma0 +
        #                              b * grad_emitter.line_sigma * (x - y),
        #                                  sols_1, sols_2, noise_a, noise_b)

        # return grad_emitter, new_sols

        key_a, key_b = jax.random.split(key_variation, 2)
        iso_gaussian = jax.random.normal(key_a,
                            shape=(batch_size, solution_dim),
                            dtype=D_TYPE) * grad_emitter.sigma0

        # expanded last dimension used for multiplication later
        line_gaussian = jax.random.normal(key_b,
                            shape=(batch_size, 1),
                            dtype=D_TYPE) * grad_emitter.line_sigma

        directions = (sols_1 - sols_2).astype(D_TYPE)

        new_sols = sols_2 + iso_gaussian + jnp.multiply(jnp.array(line_gaussian), directions)

        return grad_emitter, grad_emitter._ask_clip(new_sols,
                        grad_emitter.lower_bounds, grad_emitter.upper_bounds)

    @staticmethod
    def ask_grad_estimate(static_settings, grad_emitter, repertoire, key):
        operator_type = grad_emitter._get_operator_type(static_settings)
        if operator_type == OperatorTypes.ISO_LINE_DD:
            grad_emitter,sols = grad_emitter._ask_as_iso_line_emitter(static_settings,
                            grad_emitter, repertoire, key)
        else:
            grad_emitter,sols = grad_emitter._ask_as_gaussian_emitter(static_settings,
                            grad_emitter, repertoire, key)

        return grad_emitter.replace(parents = sols), sols

    @staticmethod
    def ask(static_settings, grad_emitter, repertoire, key):
        num_coefficients = grad_emitter._get_num_coefficients(static_settings)
        batch_size = grad_emitter._get_batch_size(static_settings)
        measure_gradients = grad_emitter._get_measure_gradients(static_settings)
        if measure_gradients == 1:
            # Calculate gradient offsets
            noise = grad_emitter.sigma_g * jax.random.normal(key,
                        shape=(batch_size,num_coefficients))
            noise = noise.at[:, 0].set(jnp.abs(noise[:, 0]))
            noise = jnp.expand_dims(noise, axis=2)
            offsets = jnp.sum(jnp.multiply(grad_emitter.jacobian, noise), axis=1)
            # Calculate new solutions based on the gradient offsets
            new_sols = jnp.add(grad_emitter.parents, offsets)
        else:
            # isolate the gradients of objective values
            noise = grad_emitter.sigma_g * jax.random.normal(key,
                        shape=(batch_size,1))
            noise = jnp.abs(noise)
            jacobian = grad_emitter.jacobian[:,0,:]
            # jnp.squeeze(grad_emitter.jacobian[:,0:1,:], axis=1)
            offsets = jnp.multiply(jacobian, noise)
            grad_emitter = grad_emitter.replace(jacobian = jacobian)
            # Calculate new solutions based on the gradient offsets of only
            new_sols = jnp.add(grad_emitter.parents, offsets)

        return grad_emitter, new_sols

    @staticmethod
    def _normalize_gradients(jacobian):
        """
        Normalises the gradients of the jacobian matrix.

        More info can be found at the appendix E of the paper "Fontaine,
        M. C., & Nikolaidis, S. (2021). Differentiable Quality
        Diversity. arXiv [cs.AI]" for an in-depth explanation
        """
```

```
        norms = jnp.linalg.norm(jacobian, axis=2)
        norms += 1e-8 # Make this configurable later
        norms = jnp.expand_dims(norms, axis=2)
        jacobian /= norms
        return jacobian

    @staticmethod
    def tell_jacobian(static_settings, grad_emitter, solutions, objective_values,
                behavior_values, dead, repertoire, key, jacobian):
        normalize_gradients = grad_emitter._get_normalize_gradients(static_settings)
        if normalize_gradients > 0:
            jacobian = grad_emitter._normalize_gradients(jacobian)

        grad_emitter = grad_emitter.replace(jacobian = jacobian)
        return grad_emitter.tell(static_settings, grad_emitter, solutions,
                objective_values, behavior_values, dead, repertoire, key)

    @staticmethod
    def tell(static_settings, iso_emitter, solutions, objective_values,
                behavior_values, dead, repertoire, key):
        """Inserts entries into the archive.
        """
        repertoire = repertoire.add_to_archive(repertoire = repertoire,
                                    pop_p = solutions,
                                    bds = behavior_values,
                                    eval_scores = objective_values,
                                    dead = dead)
        return iso_emitter, repertoire
```

Code Snippet A.14: Implementation of the Gradient Emitter in JAX (_gradient_emitter.py)

A.4.2 Gradient Improvement Emitter (CMA-MEGA & CMA-MEGA with Adam)

```
"""Provides the GradientImprovementEmitter.
Adapted from: https://github.com/icaros-
usc/dqd/blob/main/ribs/emitters/_gradient_improvement_emitter.py
"""
from typing import Any
import jax.numpy as jnp
import flax
import jax
from jax import lax

from qdax.training.emitters.emitters_utils import EmitterBase
from qdax.training.opt._cma_es import CMAEvolutionStrategy, WeightRules
from qdax.training.opt._adam import AdamOpt
from qdax.training.opt._gradient_ascent import  GradientAscentOpt
from collections import namedtuple
```

```python
Array = jnp.ndarray

class SelectionRules:
    MU = 1
    FILTER = 2

class RestartRules:
    BASIC = 1
    NO_IMPROVEMENT = 2

class GradientOptimizers:
    ADAM = 1
    GRADIENT_ASCENT = 2


D_TYPE = jnp.float32

@flax.struct.dataclass
class GradientImprovementEmitter():
    """Adapts a covariance matrix in behavior space towards changes in the archive.
    """

    batch_size: jnp.int32
    x0: Array
    behavior_dim: jnp.int32
    sigma_g: D_TYPE
    normalize_gradients: jnp.int32
    solution_dim: jnp.int32
    lower_bounds: Array
    upper_bounds: Array
    gradient_opt: Any
    selection_rule: jnp.int32
    restart_rule: jnp.int32
    num_coefficients: jnp.int32
    num_parents: jnp.int32
    opt: CMAEvolutionStrategy
    restarts: jnp.int32
    grad_coefficients: Array
    jacobian: Array

    @classmethod
    def create(cls,
               x0,
               behavior_dim,
               sigma_g,
               stepsize,
               selection_rule=SelectionRules.MU,
               restart_rule=RestartRules.NO_IMPROVEMENT,
               weight_rule=WeightRules.TRUNCATION,
               gradient_optimizer=GradientOptimizers.ADAM,
               normalize_gradients=1,  # 0 --> false 1 --> true
               bounds=None,
               batch_size=None):

      # static normalize_gradients solution_dim batch_size selection_rule restart_rule
      # self._rng = np.random.default_rng(key)
      x0 = jnp.array(x0, dtype=D_TYPE)
      solution_dim = len(x0)
      # What's the difference with the manual creation of bounds in ask?
      lower_bounds, upper_bounds = EmitterBase.process_bounds(bounds, solution_dim)

      gradient_opt = None

      if gradient_optimizer == GradientOptimizers.ADAM:
          gradient_opt, static_settings_obj_opt = AdamOpt.create(x0, stepsize, betas=(0.9,
0.999), epsilon=1e-8)
      elif gradient_optimizer == GradientOptimizers.GRADIENT_ASCENT:
          gradient_opt, static_settings_obj_opt = GradientAscentOpt.create(x0, stepsize,
epsilon=1e-8)
      else:
          raise ValueError(f"Invalid Gradient Ascent Optimizer {gradient_optimizer}")

      if selection_rule not in [SelectionRules.MU, SelectionRules.FILTER]:
          raise ValueError(f"Invalid selection_rule {selection_rule}")

      if restart_rule not in [RestartRules.BASIC, RestartRules.NO_IMPROVEMENT]:
          raise ValueError(f"Invalid restart_rule {restart_rule}")

      # key, elite_key = jax.random.split(key, 2)

      num_coefficients = behavior_dim + 1
      behavior_x0 = jnp.zeros(num_coefficients)

      opt, static_settings_bd_opt = CMAEvolutionStrategy.create(sigma_g, batch_size,
                                                               num_coefficients, weight_rule)
      opt = opt.reset(static_settings_bd_opt, opt, behavior_x0)
      # get_num_of_parents_fn = jax.jit(functools.partial(cls._get_num_of_parents,
selection_rule))
      num_parents = (opt.batch_size // 2 if selection_rule == SelectionRules.MU else None)
      batch_size = opt.batch_size

      static_settings = dict()
      # static_settings['get_num_of_parents_fn'] = get_num_of_parents_fn
      static_settings['num_coefficients'] = num_coefficients
      static_settings['batch_size'] = batch_size
      static_settings['restart_rule'] = restart_rule
      static_settings['selection_rule'] = selection_rule
      static_settings['behavior_dim'] = behavior_dim
      static_settings['normalize_gradients'] = normalize_gradients
      static_settings['bd_opt_settings'] = static_settings_bd_opt
      static_settings['obj_opt_settings'] = static_settings_obj_opt
      StaticSettings = namedtuple('StaticSettings', static_settings)

      restarts = 0
      grad_coefficients = jnp.zeros((batch_size, num_coefficients))
      jacobian = jnp.zeros((batch_size, num_coefficients, solution_dim))
      return (cls(batch_size, x0, behavior_dim, sigma_g, normalize_gradients, solution_dim,
                  lower_bounds, upper_bounds, gradient_opt, selection_rule, restart_rule,
                  num_coefficients, num_parents, opt, restarts, grad_coefficients,
jacobian),
                  StaticSettings(**static_settings))

  @staticmethod
  def _get_batch_size(static_settings):
      return static_settings.batch_size

  @staticmethod
  def _get_restart_rule(static_settings):
      return static_settings.restart_rule

  @staticmethod
  def _get_num_coefficients(static_settings):
      return static_settings.num_coefficients

  @staticmethod
  def _get_normalize_gradients(static_settings):
      return static_settings.normalize_gradients

  @staticmethod
  def _get_selection_rule(static_settings):
      return static_settings.selection_rule

  @staticmethod
  def _get_static_settings_bd_opt(static_settings):
      return static_settings.bd_opt_settings

  @staticmethod
  def _get_obj_optimiser_settings(static_settings):
      return static_settings.obj_opt_settings
  @staticmethod
  def ask_grad_estimate(static_settings, gradImprEmitter, repertoire, key):
      batch_size = gradImprEmitter._get_batch_size(static_settings)
      sols = jnp.repeat(jnp.array([gradImprEmitter.gradient_opt.theta]), batch_size, axis=0)
      return gradImprEmitter, sols

  @staticmethod
  def ask(static_settings, gradImprEmitter, repertoire, key):
      """Samples new solutions from a multivariate Gaussian.

      The multivariate Gaussian is parameterized by the CMA-ES optimizer.
      """
      num_coefficients = gradImprEmitter._get_num_coefficients(static_settings)
      lower_bounds = jnp.full(num_coefficients, -jnp.inf, dtype=D_TYPE)
      upper_bounds = jnp.full(num_coefficients, jnp.inf, dtype=D_TYPE)
      opt, noise =
gradImprEmitter.opt.ask(gradImprEmitter._get_static_settings_bd_opt(static_settings),
                        gradImprEmitter.opt, lower_bounds, upper_bounds, key)
      grad_coefficients = noise
      noise = jnp.expand_dims(noise, axis=2)
      offset = jnp.sum(jnp.multiply(gradImprEmitter.jacobian, noise), axis=1)
      sols = offset + gradImprEmitter.gradient_opt.theta
      return gradImprEmitter.replace(grad_coefficients = grad_coefficients, opt = opt), sols

  @staticmethod
  def _check_restart(restart_rule, num_parents):
      """Emitter-side checks for restarting the optimizer.

      The optimizer also has its own checks.
      """
      return restart_rule == RestartRules.NO_IMPROVEMENT and num_parents == 0

  @staticmethod
  def _normalize_gradients(jacobian):
      """
      Normalises the gradients of the jacobian matrix.

      More info can be found at the appendix E of the paper "Fontaine,
      M. C., & Nikolaidis, S. (2021). Differentiable Quality
      Diversity. arXiv [cs.AI]" for an in-depth explanation
      """
      norms = jnp.linalg.norm(jacobian, axis=2)
      norms += 1e-8 # Make this configurable later
      norms = jnp.expand_dims(norms, axis=2)
      jacobian /= norms
      return jacobian
  @staticmethod
  def _reset_opt(static_settings, gradImprEmitter, repertoire, elite_key):
      obj_optimiser_settings = gradImprEmitter._get_obj_optimiser_settings(static_settings)
      num_coefficients = gradImprEmitter._get_num_coefficients(static_settings)
      new_x0 = repertoire.get_random_elite(repertoire, elite_key)
      gradient_opt = gradImprEmitter.gradient_opt.reset(obj_optimiser_settings,
gradImprEmitter.gradient_opt, new_x0)
      behavior_x0 = jnp.zeros(num_coefficients)
      opt =
CMAEvolutionStrategy.reset(gradImprEmitter._get_static_settings_bd_opt(static_settings),
                       gradImprEmitter.opt, behavior_x0)
      restarts = gradImprEmitter.restarts + 1
      return gradImprEmitter.replace(gradient_opt = gradient_opt, opt = opt, restarts =
restarts), repertoire

  @staticmethod
  def tell_jacobian(static_settings, gradImprEmitter, solutions, objective_values,
                  behavior_values, dead, repertoire, key, jacobian):
      normalize_gradients = gradImprEmitter._get_normalize_gradients(static_settings)
      if normalize_gradients > 0:
          jacobian = gradImprEmitter._normalize_gradients(jacobian)

      gradImprEmitter = gradImprEmitter.replace(jacobian = jacobian)
      return gradImprEmitter.tell(static_settings, gradImprEmitter, solutions,
                  objective_values, behavior_values, dead, repertoire, key)

  @staticmethod
  def tell(static_settings, gradImprEmitter, solutions, objective_values, behavior_values,
dead, repertoire, key):
      """Gives the emitter results from evaluating solutions.

      As solutions are inserted into the archive, we record their "improvement
      value" -- conveniently, this is the ``value`` returned by
      :meth:`ribs.archives.ArchiveBase.add`. We then rank the solutions
      according to their add status (new solutions rank in front of
      solutions that improved existing entries in the archive, which rank
      ahead of solutions that were not added), followed by their improvement
      value.  We then pass the ranked solutions to the underlying CMA-ES
      optimizer to update the search parameters.
      """
      # static settings that need to be provided before the rest of the method is compiled
      normalize_gradients = gradImprEmitter._get_normalize_gradients(static_settings)
      batch_size = gradImprEmitter._get_batch_size(static_settings)
      restart_rule = gradImprEmitter._get_restart_rule(static_settings)
      selection_rule = gradImprEmitter._get_selection_rule(static_settings)
      obj_optimiser_settings = gradImprEmitter._get_obj_optimiser_settings(static_settings)
      repertoire, sols_entries, new_sols = repertoire.add_to_archive_extended(repertoire = 
repertoire,
                                        pop_p = solutions,
                                        bds = behavior_values,
                                        eval_scores = objective_values,
                                        dead = dead)

      # Find the indices of the sorted array on status codes and then improvement scores
      r_indices = jnp.lexsort((sols_entries[:,2], sols_entries[:,1], sols_entries[:,0]))
      ranked_sols_entries = jnp.flip(sols_entries[r_indices], axis=0)

      # we want the descending order of the rankings and not the ascending
      indices = ranked_sols_entries[:,2].astype(jnp.int32)
      # Expected to be a statically defined if statement evaluated at jit compile time of 
this function
      num_parents = (new_sols if selection_rule == SelectionRules.FILTER else 
gradImprEmitter.num_parents)

      opt = 
gradImprEmitter.opt.tell(gradImprEmitter._get_static_settings_bd_opt(static_settings),
                                    gradImprEmitter.opt, 
gradImprEmitter.grad_coefficients[indices], num_parents)
      gradImprEmitter = gradImprEmitter.replace(opt = opt)

      # Calculate a new mean in solution space
      parents = solutions[indices]
      # Create a mask for identifying the solutions that are parents
      temp_indices = jnp.arange(0, batch_size, 1)
      parents_mask = jnp.where(temp_indices < num_parents, 1, 0)
      weights = (jnp.log(num_parents + 0.5) -
              jnp.log(jnp.arange(1, batch_size + 1)))
      # make the non-parent entries to zero arrays so that the sum is not affected
      # by the extra entries in the array (elements that are not parents)
      filtered_weights = jnp.multiply(weights, parents_mask)
      total_weights = jnp.sum(filtered_weights)
      weights = filtered_weights / total_weights
      # calcualate the new mean
      masked_parents = jnp.multiply(parents, jnp.expand_dims(parents_mask, axis=-1))
      new_mean = jnp.sum(jnp.multiply(masked_parents, jnp.expand_dims(weights, axis=1)), 
axis=0)

      # Use the mean to calculate a gradient step and step the optimizer
      gradient_step = new_mean - gradImprEmitter.gradient_opt.theta
      gradient_opt = gradImprEmitter.gradient_opt.step(obj_optimiser_settings, 
gradImprEmitter.gradient_opt, gradient_step)

      gradImprEmitter = gradImprEmitter.replace(gradient_opt = gradient_opt)
      should_reset_opt = jnp.logical_or(
                                    gradImprEmitter.opt.check_stop(opt,
ranked_sols_entries[:,1], num_parents),
                                    gradImprEmitter._check_restart(restart_rule, new_sols)
                                )

      key, elite_key = jax.random.split(key, 2)

      return lax.cond(should_reset_opt,
              lambda x: gradImprEmitter._reset_opt(static_settings, gradImprEmitter,
repertoire, elite_key),
              lambda x: (gradImprEmitter, repertoire),
              None
          )
```

Code Snippet A.15 Implementation of the Gradient Improvement Emitter in JAX (_gradient_improvement_emitter.py)

A.5 QD Emitters (PyTree Version)

A.5.1 MAP-Elites (Isotropic Gaussian)

```
"""Provides the GaussianEmitter.
https://github.com/icaros-usc/dqd/blob/main/ribs/emitters/_gaussian_emitter.py
"""

from typing import Any
import jax.numpy as jnp
import flax
import jax
from qd_utils.grid_archive import Repertoire
from training.ext_emitters.emitters_utils import EmitterBase
from collections import namedtuple

D_TYPE = jnp.float32
Array = jnp.ndarray

@flax.struct.dataclass
class GaussianEmitter:
    """Emits solutions by adding Gaussian noise to existing archive solutions.

    If the archive is empty, calls to :meth:`ask` will generate solutions from a
    user-specified Gaussian distribution with mean ``x0`` and standard deviation
```

```
``sigma0``. Otherwise, this emitter selects solutions from the archive and
generates solutions from a Gaussian distribution centered around each
solution with standard deviation ``sigma0``.

This is the classic variation operator presented in `Mouret 2015
<https://arxiv.org/pdf/1504.04909.pdf>`_.

"""
x0: Array
sigma0: D_TYPE
lower_bounds: Array
upper_bounds: Array
batch_size: jnp.int32
solution_dim: jnp.int32

@classmethod
def create(cls,
             x0,
             sigma0,
             batch_size,
             bounds=None):
    batch_size = batch_size
    sigma0 = float(sigma0)
    (solution_dim, ravel_single, unravel_single, ravel_batch_size,
            unravel_batch_size) = EmitterBase.get_ravel_info(batch_size, x0)
    x0_raveled = jnp.array(ravel_single(x0), dtype=D_TYPE)
    lower_bounds, upper_bounds = EmitterBase.process_bounds(bounds, solution_dim)
    # Define the Static Settings
    ssd = dict()
    ssd['batch_size'] = batch_size
    ssd['solution_dim'] = solution_dim
    ssd['ravel_single'] = ravel_single
    ssd['unravel_single'] = unravel_single
    ssd['ravel_batch_size'] = ravel_batch_size
    ssd['unravel_batch_size'] = unravel_batch_size
    StaticSettings = namedtuple('StaticSettings', ssd)

    return (cls(x0_raveled, sigma0, lower_bounds, upper_bounds, batch_size, solution_dim),
                    StaticSettings(**ssd))

@staticmethod
def ravel_single(static_settings, params):
    return static_settings.ravel_single(params)

@staticmethod
def unravel_single(static_settings, params):
    return static_settings.unravel_single(params)
    @staticmethod
    def ravel_batch_size(static_settings, params):
        return static_settings.ravel_batch_size(params)

    @staticmethod
    def unravel_batch_size(static_settings, params):
        return static_settings.unravel_batch_size(params)

    @staticmethod
    def _get_batch_size(static_settings):
        return static_settings.batch_size

    @staticmethod
    def _get_solution_dim(static_settings):
        return static_settings.solution_dim

    @staticmethod
    def _ask_clip(parents, lower_bounds, upper_bounds):
        return jnp.minimum(jnp.maximum(parents, lower_bounds), upper_bounds)

    @staticmethod
    def ask(static_settings, gaussian_emitter, repertoire, key):
        """Creates solutions by adding Gaussian noise to elites in the archive.
        """
        batch_size = gaussian_emitter._get_batch_size(static_settings)
        solution_dim = gaussian_emitter._get_solution_dim(static_settings)
        key_selection, key_variation = jax.random.split(key, 2)

        # SELECTION #
        idx_p1 = jax.random.randint(key_selection, shape=(batch_size,), minval=0,
maxval=repertoire.num_indivs)
        tot_indivs = repertoire.fitness.ravel().shape[0]
        indexes = jnp.argwhere(jnp.logical_not(jnp.isnan(repertoire.fitness)), size =
tot_indivs)
        indexes = jnp.transpose(indexes, axes=(1, 0))
        indiv_indices = jnp.array(jnp.ravel_multi_index(indexes, repertoire.fitness.shape,
mode='clip')).astype(int)

        idx_p1 = indiv_indices.at[idx_p1].get()
        sols = jax.tree_map(lambda x: x.at[idx_p1].get(),repertoire.archive)

        sols = gaussian_emitter.ravel_batch_size(static_settings, sols)
        # # VARIATION - MUTATION #
        # # Better approach since it operates directly on the tree
        # # structure of the solutions
        # num_vars = len(jax.tree_leaves(sols))
        # treedef = jax.tree_structure(sols)
        # all_keys = jax.random.split(key_variation, num=num_vars)

        # # Gaussian noise
        # noise = jax.tree_multimap(
        #     lambda g, k: jax.random.normal(k, shape=g.shape, dtype=g.dtype), sols,
        #     jax.tree_unflatten(treedef, all_keys))

        # # Added noise in positive direction
        # mutated_sols = jax.tree_multimap(lambda g, n: g + n * gaussian_emitter.sigma0, sols,
noise)
        # # Added noise in negative direction
        # anit_mutated_sols = jax.tree_multimap(lambda g, n: g - n * gaussian_emitter.sigma0,
sols, noise)

        # return gaussian_emitter, mutated_sols

        noise = jax.random.normal(key_variation, shape=(batch_size, solution_dim),
dtype=D_TYPE) * gaussian_emitter.sigma0
        new_sols = gaussian_emitter._ask_clip(sols + noise,
                                             gaussian_emitter.lower_bounds,
                                             gaussian_emitter.upper_bounds)
        return gaussian_emitter, gaussian_emitter.unravel_batch_size(static_settings,
new_sols)

    @staticmethod
    def tell(static_settings, gaussian_emitter, solutions, objective_values, behavior_values,
dead, repertoire, key):
        """Inserts entries into the archive.
        """
        repertoire = repertoire.add_to_archive(repertoire = repertoire,
                                                pop_p = solutions,
                                                bds = behavior_values,
                                                eval_scores = objective_values,
                                                dead = dead)
        return gaussian_emitter, repertoire
```

Code Snippet A.16: Implementation of the MAP-Elites with Isotropic Gaussian in JAX (_gaussian_emitter.py)

A.5.2 MAP-Elites (Iso + LineDD)

```
"""Provides the IsoLineEmitter.
Adapted from https://github.com/icaros-usc/dqd/blob/main/ribs/emitters/_iso_line_emitter.py
"""


import jax.numpy as jnp
import flax
import jax
```

```python
from qdax.training.ext_emitters.emitters_utils import EmitterBase
from collections import namedtuple

D_TYPE = jnp.float32
Array = jnp.ndarray

@flax.struct.dataclass
class IsoLineEmitter:
    """Emits solutions that are nudged towards other archive solutions.
    If the archive is empty, calls to :meth:`ask` will generate solutions from
    an isotropic Gaussian distribution with mean ``x0`` and standard deviation
    ``iso_sigma``. Otherwise, to generate each new solution, the emitter selects
    a pair of elites :math:`x_i` and :math:`x_j` and samples from
    .. math::
        x_i + \\sigma_{iso} \\mathcal{N}(0,\\mathcal{I}) +
            \\sigma_{line}(x_j - x_i)\\mathcal{N}(0,1)
    This emitter is based on the Iso+LineDD operator presented in `Vassiliades
    2018 <https://arxiv.org/abs/1804.03906>`_.
    """
    x0: Array
    iso_sigma: D_TYPE
    line_sigma: D_TYPE
    lower_bounds: Array
    upper_bounds: Array
    batch_size: jnp.int32
    solution_dim: jnp.int32


    @classmethod
    def create(cls,
                 x0,
                 iso_sigma,
                 line_sigma,
                 batch_size,
                 bounds=None):
        (solution_dim, ravel_single, unravel_single, ravel_batch_size,
                    unravel_batch_size) = EmitterBase.get_ravel_info(batch_size, x0)
        x0_raveled = jnp.array(ravel_single(x0), dtype=D_TYPE)
        lower_bounds, upper_bounds = EmitterBase.process_bounds(bounds, solution_dim)
        batch_size = batch_size
        iso_sigma = float(iso_sigma)
        line_sigma = float(line_sigma)
        ssd = dict()
        ssd['batch_size'] = batch_size
        ssd['solution_dim'] = solution_dim
        ssd['ravel_single'] = ravel_single
        ssd['unravel_single'] = unravel_single
        ssd['ravel_batch_size'] = ravel_batch_size
        ssd['unravel_batch_size'] = unravel_batch_size
        StaticSettings = namedtuple('StaticSettings', ssd)

        return (cls(x0_raveled, iso_sigma, line_sigma, lower_bounds,
                    upper_bounds, batch_size,
                     solution_dim), StaticSettings(**ssd))

    @staticmethod
    def ravel_single(static_settings, params):
        return static_settings.ravel_single(params)

    @staticmethod
    def unravel_single(static_settings, params):
        return static_settings.unravel_single(params)

    @staticmethod
    def ravel_batch_size(static_settings, params):
        return static_settings.ravel_batch_size(params)

    @staticmethod
    def unravel_batch_size(static_settings, params):
        return static_settings.unravel_batch_size(params)

    @staticmethod
    def _get_batch_size(static_settings):
        return static_settings.batch_size

    @staticmethod
    def _get_solution_dim(static_settings):
        return static_settings.solution_dim

    @staticmethod
    def _ask_clip(parents, lower_bounds, upper_bounds):
        return jnp.minimum(jnp.maximum(parents, lower_bounds), upper_bounds)

    @staticmethod
    def ask(static_settings, iso_emitter, repertoire, key):
        """Generates ``batch_size`` solutions.
        """
        batch_size = iso_emitter._get_batch_size(static_settings)
        solution_dim = iso_emitter._get_solution_dim(static_settings)
        key_selection, key_variation = jax.random.split(key, 2)

        # SELECTION #
        key_select_p1, key_select_p2 = jax.random.split(key_selection, 2)
        idx_s1 = jax.random.randint(key_select_p1, shape=(batch_size,),
                    minval=0, maxval=repertoire.num_indivs)
        idx_s2 = jax.random.randint(key_select_p2, shape=(batch_size,),
                    minval=0, maxval=repertoire.num_indivs)
        tot_indivs = repertoire.fitness.ravel().shape[0]
        indices = jnp.argwhere(jnp.logical_not(jnp.isnan(repertoire.fitness)),
                        size = tot_indivs)
        indices = jnp.transpose(indices, axes=(1, 0))
        indiv_indices = jnp.array(jnp.ravel_multi_index(indices,
                    repertoire.fitness.shape, mode='clip')).astype(int)

        idx_s1 = indiv_indices.at[idx_s1].get()
        idx_s2 = indiv_indices.at[idx_s2].get()
        sols_1 = jax.tree_map(lambda x: x.at[idx_s1].get(),repertoire.archive)
        sols_2 = jax.tree_map(lambda x: x.at[idx_s2].get(),repertoire.archive)

        # # VARIATION #
        # # Better approach since it operates directly on the tree
        # # structure of the solutions
        # num_vars = len(jax.tree_leaves(sols_1))
        # treedef = jax.tree_structure(sols_1)
        # key_a, key_b = jax.random.split(key_variation, 2)
        # all_keys_a = jax.random.split(key_a, num_vars)
        # all_keys_b = jax.random.split(key_b, num_vars)

        # noise_a = jax.tree_multimap(
        #     lambda g, k: jax.random.normal(k, shape=g.shape, dtype=g.dtype), sols_1,
        #     jax.tree_unflatten(treedef, all_keys_a))
        # noise_b = jax.tree_multimap(
        #     lambda g, k: jax.random.normal(k, shape=g.shape, dtype=g.dtype), sols_2,
        #     jax.tree_unflatten(treedef, all_keys_b))

        # new_sols = jax.tree_multimap(lambda x, y, a, b:
        #                             x + a * iso_emitter.iso_sigma +
        #                             b * iso_emitter.line_sigma * (x - y),
        #                                 sols_1, sols_2, noise_a, noise_b)

        # return iso_emitter, new_sols

        sols_1_raveled = iso_emitter.ravel_batch_size(static_settings, sols_1)
        sols_2_raveled = iso_emitter.ravel_batch_size(static_settings, sols_2)

        key_a, key_b = jax.random.split(key_variation, 2)
        iso_gaussian = jax.random.normal(key_a,
                            shape=(batch_size, solution_dim),
                            dtype=D_TYPE) * iso_emitter.iso_sigma

        # expanded last dimension used for multiplication later
```

```
        line_gaussian = jax.random.normal(key_b,
                                shape=(batch_size, 1),
                                dtype=D_TYPE) * iso_emitter.line_sigma

        directions = (sols_1_raveled - sols_2_raveled).astype(D_TYPE)

        new_sols = sols_2_raveled + iso_gaussian + jnp.multiply(
                        jnp.array(line_gaussian), directions)

        new_sols =  iso_emitter._ask_clip(new_sols,
                        iso_emitter.lower_bounds, iso_emitter.upper_bounds)

        return iso_emitter, iso_emitter.unravel_batch_size(static_settings, new_sols)

    @staticmethod
    def tell(static_settings, iso_emitter, solutions, objective_values,
                    behavior_values, dead, repertoire, key):
        """Inserts entries into the archive.
        """
        repertoire = repertoire.add_to_archive(repertoire = repertoire,
                                    pop_p = solutions,
                                    bds = behavior_values,
                                    eval_scores = objective_values,
                                    dead = dead)
        return iso_emitter, repertoire
```

Code Snippet A.17: Implementation of the MAP-Elites (Iso + LineDD) in JAX (_iso_line_emitter.py)

A.5.3 Covariance Matrix Adaptation MAP-Elites (CMA-ME) - Improvement

```
"""Provides the ImprovementEmitter.
Adapted from https://github.com/icaros-usc/dqd/blob/main/ribs/emitters/_improvement_emitter.py
"""

from typing import Any
import jax.numpy as jnp
import flax
import jax
from jax import lax
import functools
from qd_utils.grid_archive import Repertoire
from training.ext_emitters.emitters_utils import EmitterBase
from training.opt._cma_es import CMAEvolutionStrategy, WeightRules
from collections import namedtuple

Array = jnp.ndarray

class SelectionRules:
    MU = 1
    FILTER = 2

class RestartRules:
    BASIC = 1
    NO_IMPROVEMENT = 2

D_TYPE = jnp.float32

@flax.struct.dataclass
class ImprovementEmitter:
    """Adapts a covariance matrix towards changes in the archive.

    This emitter originates in `Fontaine 2020
    <https://arxiv.org/abs/1912.02400>`_. Initially, it starts at ``x0`` and
    uses CMA-ES to search for solutions that improve the archive, i.e. solutions
    that add new entries to the archive or improve existing entries. Once CMA-ES
    restarts (see ``restart_rule``), the emitter starts from a randomly chosen
    elite in the archive and continues searching for solutions that improve the
    archive.
    """

    x0: Array
    sigma0: D_TYPE
    lower_bounds: Array
    upper_bounds: Array
    batch_size: jnp.int32
    opt: CMAEvolutionStrategy
    solution_dim: jnp.int32
    num_parents: jnp.int32
    restarts: jnp.int32
    restart_rule: jnp.int32
    selection_rule: jnp.int32

    @classmethod
    def create(cls,
               x0,
               sigma0,
               selection_rule=SelectionRules.FILTER,
               restart_rule=RestartRules.NO_IMPROVEMENT,
               weight_rule=WeightRules.TRUNCATION,
               bounds=None,
               batch_size=None):
        (solution_dim, ravel_single, unravel_single, ravel_batch_size,
                  unravel_batch_size) = EmitterBase.get_ravel_info(batch_size, x0)
        x0_raveled = jnp.array(ravel_single(x0), dtype=D_TYPE)
        sigma0 = sigma0
        lower_bounds, upper_bounds = EmitterBase.process_bounds(bounds, solution_dim)
        batch_size = batch_size

        if selection_rule not in [SelectionRules.MU, SelectionRules.FILTER]:
            raise ValueError(f"Invalid selection_rule {selection_rule}")

        if restart_rule not in [RestartRules.BASIC, RestartRules.NO_IMPROVEMENT]:
            raise ValueError(f"Invalid restart_rule {restart_rule}")

        opt, ssd_opt = CMAEvolutionStrategy.create(sigma0, batch_size,
                                                    solution_dim, weight_rule)
        opt = opt.reset(ssd_opt, opt, x0_raveled)
        get_num_of_parents_fn = jax.jit(functools.partial(
                    cls._get_num_of_parents, selection_rule))
        num_parents = (opt.batch_size // 2 if selection_rule ==
                                   SelectionRules.MU else None)
        batch_size = opt.batch_size

        ssd = dict()
        ssd['get_num_of_parents_fn'] = get_num_of_parents_fn
        ssd['solution_dim'] = solution_dim
        ssd['batch_size'] = batch_size
        ssd['restart_rule'] = restart_rule
        ssd['opt_settings'] = ssd_opt
        ssd['ravel_single'] = ravel_single
        ssd['unravel_single'] = unravel_single
        ssd['ravel_batch_size'] = ravel_batch_size
        ssd['unravel_batch_size'] = unravel_batch_size
        StaticSettings = namedtuple('StaticSettings', ssd)

        restarts = 0
        return (cls(x0_raveled, sigma0, lower_bounds, upper_bounds , batch_size, opt,
                    solution_dim, num_parents, restarts, restart_rule, selection_rule),
                    StaticSettings(**ssd))

    @staticmethod
    def ravel_single(static_settings, params):
        return static_settings.ravel_single(params)

    @staticmethod
    def unravel_single(static_settings, params):
        return static_settings.unravel_single(params)

    @staticmethod
    def ravel_batch_size(static_settings, params):
        return static_settings.ravel_batch_size(params)
    @staticmethod
    def unravel_batch_size(static_settings, params):
        return static_settings.unravel_batch_size(params)

    @staticmethod
    def _get_batch_size(static_settings):
        return static_settings.batch_size

    @staticmethod
    def _get_restart_rule(static_settings):
        return static_settings.restart_rule

    @staticmethod
    def _get_solution_dim(static_settings):
        return static_settings.solution_dim

    @staticmethod
    def _get_num_of_parents_fn(static_settings):
        return static_settings.get_num_of_parents_fn

    @staticmethod
    def _get_optimiser_settings(static_settings):
        return static_settings.opt_settings

    @staticmethod
    def _get_num_of_parents(selection_rule, new_sols, num_parents):
        return (new_sols if selection_rule == SelectionRules.FILTER else num_parents)

    @staticmethod
    def ask(static_settings, imp_emitter, repertoire, key):
        """Samples new solutions from a multivariate Gaussian.

        The multivariate Gaussian is parameterized by the CMA-ES optimizer.
        """
        opt, solutions = imp_emitter.opt.ask(
                    imp_emitter._get_optimiser_settings(static_settings),
                    imp_emitter.opt, imp_emitter.lower_bounds,
                    imp_emitter.upper_bounds, key)

        unraveled_sols = imp_emitter.unravel_batch_size(static_settings, solutions)

        return imp_emitter.replace(opt = opt), unraveled_sols

    @staticmethod
    def _check_restart(restart_rule, num_parents):
        """Emitter-side checks for restarting the optimizer.
        The optimizer also has its own checks.
        """
        return restart_rule == RestartRules.NO_IMPROVEMENT and num_parents == 0

    @staticmethod
    def _reset_opt(static_settings, imp_emitter, repertoire, elite_key):
        new_x0 = Repertoire.get_random_elite_ext(repertoire, elite_key)
        new_x0 = imp_emitter.ravel_single(static_settings, new_x0)
        opt = imp_emitter.opt.reset(imp_emitter._get_optimiser_settings(
                        static_settings), imp_emitter.opt, new_x0)
        restarts = imp_emitter.restarts + 1
        return imp_emitter.replace(opt = opt, restarts = restarts), repertoire

    @staticmethod
    def tell(static_settings, imp_emitter, solutions, objective_values,
             behavior_values, dead, repertoire, key):
        """Gives the emitter results from evaluating solutions.

        As solutions are inserted into the archive, we record their "improvement
        value" -- conveniently, this is the ``value`` returned by
        :meth:`ribs.archives.ArchiveBase.add`. We then rank the solutions
        according to their add status (new solutions rank in front of
        solutions that improved existing entries in the archive, which rank
        ahead of solutions that were not added), followed by their improvement
        value.  We then pass the ranked solutions to the underlying CMA-ES
        optimizer to update the search parameters.
        """
        get_num_of_parents_fn = ImprovementEmitter._get_num_of_parents_fn(static_settings)
        solution_dim = ImprovementEmitter._get_solution_dim(static_settings)
        restart_rule = ImprovementEmitter._get_restart_rule(static_settings)
        # new_sols = repertoire.num_indivs
        repertoire, sols_entries, new_sols = repertoire.add_to_archive_extended(
                                            repertoire = repertoire,
                                            pop_p = solutions,
                                            bds = behavior_values,
                                            eval_scores = objective_values,
                                            dead = dead)
        # Find the indices of the sorted array on status codes and then objective/evaluation
scores
        r_indices = jnp.lexsort((sols_entries[:,2], sols_entries[:,1], sols_entries[:,0]))
        ranked_sols_entries = jnp.flip(sols_entries[r_indices], axis=0)
        # new_sols = repertoire.num_indivs - new_sols
        # we want the descending order of the rankings and not the ascending
        indices = ranked_sols_entries[:,2].astype(jnp.int32)
        num_parents = get_num_of_parents_fn(new_sols, imp_emitter.num_parents)
        raveled_sols = imp_emitter.ravel_batch_size(static_settings, solutions)
        opt = CMAEvolutionStrategy.tell(imp_emitter._get_optimiser_settings(static_settings),
                                imp_emitter.opt, raveled_sols[indices], num_parents)

        key, elite_key = jax.random.split(key, 2)

        imp_emitter = imp_emitter.replace(opt = opt)

        should_reset_opt = jnp.logical_or(
                                    CMAEvolutionStrategy.check_stop(opt,
                                    ranked_sols_entries[:,1], num_parents),
                                    ImprovementEmitter._check_restart(restart_rule, new_sols)
                                )

        return lax.cond(should_reset_opt,
                lambda x: ImprovementEmitter._reset_opt(static_settings, imp_emitter,
                        repertoire, elite_key),
                lambda x: (imp_emitter, repertoire),
                None
            )
```

Code Snippet A.18: Implementation of CMA-ME Improvement the in JAX (_improvement_emitter.py)

A.5.4 Covariance Matrix Adaptation MAP-Elites (CMA-ME) - Optimizing

```
"""Provides the OptimizingEmitter.
Adapted from https://github.com/icaros-usc/dqd/blob/main/ribs/emitters/_optimizing_emitter.py
"""
from typing import Any
import jax.numpy as jnp
import flax
import jax
from jax import lax
from qd_utils.grid_archive import Repertoire
from training.ext_emitters.emitters_utils import EmitterBase
from training.opt._cma_es import CMAEvolutionStrategy, WeightRules
from collections import namedtuple

Array = jnp.ndarray

class SelectionRules:
    MU = 1
    FILTER = 2

class RestartRules:
    BASIC = 1
    NO_IMPROVEMENT = 2
```

```python
D_TYPE = jnp.float32

@flax.struct.dataclass
class OptimizingEmitter:
    """Adapts a covariance matrix towards the objective.

    This emitter originates in `Fontaine 2020
    <https://arxiv.org/abs/1912.02400>`_. Initially, it starts at ``x0`` and
    uses CMA-ES to optimize for objective values. After CMA-ES converges, the
    emitter restarts the optimizer. It picks a random elite in the archive and
    begins optimizing from there.
    """

    x0: Array
    sigma0: D_TYPE
    lower_bounds: Array
    upper_bounds: Array
    batch_size: jnp.int32
    opt: CMAEvolutionStrategy
    solution_dim: jnp.int32
    num_parents: jnp.int32
    restarts: jnp.int32
    restart_rule: jnp.int32
    selection_rule: jnp.int32

    @classmethod
    def create(cls,
               x0,
               sigma0,
               selection_rule=SelectionRules.FILTER,
               restart_rule=RestartRules.NO_IMPROVEMENT,
               weight_rule=WeightRules.TRUNCATION,
               bounds=None,
               batch_size=None):
        (solution_dim, ravel_single, unravel_single, ravel_batch_size,
                     unravel_batch_size) = EmitterBase.get_ravel_info(
                         batch_size, x0)
        x0_raveled = jnp.array(ravel_single(x0), dtype=D_TYPE)
        sigma0 = sigma0
        lower_bounds, upper_bounds = EmitterBase.process_bounds(bounds,
                                                                solution_dim)
        batch_size = batch_size

        if selection_rule not in [SelectionRules.MU, SelectionRules.FILTER]:
            raise ValueError(f"Invalid selection_rule {selection_rule}")

        if restart_rule not in [RestartRules.BASIC, RestartRules.NO_IMPROVEMENT]:
            raise ValueError(f"Invalid restart_rule {restart_rule}")

        opt, ssd_opt = CMAEvolutionStrategy.create(sigma0, batch_size,
                                                    solution_dim, weight_rule)
        opt = opt.reset(ssd_opt, opt, x0_raveled)
        # get_num_of_parents_fn = jax.jit(functools.partial(cls._get_num_of_parents,
selection_rule))
        num_parents = (opt.batch_size // 2 if selection_rule == SelectionRules.MU else None)
        batch_size = opt.batch_size

        ssd = dict()
        # static_settings['get_num_of_parents_fn'] = get_num_of_parents_fn
        ssd['solution_dim'] = solution_dim
        ssd['batch_size'] = batch_size
        ssd['restart_rule'] = restart_rule
        ssd['selection_rule'] = selection_rule
        ssd['opt_settings'] = ssd_opt
        ssd['ravel_single'] = ravel_single
        ssd['unravel_single'] = unravel_single
        ssd['ravel_batch_size'] = ravel_batch_size
        ssd['unravel_batch_size'] = unravel_batch_size
        StaticSettings = namedtuple('StaticSettings', ssd)

        restarts = 0
        return (cls(x0_raveled, sigma0, lower_bounds, upper_bounds , batch_size, opt,
                    solution_dim, num_parents, restarts, restart_rule, selection_rule),
                    StaticSettings(**ssd))

    @staticmethod
    def ravel_single(static_settings, params):
        return static_settings.ravel_single(params)

    @staticmethod
    def unravel_single(static_settings, params):
        return static_settings.unravel_single(params)

    @staticmethod
    def ravel_batch_size(static_settings, params):
        return static_settings.ravel_batch_size(params)

    @staticmethod
    def unravel_batch_size(static_settings, params):
        return static_settings.unravel_batch_size(params)

    @staticmethod
    def _get_batch_size(static_settings):
        return static_settings.batch_size
    @staticmethod
    def _get_restart_rule(static_settings):
        return static_settings.restart_rule

    @staticmethod
    def _get_solution_dim(static_settings):
        return static_settings.solution_dim

    @staticmethod
    def _get_selection_rule(static_settings):
        return static_settings.selection_rule

    @staticmethod
    def _get_optimiser_settings(static_settings):
        return static_settings.opt_settings

    @staticmethod
    def ask(static_settings, optEmitter, repertoire, key):
        """Samples new solutions from a multivariate Gaussian.

        The multivariate Gaussian is parameterized by the CMA-ES optimizer.
        """
        opt, solutions = optEmitter.opt.ask(
                optEmitter._get_optimiser_settings(static_settings),
                optEmitter.opt,
                optEmitter.lower_bounds,
                optEmitter.upper_bounds, key)
        unraveled_sols = optEmitter.unravel_batch_size(static_settings,
                                solutions)
        return optEmitter.replace(opt = opt), unraveled_sols

    @staticmethod
    def _check_restart(restart_rule, num_parents):
        """Emitter-side checks for restarting the optimizer.

        The optimizer also has its own checks.
        """
        return restart_rule == RestartRules.NO_IMPROVEMENT and num_parents == 0

    @staticmethod
    def _reset_opt(static_settings, optEmitter, repertoire, elite_key):
        new_x0 = Repertoire.get_random_elite_ext(repertoire, elite_key)
        new_x0 = optEmitter.ravel_single(static_settings, new_x0)
        opt = optEmitter.opt.reset(optEmitter._get_optimiser_settings(
                        static_settings), optEmitter.opt, new_x0)
        restarts = optEmitter.restarts + 1
        return optEmitter.replace(opt = opt, restarts = restarts), repertoire

    @staticmethod
    def tell(static_settings, optEmitter, solutions, objective_values,
                    behavior_values, dead, repertoire, key):
        """Gives the emitter results from evaluating solutions.

        As solutions are inserted into the archive, we record their "improvement
        value" -- conveniently, this is the ``value`` returned by
        :meth:`ribs.archives.ArchiveBase.add`. We then rank the solutions
        according to their add status (new solutions rank in front of
        solutions that improved existing entries in the archive, which rank
        ahead of solutions that were not added), followed by their improvement
        value.  We then pass the ranked solutions to the underlying CMA-ES
        optimizer to update the search parameters.
        """
        solution_dim = optEmitter._get_solution_dim(static_settings)
        restart_rule = optEmitter._get_restart_rule(static_settings)
        selection_rule = optEmitter._get_selection_rule(static_settings)

        repertoire, sols_entries, new_sols = repertoire.add_to_archive_extended(
                                        repertoire = repertoire,
                                        pop_p = solutions,
                                        bds = behavior_values,
                                        eval_scores = objective_values,
                                        dead = dead)

        if selection_rule == SelectionRules.FILTER:
            # Sort by whether the solution was added into the archive, followed
            # by objective value.
            sort_elements = (objective_values[sols_entries[:,2].astype(jnp.int32)],
                                sols_entries[:,0])
        elif selection_rule == SelectionRules.MU:
            # Sort only by objective value.
            sort_elements = (objective_values[sols_entries[:,2].astype(jnp.int32)])

        # Find the indices of the sorted array on status codes and then objective/evaluation
scores
        r_indices = jnp.lexsort(sort_elements)
        ranked_sols_entries = jnp.flip(sols_entries[r_indices], axis=0)
        # we want the descending order of the rankings and not the ascending
        indices = ranked_sols_entries[:,2].astype(jnp.int32)

        num_parents = (new_sols if selection_rule == SelectionRules.FILTER
                                else optEmitter.num_parents)
        raveled_sols = optEmitter.ravel_batch_size(static_settings, solutions)
        opt = optEmitter.opt.tell(optEmitter._get_optimiser_settings(static_settings),
                            optEmitter.opt, raveled_sols[indices], num_parents)

        key, elite_key = jax.random.split(key, 2)

        optEmitter = optEmitter.replace(opt = opt)

        should_reset_opt = jnp.logical_or(
                                   CMAEvolutionStrategy.check_stop(opt,
                                       ranked_sols_entries[:,1], num_parents),
                                   optEmitter._check_restart(restart_rule, new_sols)
                               )

        return lax.cond(should_reset_opt,
                lambda x: optEmitter._reset_opt(static_settings, optEmitter,
                            repertoire, elite_key),
                lambda x: (optEmitter, repertoire),
                None
            )
```

Code Snippet A.19: Implementation of the CMA-ME - Optimizing in JAX (_optimizing_emitter.py)

A.5.5 Covariance Matrix Adaptation MAP-Elites (CMA-ME) – Random Direction

```
"""Provides the RandomDirectionEmitter.
Adapted from https://github.com/icaros-
usc/dqd/blob/main/ribs/emitters/_random_direction_emitter.py
"""
from typing import Any
import jax.numpy as jnp
import flax
import jax
from jax import lax
from qd_utils.grid_archive import Repertoire
from training.ext_emitters.emitters_utils import EmitterBase
from training.opt._cma_es import CMAEvolutionStrategy, WeightRules
from collections import namedtuple

Array = jnp.ndarray

class SelectionRules:
    MU = 1
    FILTER = 2

class RestartRules:
    BASIC = 1
    NO_IMPROVEMENT = 2
```

```python
D_TYPE = jnp.float32

@flax.struct.dataclass
class RandomDirectionEmitter(EmitterBase):
    """Performs a random walk in behavior space by pursuing randomly chosen
    behavior space directions.

    This emitter originates in `Fontaine 2020
    <https://arxiv.org/abs/1912.02400>`_. Initially, it starts at ``x0`` and
    uses CMA-ES to search for solutions along a randomly chosen direction. Once
    CMA-ES restarts (see ``restart_rule``), the emitter starts from a randomly
    chosen elite in the archive and pursues a new random direction.
    """

    x0: Array
    sigma0: D_TYPE
    lower_bounds: Array
    upper_bounds: Array
    batch_size: jnp.int32
    opt: CMAEvolutionStrategy
    solution_dim: jnp.int32
    num_parents: jnp.int32
    restarts: jnp.int32
    restart_rule: jnp.int32
    selection_rule: jnp.int32
    archive_bounds: Array
    target_behavior_dir:Array

    @classmethod
    def create(cls,
               x0,
               sigma0,
               archive_bounds,
               key,
               selection_rule=SelectionRules.FILTER,
               restart_rule=RestartRules.NO_IMPROVEMENT,
               weight_rule=WeightRules.TRUNCATION,
               bounds=None,
               batch_size=None):
        (solution_dim, ravel_single, unravel_single, ravel_batch_size,
                  unravel_batch_size) = EmitterBase.get_ravel_info(
                                        batch_size, x0)
        x0_raveled = jnp.array(ravel_single(x0), dtype=D_TYPE)
        sigma0 = sigma0
        lower_bounds, upper_bounds = EmitterBase.process_bounds(
                           bounds, solution_dim)
        batch_size = batch_size
        if selection_rule not in [SelectionRules.MU, SelectionRules.FILTER]:
            raise ValueError(f"Invalid selection_rule {selection_rule}")

        if restart_rule not in [RestartRules.BASIC, RestartRules.NO_IMPROVEMENT]:
            raise ValueError(f"Invalid restart_rule {restart_rule}")

        opt, ssd_opt = CMAEvolutionStrategy.create(sigma0, batch_size,
                          solution_dim, weight_rule)
        opt = opt.reset(ssd_opt, opt, x0_raveled)

        num_parents = (opt.batch_size // 2 if selection_rule ==
                                     SelectionRules.MU else None)
        batch_size = opt.batch_size

        ssd = dict()
        ssd['solution_dim'] = solution_dim
        ssd['solution_dim'] = solution_dim
        ssd['batch_size'] = batch_size
        ssd['restart_rule'] = restart_rule
        ssd['selection_rule'] = selection_rule
        ssd['archive_bounds'] = archive_bounds
        ssd['opt_settings'] = ssd_opt
        ssd['ravel_single'] = ravel_single
        ssd['unravel_single'] = unravel_single
        ssd['ravel_batch_size'] = ravel_batch_size
        ssd['unravel_batch_size'] = unravel_batch_size
        StaticSettings = namedtuple('StaticSettings', ssd)
        target_behavior_dir = cls._generate_random_direction(ssd,
                            archive_bounds, key)

        restarts = 0
        return cls(x0, sigma0, lower_bounds, upper_bounds , batch_size,
                   opt, solution_dim, num_parents, restarts, restart_rule,
                   selection_rule, archive_bounds,
                   target_behavior_dir), StaticSettings(**ssd)

    @staticmethod
    def ravel_single(static_settings, params):
        return static_settings.ravel_single(params)

    @staticmethod
    def unravel_single(static_settings, params):
        return static_settings.unravel_single(params)

    @staticmethod
    def ravel_batch_size(static_settings, params):
        return static_settings.ravel_batch_size(params)

    @staticmethod
    def unravel_batch_size(static_settings, params):
        return static_settings.unravel_batch_size(params)

    @staticmethod
    def _get_batch_size(static_settings):
        return static_settings.batch_size

    @staticmethod
    def _get_restart_rule(static_settings):
        return static_settings.restart_rule

    @staticmethod
    def _get_solution_dim(static_settings):
        return static_settings.solution_dim

    @staticmethod
    def _get_selection_rule(static_settings):
        return static_settings.selection_rule

    @staticmethod
    def _get_archive_bounds(static_settings):
        return static_settings.archive_bounds

    @staticmethod
    def _get_optimiser_settings(static_settings):
        return static_settings.opt_settings


    @staticmethod
    def ask(static_settings, randomDirEmitter, repertoire, key):
        """Samples new solutions from a multivariate Gaussian.

        The multivariate Gaussian is parameterized by the CMA-ES optimizer.

        Returns:
            ``(batch_size, solution_dim)`` array -- contains ``batch_size`` new
            solutions to evaluate.
        """
        opt, solutions = randomDirEmitter.opt.ask(
                        randomDirEmitter._get_optimiser_settings(static_settings),
                        randomDirEmitter.opt,
                        randomDirEmitter.lower_bounds,
                        randomDirEmitter.upper_bounds, key)
        unraveled_sols = randomDirEmitter.unravel_batch_size(static_settings,
                                solutions)

        return randomDirEmitter.replace(opt = opt), unraveled_sols

    @staticmethod
    def _generate_random_direction(static_settings, archive_bounds, key):
        """Generates a new random direction in the behavior space.

        The direction is sampled from a standard Gaussian -- since the standard
        Gaussian is isotropic, there is equal probability for any direction. The
        direction is then scaled to the behavior space bounds.
        """
        ranges = archive_bounds[:,1] - archive_bounds[:,0]
        behavior_dim = archive_bounds.shape[0]
        unscaled_dir = jax.random.normal(key,
                                shape=(behavior_dim,),
                                dtype=D_TYPE)
        return unscaled_dir * ranges

    @staticmethod
    def _check_restart(restart_rule, num_parents):
        """Emitter-side checks for restarting the optimizer.

        The optimizer also has its own checks.
        """
        return restart_rule == RestartRules.NO_IMPROVEMENT and num_parents == 0

    @staticmethod
    def _reset_opt(static_settings, randomDirEmitter, repertoire, key):
        dir_key, elite_key = jax.random.split(key, 2)
        new_x0 = Repertoire.get_random_elite_ext(repertoire, elite_key)
        new_x0 = randomDirEmitter.ravel_single(static_settings, new_x0)
        opt = randomDirEmitter.opt.reset(randomDirEmitter._get_optimiser_settings(
                                            static_settings),
                                        randomDirEmitter.opt, new_x0)
        archive_bounds = randomDirEmitter._get_archive_bounds(static_settings)
        target_behavior_dir = randomDirEmitter._generate_random_direction(
                                            static_settings,
                                            archive_bounds, dir_key)
        restarts = randomDirEmitter.restarts + 1
        return randomDirEmitter.replace(opt = opt, restarts = restarts,
                        target_behavior_dir = target_behavior_dir), repertoire

    @staticmethod
    def tell(static_settings, randomDirEmitter, solutions, objective_values,
                    behavior_values, dead, repertoire, key):
```

```
        """Gives the emitter results from evaluating solutions.

        As solutions are inserted into the archive, we record their "improvement
        value" -- conveniently, this is the ``value`` returned by
        :meth:`ribs.archives.ArchiveBase.add`. We then rank the solutions
        according to their add status (new solutions rank in front of
        solutions that improved existing entries in the archive, which rank
        ahead of solutions that were not added), followed by their improvement
        value.  We then pass the ranked solutions to the underlying CMA-ES
        optimizer to update the search parameters.
        """
        solution_dim = randomDirEmitter._get_solution_dim(static_settings)
        restart_rule = randomDirEmitter._get_restart_rule(static_settings)
        selection_rule = randomDirEmitter._get_selection_rule(static_settings)

        repertoire, sols_entries, new_sols = repertoire.add_to_archive_extended(
                                        repertoire = repertoire,
                                        pop_p = solutions,
                                        bds = behavior_values,
                                        eval_scores = objective_values,
                                        dead = dead)
        projection = jnp.dot(behavior_values, randomDirEmitter.target_behavior_dir)
        # Rearrange based on their initial indices to make it parallel to sols_entries
        projection = projection[sols_entries[:,2].astype(jnp.int32)]
        if selection_rule == SelectionRules.FILTER:
            # Sort by whether the solution was added into the archive, followed
            # by projection.
            sort_elements = (sols_entries[:,2], projection, sols_entries[:,0])
        elif selection_rule == SelectionRules.MU:
            # Sort only by projection.
            sort_elements = (projection)

        # Find the indices of the sorted array on status codes and then objective/evaluation
scores
        r_indices = jnp.lexsort(sort_elements)
        ranked_sols_entries = jnp.flip(sols_entries[r_indices], axis=0)
        # we want the descending order of the rankings and not the ascending
        indices = ranked_sols_entries[:,2].astype(jnp.int32)

        num_parents = (new_sols if selection_rule == SelectionRules.FILTER
                                            else randomDirEmitter.num_parents)
        raveled_sols = randomDirEmitter.ravel_batch_size(static_settings, solutions)
        opt = randomDirEmitter.opt.tell(randomDirEmitter._get_optimiser_settings(
                                static_settings), randomDirEmitter.opt,
                                raveled_sols[indices], num_parents)

        key, reset_key = jax.random.split(key, 2)
        randomDirEmitter = randomDirEmitter.replace(opt = opt)

        should_reset_opt = jnp.logical_or(
                                    CMAEvolutionStrategy.check_stop(opt, projection,
                                        num_parents),
                                    randomDirEmitter._check_restart(restart_rule, new_sols)
                                )

        return lax.cond(should_reset_opt,
                lambda x: randomDirEmitter._reset_opt(static_settings, randomDirEmitter,
                                                    repertoire, reset_key),
                lambda x: (randomDirEmitter, repertoire),
                None
            )
```

Code Snippet A.20: Implementation of the CMA-ME Random Direction in JAX (_random_direction_emitter.py)

A.6 Emitters' Utilities

```
import jax
import jax.numpy as jnp
from jax.flatten_util import ravel_pytree
import functools
D_TYPE = jnp.float32

class EmitterBase:
    @staticmethod
    def process_bounds(bounds, solution_dim):
        """Processes the input bounds.

        Returns:
            tuple: Two arrays containing all the lower bounds and all the upper
                bounds.
        Raises:
            ValueError: There is an error in the bounds configuration.
        """
        lower_bounds = jnp.full(solution_dim, -jnp.inf, dtype=D_TYPE)
        upper_bounds = jnp.full(solution_dim, jnp.inf, dtype=D_TYPE)

        if bounds is None:
            return lower_bounds, upper_bounds
        # Handle array-like bounds.
        if len(bounds) != solution_dim:
            raise ValueError("If it is an array-like, bounds must have the "
                             "same length as x0")
        for idx, bnd in enumerate(bounds):
            if bnd is None:
                continue  # Bounds already default to -inf and inf.
            if len(bnd) != 2:
                raise ValueError("All entries of bounds must be length 2")
            lower_bounds = lower_bounds.at[idx].set(-jnp.inf
                                if bnd[0] is None else bnd[0])
            upper_bounds = upper_bounds.at[idx].set(jnp.inf
                                if bnd[1] is None else bnd[1])
        return lower_bounds, upper_bounds

    @staticmethod
    def ravel_single(params):
        return ravel_pytree(params)

    @staticmethod
    def unravel_single(unravel, params):
        return unravel(params)

    @staticmethod
    def ravel_batch_size(params):
        batch_size = jax.tree_util.tree_leaves(params)[0].shape[0]
        params_leaves_transposed = jax.tree_map(lambda x:
                                   jnp.transpose(x), params)
        flatten_params, unravel = ravel_pytree(params_leaves_transposed)
        raveled_sols = flatten_params.reshape(-1, batch_size)
        return jnp.transpose(raveled_sols), unravel

    @staticmethod
    def unravel_batch_size(unravel_f, params):
        flatten_params = jnp.ravel(params,  order='F')
        params_tree = unravel_f(flatten_params)
        params_tree = jax.tree_map(lambda x: jnp.transpose(x), params_tree)
        return params_tree

    @staticmethod
    def expand_params_to_batch_size(batch_size, params):
        return jax.tree_map(lambda x: jnp.repeat(jnp.expand_dims(x, axis=0),
                    batch_size , axis=0), params)



    @staticmethod
    def get_ravel_info(batch_size, x0):
        if isinstance(x0, jnp.ndarray):
            solution_dim = len(x0)
            ravel_single = lambda x: x
            unravel_single = lambda x: x
            ravel_batch_size = lambda x: x
            unravel_batch_size = lambda x: x
        else:
            # Calculate ravel and unravel method for a single instance of sols
            raveled_x, unravel_single_fn = EmitterBase.ravel_single(x0)
            ravel_single = lambda x: EmitterBase.ravel_single(x)[0]
            unravel_single = functools.partial(EmitterBase.unravel_single,
                            unravel_single_fn)
            # unravel_single = jax.tree_util.Partial(EmitterBase.unravel_single,
unravel_single_fn)
            solution_dim = len(raveled_x)
            # Calculate ravel and unravel method for a batch size instances of sols
            expanded_xs = EmitterBase.expand_params_to_batch_size(batch_size, x0)
            raveled_xs, unravel_bs_fn = EmitterBase.ravel_batch_size(expanded_xs)
            ravel_batch_size = lambda x: EmitterBase.ravel_batch_size(x)[0]
            unravel_batch_size = functools.partial(
                            EmitterBase.unravel_batch_size, unravel_bs_fn)
            # unravel_batch_size = jax.tree_util.Partial(
            #                 EmitterBase.unravel_batch_size, unravel_bs_fn)
        return (solution_dim, ravel_single, unravel_single, ravel_batch_size,
                        unravel_batch_size)
```

Code Snippet A.21: Emitters’ Utilities file (emitters_utils.py)

A.7 Containers

A.7.1 N-Dimensional Grid Archive

```
"""
Adapted from QDax:
https://github.com/adaptive-intelligent-robotics/QDax/blob/main/qdax/qd_utils/grid_archive.py
"""


from typing import Any,List
import numpy as np
import jax.numpy as jnp
import flax
import jax
from jax import jit,vmap,grad
from training import simple_emitters
Array = Any
```

```python
# Adding solutions to archive Status
# Higher number means higher importance
class AddStatus:
    NEW = 2
    IMPROVE_EXISTING = 1
    NOT_ADDED = 0

@flax.struct.dataclass
class Repertoire:
    archive: List
    fitness: Array
    bd: Array
    grid_shape: Array
    min: np.float64
    max: np.float64
    num_indivs: int
    indiv_indices: Array

    select_elite_from_repertoire = simple_emitters.get_select_from_repertoire_fn(1, 1)

    @staticmethod
    def get_random_elite(repertoire, key):
        return Repertoire.select_elite_from_repertoire(repertoire, key)[0][0]

    @staticmethod
    def get_random_elite_ext(repertoire, key):
        sols = Repertoire.select_elite_from_repertoire(repertoire, key)
        return jax.tree_map(lambda x: jnp.squeeze(x), sols[0])

    @staticmethod
    def _clip_bds( min_bound, max_bound, bds):
        lower_bounds, upper_bounds = (jnp.repeat(min_bound, bds.shape[1]),
                                    jnp.repeat(max_bound,bds.shape[1]))
        return jnp.minimum(jnp.maximum(bds, jnp.expand_dims(lower_bounds,axis=0)),
                                    jnp.expand_dims(upper_bounds,axis=0))

    @classmethod
    def create(cls, policy_params, max, min, grid_shape):
        grid_shape = jnp.array(grid_shape)
        num_indivs = 0
        indiv_indices = jnp.array([])

        bd = jnp.zeros(grid_shape)
        fitness = jnp.full(grid_shape,jnp.nan)
        #NOTE only 2D atm
        archive = jax.tree_map(lambda x: jnp.zeros(jnp.repeat(
                                jnp.expand_dims(x, axis=0), jnp.prod(grid_shape),
                        axis=0).shape), policy_params)
        return cls(archive, fitness, bd, grid_shape,  min, max, num_indivs, indiv_indices)

    @staticmethod
    def binning(normed, shape):
        return tuple(jnp.multiply(normed, shape - 1).astype(int))

    @staticmethod
    def add_to_archive(repertoire, pop_p, bds, eval_scores, dead):
        # bds = repertoire._clip_bds(repertoire.min, repertoire.max, bds)
        normalized_bds = ((bds-repertoire.min)/(repertoire.max-repertoire.min))  #Normlalized
BD should be between zero and 1
        bd_cells = jit(jax.vmap(Repertoire.binning, in_axes=(0,None),
                    out_axes=0))(normalized_bds,repertoire.grid_shape)
        # print(bd_cells)
        bd_indexes = jnp.ravel_multi_index(bd_cells, repertoire.bd.shape,mode = 'clip')
        maximum_fitness = jax.ops.segment_max(eval_scores, bd_indexes,
                    num_segments=repertoire.fitness.ravel().shape[0])
        eval_scores_filtered = jnp.where(maximum_fitness.at[bd_indexes].get()==eval_scores,
                        eval_scores,np.iinfo(np.int32).min)
        keep_eval_scores_filtered = jnp.where(maximum_fitness.at[bd_indexes].get()
                                ==eval_scores,True,False)
        # Checking Conditions for fitness function
        current_fitness = repertoire.fitness.ravel().at[bd_indexes].get()
        # Checking if fitness function is nan or not, since nan means we do not have an
individual yet
        current_fitness_nan = jnp.logical_and(jnp.isnan(current_fitness),
                                keep_eval_scores_filtered)
        # Checking if fitness that we have is better than the one we observed
        better_fitness = current_fitness < eval_scores_filtered

        #NOTE We need to check if two individuals have the same bd and different fitness!!
        #Adding both boolean arrays to perform an OR
        to_be_added = better_fitness + current_fitness_nan



        #We Apply the Mask to remove dead individuals
        to_be_added = jnp.where(dead,False,to_be_added)

        #Every Individual that is not valid will be assigned index 100000 because we cannot
cut our arrays. Jit needs to know the size of the array.
        #When adding, every individual will be clipped and sent to the same location
        mult_to_be_added = jnp.where(to_be_added,0,100000)

        bd_insertion = bd_indexes + mult_to_be_added

        # Adding individuals indivs to grid
        leaves = []
        for i, weight in enumerate(jax.tree_leaves(pop_p)):
            leaf = jax.tree_leaves(repertoire.archive)[i].at[bd_insertion].set(weight)
            leaves.append(leaf)
        # replacing grid with new leaves that have the updated weights
        new_archive = jax.tree_unflatten(jax.tree_structure(repertoire.archive), leaves)
        unraveled_indices = jnp.unravel_index(bd_insertion, repertoire.fitness.shape)
        # new_fitness = repertoire.fitness.at[jnp.unravel_index(bd_insertion,
repertoire.fitness.shape)].set(eval_scores,mode='clip')

        new_fitness = jnp.reshape(repertoire.fitness.ravel()
                        .at[bd_insertion].set(eval_scores),repertoire.fitness.shape)
        # print(repertoire.fitness)
        num_indivs = (jnp.where(~jnp.isnan(new_fitness),1,0)).sum()

        #returning this to make it jit friendly
        return repertoire.replace(archive = new_archive, fitness =
                         new_fitness, num_indivs =  num_indivs)

    @staticmethod
    # @jax.jit
    def add_to_archive_extended(repertoire, pop_p, bds, eval_scores,dead):

        normalized_bds = ((bds-repertoire.min)/(repertoire.max-repertoire.min))  #Normlalized
BD should be between zero and 1
        bd_cells = jit(jax.vmap(Repertoire.binning,
in_axes=(0,None),out_axes=0))(normalized_bds,repertoire.grid_shape)
        # print(bd_cells)
        bd_indexes = jnp.ravel_multi_index(bd_cells, repertoire.bd.shape,mode = 'clip')
        maximum_fitness = jax.ops.segment_max(eval_scores, bd_indexes,
                            num_segments=repertoire.fitness.ravel().shape[0])
        eval_scores_filtered = jnp.where(maximum_fitness.at[bd_indexes]
                            .get()==eval_scores,eval_scores,np.iinfo(np.int32).min)
        keep_eval_scores_filtered = jnp.where(maximum_fitness
                            .at[bd_indexes].get()==eval_scores,True,False)
        # Checking Conditions for fitness function
        current_fitness = repertoire.fitness.ravel().at[bd_indexes].get()
        # Checking if fitness function is nan or not, since nan means we do not have an
individual yet
        current_fitness_nan = jnp.logical_and(jnp.isnan(current_fitness),
                                              keep_eval_scores_filtered)
        # Checking if fitness that we have is better than the one we observed
        better_fitness = current_fitness < eval_scores_filtered

        #NOTE We need to check if two individuals have the same bd and different fitness!!
        #Adding both boolean arrays to perform an OR
        to_be_added = better_fitness + current_fitness_nan
        imp_values = eval_scores - jnp.where(current_fitness_nan
                                   , 0, current_fitness)

        #We Apply the Mask to remove dead individuals
        to_be_added = jnp.where(dead,False,to_be_added)
        # new individuals that were added to the archive
        new_indvds = jnp.sum(to_be_added.astype(jnp.int32))
        #Every Individual that is not valid will be assigned index 100000 because we cannot 
cut our arrays. Jit needs to know the size of the array.
        #When adding, every individual will be clipped and sent to the same location
        mult_to_be_added = jnp.where(to_be_added,0,100000)

        bd_insertion = bd_indexes + mult_to_be_added

        new_sols = jnp.where(jnp.logical_and(
                    current_fitness_nan,
                    to_be_added,
                ), AddStatus.NEW,  AddStatus.NOT_ADDED)
        better_sols = jnp.where(jnp.logical_and(
                    better_fitness,
                    to_be_added,
                ), AddStatus.IMPROVE_EXISTING,  AddStatus.NOT_ADDED)
        sols_statuses = new_sols + better_sols
        # Sols entries is an array showing the add to archive status and objective/evaluation 
score
        #  of each solution
        sols_entries = jnp.stack((sols_statuses, imp_values, jnp.arange(0,
                                len(sols_statuses), 1, dtype=int)),axis=-1)

        # Adding individuals indivs to grid
        leaves = []
        for i, weight in enumerate(jax.tree_leaves(pop_p)):
            leaf = jax.tree_leaves(repertoire.archive)[i].at[bd_insertion].set(weight)
            leaves.append(leaf)
        # replacing grid with new leaves that have the updated weights
        new_archive = jax.tree_unflatten(jax.tree_structure(repertoire.archive), leaves)
        unraveled_indices = jnp.unravel_index(bd_insertion, repertoire.fitness.shape)
        # new_fitness = repertoire.fitness.at[jnp.unravel_index(bd_insertion, 
repertoire.fitness.shape)].set(eval_scores,mode='clip')

        new_fitness = 
jnp.reshape(repertoire.fitness.ravel().at[bd_insertion].set(eval_scores),repertoire.fitness.sh
ape)
        # print(repertoire.fitness)
        num_indivs = (jnp.where(~jnp.isnan(new_fitness),1,0)).sum()
        #returning this to make it jit friendly
        return repertoire.replace(archive = new_archive, fitness =
                            new_fitness, num_indivs =  num_indivs), sols_entries, new_indvds
```

Code Snippet A.22: Implementation of N-Dimensional Grid Archive (grid_archive.py)

A.8 Experiment Utilities

A.8.1 Module for Instantiating QD and DQD Emitters

```
import functools
import jax.numpy as jnp
from training import qd_loop_simple as qd_simple
# Import QD Behavioral and Objective functions
from training.qd_functions import calc_rastrigin
from training.qd_functions import calc_bds_rastrigin, calc_bds_rastrigin_simple
from training.qd_functions import calc_grasp_bds, calc_grasp_objs
# Import different emitters
from training.emitters._optimizing_emitter import OptimizingEmitter
import training.emitters._optimizing_emitter as opt_emitter
from training.emitters._improvement_emitter import ImprovementEmitter
import training.emitters._improvement_emitter as impr_emitter
from training.emitters._random_direction_emitter import RandomDirectionEmitter
import training.emitters._random_direction_emitter as rand_dir_emitter
from training.emitters._gradient_improvement_emitter import GradientImprovementEmitter
import training.emitters._gradient_improvement_emitter as grad_impr_emitter
from training.emitters._gaussian_emitter import GaussianEmitter
from training.emitters._iso_line_emitter import IsoLineEmitter
from training.emitters._gradient_emitter import GradientEmitter
import training.emitters._gradient_emitter as grad_emitter
from training.opt._cma_es import WeightRules

QD_EMITTERS_SUPPORTED = ["map_elites", "map_elites_line", "cma_me_imp",
                        "cma_me_rd", "cma_me_opt"]
DQD_EMITTERS_SUPPORTED = ["og_map_elites_iso", "og_map_elites_line",
                          "omg_mega_iso", "omg_mega_line",
                          "omg_mega",  "cma_mega", "cma_mega_adam"]
ALL_EMITTERS_SUPPORTED = QD_EMITTERS_SUPPORTED + DQD_EMITTERS_SUPPORTED

ENVS = ['rastrigin-distorted', 'rastrigin', 'arm']

def get_env_info(args):
  env_name = args.env_name
  sols_dim = args.sols_dim
  min_bound = 0
  max_bound = 1
  archive_bounds = jnp.array([[min_bound, max_bound] for _ in range(2)])
  bounds = None
  # Environment selection
  if env_name == "rastrigin-distorted":
    max_bound = args.sols_dim / 2 * 5.12
    min_bound = -max_bound
    archive_bounds = jnp.array([[min_bound, max_bound] for _ in range(2)])
    # bounds = [[-5.12, 5.12] for _ in range(args.sols_dim)]
    bounds = None
    return archive_bounds, bounds, calc_rastrigin, calc_bds_rastrigin
  elif env_name == "rastrigin":
    max_bound = 1
    min_bound = 0
    archive_bounds = jnp.array([[min_bound, max_bound] for _ in range(2)])
    # bounds = [[min_bound, max_bound] for _ in range(args.sols_dim)]
    # bounds = None
    return archive_bounds, bounds, calc_rastrigin, calc_bds_rastrigin_simple
  elif env_name == "arm":
    link_lengths = jnp.ones(sols_dim)
    max_bound = sols_dim
    min_bound = -max_bound
    archive_bounds = jnp.array([[min_bound, max_bound] for _ in range(2)])
    # bounds = [[min_bound, max_bound] for _ in range(args.sols_dim)]
    # bounds = None
    calc_grasp_objs_p = functools.partial(calc_grasp_objs,
                                          link_lengths=link_lengths,
                                          calc_jacobians=True
                                          )
    calc_grasp_bds_p = functools.partial(calc_grasp_bds,
                                          link_lengths=link_lengths,
                                          calc_jacobians=True
                                          )
    return archive_bounds, bounds, calc_grasp_objs_p, calc_grasp_bds_p
  return archive_bounds, bounds, None, None

def get_emitter(args, qd_params, key):

  archive_bounds, bounds, _, _= get_env_info(args)

  emitter_name = args.emitter
  sigma0 = args.sigma0
  sigma1 = args.sigma1
  sigma_g = args.sigma_g
  batch_size = args.batch_size
  selection_rule = args.selection_rule
  restart_rule = args.restart_rule
  weight_rule = args.weight_rule
```

```
sols_dim = args.sols_dim
norm_grad = args.norm_grad
stepsize = args.stepsize
initial_sol = jnp.zeros(sols_dim)
behavior_dim = len(archive_bounds)
use_dqd = 0
if emitter_name == "cma_me_opt":
  sigma0 = sigma0 if sigma0 else 0.5
  emitter, static_settings = OptimizingEmitter.create(
                x0=initial_sol,
                sigma0=sigma0,
                selection_rule=opt_emitter.SelectionRules.FILTER
                                    if selection_rule is None else selection_rule,
                restart_rule=opt_emitter.RestartRules.NO_IMPROVEMENT
                                    if restart_rule is None else restart_rule,
                weight_rule=WeightRules.TRUNCATION
                                    if weight_rule is None else weight_rule,
                bounds=None,
                batch_size=batch_size)
elif emitter_name == "cma_me_imp":
  sigma0 = sigma0 if sigma0 else 0.5
  emitter, static_settings = ImprovementEmitter.create(
                x0=initial_sol,
                sigma0=sigma0,
                selection_rule=impr_emitter.SelectionRules.FILTER
                                    if selection_rule is None else selection_rule,
                restart_rule=impr_emitter.RestartRules.NO_IMPROVEMENT
                                    if restart_rule is None else restart_rule,
                weight_rule=WeightRules.TRUNCATION
                                    if weight_rule is None else weight_rule,
                bounds=None,
                batch_size=batch_size)
elif emitter_name == "cma_me_rd":
  sigma0=sigma0 if sigma0 else 0.5
  emitter, static_settings = RandomDirectionEmitter.create(
                x0=initial_sol,
                sigma0=sigma0,
                archive_bounds=archive_bounds,
                key=key,
                selection_rule=rand_dir_emitter.SelectionRules.FILTER
                                    if selection_rule is None else selection_rule,
                restart_rule=rand_dir_emitter.RestartRules.NO_IMPROVEMENT
                                    if restart_rule is None else restart_rule,
                weight_rule=WeightRules.TRUNCATION
                                    if weight_rule is None else weight_rule,
                bounds=None,
                batch_size=batch_size)
```

```
elif emitter_name == "cma_mega":
  sigma_g=sigma_g if sigma_g else 10.0
  stepsize=stepsize if stepsize else 1.0
  emitter, static_settings = GradientImprovementEmitter.create(
                x0=initial_sol,
                behavior_dim = behavior_dim,
                sigma_g=sigma_g,
                stepsize=stepsize,
                selection_rule=grad_impr_emitter.SelectionRules.MU
                                    if selection_rule is None else selection_rule,
                restart_rule=grad_impr_emitter.RestartRules.NO_IMPROVEMENT
                                    if restart_rule is None else restart_rule,
                weight_rule=WeightRules.TRUNCATION
                                    if weight_rule is None else weight_rule,
                gradient_optimizer = grad_impr_emitter.GradientOptimizers.GRADIENT_ASCENT,
                normalize_gradients=norm_grad if norm_grad else 1,
                bounds=None,
                batch_size=batch_size)
  use_dqd = 1
elif emitter_name == "cma_mega_adam":
  sigma_g=sigma_g if sigma_g else 10.0
  stepsize=stepsize if stepsize else 0.002
  emitter, static_settings = GradientImprovementEmitter.create(
                x0=initial_sol,
                behavior_dim = behavior_dim,
                sigma_g=sigma_g,
                stepsize=stepsize,
                selection_rule=grad_impr_emitter.SelectionRules.MU
                                    if selection_rule is None else selection_rule,
                restart_rule=grad_impr_emitter.RestartRules.NO_IMPROVEMENT
                                    if restart_rule is None else restart_rule,
                weight_rule=WeightRules.TRUNCATION
                                    if weight_rule is None else weight_rule,
                gradient_optimizer = grad_impr_emitter.GradientOptimizers.ADAM,
                normalize_gradients=norm_grad if norm_grad else 1,
                bounds=None,
                batch_size=batch_size)
  use_dqd = 1
elif emitter_name == "map_elites":
  sigma0=sigma0 if sigma0 else 0.5
  emitter, static_settings = GaussianEmitter.create(
                x0=initial_sol,
                sigma0=sigma0,
                batch_size=batch_size,
                bounds=bounds)
elif emitter_name == "map_elites_line":
  sigma0=sigma0 if sigma0 else 0.5
```

```
    sigma1=sigma1 if sigma1 else 0.5
    emitter, static_settings = IsoLineEmitter.create(
                  x0=initial_sol,
                  iso_sigma=sigma0,
                  line_sigma=sigma1,
                  batch_size=batch_size,
                  bounds=bounds)
  elif emitter_name == "og_map_elites_iso":
    sigma0=sigma0 if sigma0 else 0.5
    sigma_g=sigma_g if sigma_g else 0.5
    sigma1=sigma1 if sigma1 else 0.5
    emitter, static_settings = GradientEmitter.create(
                  x0=initial_sol,
                  sigma0=sigma0,
                  sigma_g=sigma_g,
                  line_sigma=sigma1,
                  behavior_dim = behavior_dim,
                  measure_gradients = 0,
                  normalize_gradients=norm_grad if norm_grad else 1,
                  operator_type = grad_emitter.OperatorTypes.ISOTROPIC,
                  bounds=bounds,
                  batch_size=batch_size)
    use_dqd = 1
  elif emitter_name == "og_map_elites_line":
    sigma0=sigma0 if sigma0 else 0.5
    sigma_g=sigma_g if sigma_g else 0.5
    sigma1=sigma1 if sigma1 else 0.5
    emitter, static_settings = GradientEmitter.create(
                  x0=initial_sol,
                  sigma0=sigma0,
                  sigma_g=sigma_g,
                  line_sigma=sigma1,
                  behavior_dim = behavior_dim,
                  measure_gradients = 0,
                  normalize_gradients=norm_grad if norm_grad else 1,
                  operator_type = grad_emitter.OperatorTypes.ISO_LINE_DD,
                  bounds=bounds,
                  batch_size=batch_size)
    use_dqd = 1
  elif emitter_name == "omg_mega":
    sigma0=0
    sigma_g=sigma_g if sigma_g else 0.5
    sigma1=0
    emitter, static_settings = GradientEmitter.create(
                  x0=initial_sol,
                  sigma0=sigma0,
                  sigma_g=sigma_g,
                  line_sigma=sigma1,
                  behavior_dim = behavior_dim,
                  measure_gradients = 0,
                  normalize_gradients=norm_grad if norm_grad else 1,
                  operator_type = grad_emitter.OperatorTypes.ISOTROPIC,
                  bounds=bounds,
                  batch_size=batch_size)
    use_dqd = 1
  elif emitter_name == "omg_mega_iso":
    sigma0=sigma0 if sigma0 else 0.5
    sigma_g=sigma_g if sigma_g else 0.5
    sigma1=sigma1 if sigma1 else 0.5
    emitter, static_settings = GradientEmitter.create(
                  x0=initial_sol,
                  sigma0=sigma0,
                  sigma_g=sigma_g,
                  line_sigma=sigma1,
                  behavior_dim = behavior_dim,
                  measure_gradients = 0,
                  normalize_gradients=norm_grad if norm_grad else 1,
                  operator_type = grad_emitter.OperatorTypes.ISOTROPIC,
                  bounds=bounds,
                  batch_size=batch_size)
    use_dqd = 1
  elif emitter_name == "omg_mega_line":
    sigma0=sigma0 if sigma0 else 0.5
    sigma_g=sigma_g if sigma_g else 0.5
    sigma1=sigma1 if sigma1 else 0.5
    emitter, static_settings = GradientEmitter.create(
                  x0=initial_sol,
                  sigma0=sigma0,
                  sigma_g=sigma_g,
                  line_sigma=sigma1,
                  behavior_dim = behavior_dim,
                  measure_gradients = 0,
                  normalize_gradients=norm_grad if norm_grad else 1,
                  operator_type = grad_emitter.OperatorTypes.ISO_LINE_DD,
                  bounds=bounds,
                  batch_size=batch_size)
    use_dqd = 1
  qd_params['emitter'] = emitter_name
  qd_params['use_dqd'] = use_dqd
  if sigma0 is not None:
    qd_params['s0'] = sigma0
  if sigma1 is not None:
    qd_params['s1'] = sigma1
  if sigma_g is not None:
    qd_params['sg'] = sigma_g
  if stepsize is not None:
    qd_params['stepsize'] = stepsize
  return emitter, static_settings, qd_params

def get_num_epochs_and_evaluations(num_epochs, num_evaluations,
                      population_size, use_dqd=0):
  if num_epochs is not None:
    num_evaluations = (num_epochs * population_size
            * (2 if use_dqd >= 1 else 1))
    return num_epochs, num_evaluations
  elif num_evaluations is not None:
    num_epochs = ((num_evaluations // population_size)
                    // (2 if use_dqd >= 1 else 1) + 1)
    return num_epochs, num_evaluations
  else:
    raise ValueError("One of the 2 following variables should "
          +"be defined: num_epochs or num_evaluations")
```

Code Snippet A.23: Module for instantiating QD and DQD emitters (run_qd_utils.py)

A.8.2 Main QD and DQD Module for executing QD and DQD Algorithms

```
"""
Quality-Diversity Evolution Strategy training.
"""
# TO USE MULTIPLE CPUs
# import os
# os.environ['XLA_FLAGS'] = '--xla_force_host_platform_device_count=16'

import functools
import time
from typing import Any, Callable, Dict, Optional

import jax
from jax import lax
import jax.numpy as jnp
from absl import logging

from qdax.qd_utils import grid_archive
from qdax.stats.metrics import Metrics
from qdax.stats.saving_loading_utils import make_results_folder
from qdax.stats.timings import Timings
from qdax.stats.training_state_simple import SimpleTrainingState
from qdax.tasks import BraxTask
from qdax.training.configuration import Configuration
```

```
# print('Jax devices: ',jax.devices())

Array = Any

def _update_metrics(log_frequency: int,
                    metrics: Metrics,
                    epoch: int,
                    repertoire: grid_archive.Repertoire):

  index = jnp.ceil(epoch / log_frequency).astype(int)
  scores = metrics.scores.at[index, 0].set(epoch)
  scores = scores.at[index, 1].set(repertoire.num_indivs)
  scores = scores.at[index, 2].set(jnp.nanmax(repertoire.fitness))
  scores = scores.at[index, 3].set(jnp.nansum(repertoire.fitness))
  archives = metrics.archives.at[index, :, :].set(repertoire.fitness)
  return Metrics(scores=scores, archives=archives)

def _eval_and_add_simple(local_devices_to_use,
                  objective_fn,
                  bds_fn,
                  population_size,
                  add_to_archive_fn,
                  training_state: SimpleTrainingState,
                  params,
                  key):

  # params = training_state.state

  pparams_device = jnp.reshape(params, [local_devices_to_use, -1] + list(params.shape[1:]))

  # run evaluations - evaluate params
  eval_start_t = time.time()
  prun_obj_eval = jax.pmap(objective_fn, in_axes=(0,))
  prun_bd_eval = jax.pmap(bds_fn, in_axes=(0,))
  objs, _ = prun_obj_eval(pparams_device)
  bds, _ = prun_bd_eval(pparams_device)
  logging.debug("Time Evaluation: %s ", time.time() - eval_start_t)

  dead = jnp.zeros(population_size)
  objs_flat = jnp.reshape(objs, (population_size, -1)).ravel()
  bds_flat = jnp.reshape(bds,(-1,bds.shape[-1]))

  # Update archive
  update_archive_start_t = time.time()
  repertoire = add_to_archive_fn(repertoire = training_state.repertoire,
                                 pop_p = params,
                                 bds = bds_flat,
                                  eval_scores = objs_flat,
                                  dead = dead)
  logging.debug("Time took for Adding: %s ", time.time() - update_archive_start_t)

  return repertoire, params

def _eval_and_add(local_devices_to_use,
                  objective_fn,
                  bds_fn,
                  population_size,
                  emitter_static_settings,
                  emitter,
                  training_state: SimpleTrainingState,
                  params,
                  key):
  key, key_tell= jax.random.split(key, 2)
  # params = training_state.state

  pparams_device = jnp.reshape(params, [local_devices_to_use, -1] + list(params.shape[1:]))

  # run evaluations - evaluate params
  eval_start_t = time.time()
  prun_obj_eval = jax.pmap(objective_fn, in_axes=(0,))
  prun_bd_eval = jax.pmap(bds_fn, in_axes=(0,))
  objs, _ = prun_obj_eval(pparams_device)
  bds, _ = prun_bd_eval(pparams_device)
  logging.debug("Time Evaluation: %s ", time.time() - eval_start_t)

  dead = jnp.zeros(population_size)
  objs_flat = jnp.reshape(objs, (population_size, -1)).ravel()
  bds_flat = jnp.reshape(bds,(-1,bds.shape[-1]))

  # Update archive
  update_archive_start_t = time.time()
  emitter, repertoire = emitter.tell(emitter_static_settings, emitter, params, objs_flat,
bds_flat, dead,
                                        training_state.repertoire, key_tell)
  logging.debug("Time took for Adding: %s ", time.time() - update_archive_start_t)

  return emitter, repertoire, params

def _eval_and_add_grad(local_devices_to_use,
                  objective_fn,
                  bds_fn,
                  population_size,
                  emitter_static_settings,
                  emitter,
                  training_state: SimpleTrainingState,
                  params,
                  key):
  key, key_tell= jax.random.split(key, 2)
  # params = training_state.state

  pparams_device = jnp.reshape(params, [local_devices_to_use, -1] + list(params.shape[1:]))

  # run evaluations - evaluate params
  eval_start_t = time.time()
  prun_obj_eval = jax.pmap(objective_fn, in_axes=(0,))
  prun_bd_eval = jax.pmap(bds_fn, in_axes=(0,))
  objs, jac_objs = prun_obj_eval(pparams_device)
  bds, jac_bds = prun_bd_eval(pparams_device)
  # Prepare combined objective and behavioral jacobian matrix
  jac_objs = jnp.reshape(jac_objs,[-1] + list(jac_objs.shape[2:]))
  jac_bds = jnp.reshape(jac_bds,[-1] + list(jac_bds.shape[2:]))
  jac_objs = jnp.expand_dims(jac_objs, axis=1)
  jacobian = jnp.concatenate((jac_objs, jac_bds), axis=1)
  logging.debug("Time Evaluation: %s ", time.time() - eval_start_t)

  dead = jnp.zeros(population_size)
  objs_flat = jnp.reshape(objs, (population_size, -1)).ravel()
  bds_flat = jnp.reshape(bds,(-1,bds.shape[-1]))

  # Update archive
  update_archive_start_t = time.time()
  emitter, repertoire = emitter.tell_jacobian(emitter_static_settings, emitter, params,
objs_flat, bds_flat, dead,
                                       training_state.repertoire, key_tell, jacobian)
  logging.debug("Time took for Adding: %s ", time.time() - update_archive_start_t)

  return emitter, repertoire, params

def _init_phase(population_size,
                sols_dim,
                eval_and_add_fn,
                update_metrics_fn,
                training_state: SimpleTrainingState):

  logging.info(" Initialisation with random parameters")
  init_start_t = time.time()
  key, key_params, key_eval = jax.random.split(training_state.key, 3)
  params = jax.random.normal(key_params,shape=(population_size,sols_dim))
  logging.debug("Time Random Init: %s ", time.time() - init_start_t)

  repertoire, state = eval_and_add_fn(training_state, params, key_eval)
  metrics = update_metrics_fn(training_state.metrics, 0, repertoire)

  return SimpleTrainingState(key=key, repertoire=repertoire, metrics=metrics, state=state)

def _es_one_epoch_grad(eval_and_add_fn,
                  eval_and_add_grad_fn,
                  update_metrics_fn,
                  emitter_static_settings,
                  emitter,
                  epoch: int,
                  training_state: SimpleTrainingState
                  ):
  epoch_start_t = time.time()

  # generate keys for emmitter and evaluations
  key_1, key_2, key_emitter_1, key_es_eval_1, key_emitter_2, key_es_eval_2 =
jax.random.split(training_state.key, 6)


  # EMITTER: SELECTION AND MUTATION - GRADIENT ESTIMATION #
  sel_mut_start_t = time.time()
  emitter, params =  emitter.ask_grad_estimate(emitter_static_settings, emitter,
training_state.repertoire, key_emitter_1)
  logging.debug("Time Selection and Mutation: %s ", time.time() - sel_mut_start_t)

  # EVALUATION #
  emitter, repertoire, state = eval_and_add_grad_fn(emitter, training_state, params,
key_es_eval_1)

  training_state = training_state.replace(key = key_1, repertoire = repertoire, state = state)

  # EMITTER: SELECTION AND MUTATION #
  sel_mut_start_t = time.time()
  emitter, params =  emitter.ask(emitter_static_settings, emitter, training_state.repertoire,
key_emitter_2)
  logging.debug("Time Selection and Mutation: %s ", time.time() - sel_mut_start_t)

  # EVALUATION #
  emitter, repertoire, state = eval_and_add_fn(emitter, training_state, params, key_es_eval_2)
  logging.debug("ES Epoch Time: %s",time.time()-epoch_start_t)

  # UPDATE METRICS #
  #metrics = jax.lax.cond((epoch+1)%log_frequency == 0 , update_metrics, lambda x:x[0],
(training_state.metrics, epoch//log_frequency+1, repertoire))
  logging.debug("ES Start metrics:")
  metrics = update_metrics_fn(training_state.metrics, epoch, repertoire)
  logging.debug("ES Metrics Time: %s",time.time()-epoch_start_t)
  return SimpleTrainingState(key = key_2, repertoire = repertoire, metrics = metrics, state =
state), emitter

def _es_one_epoch(eval_and_add_fn,
                  update_metrics_fn,
                  emitter_static_settings,
                  emitter,
                  epoch: int,
                  training_state: SimpleTrainingState
                  ):
  epoch_start_t = time.time()

  # generate keys for emmitter and evaluations
  key, key_emitter, key_es_eval = jax.random.split(training_state.key, 3)

  # EMITTER: SELECTION AND MUTATION #
  sel_mut_start_t = time.time()
  emitter, params =  emitter.ask(emitter_static_settings, emitter, training_state.repertoire,
key_emitter)
  logging.debug("Time Selection and Mutation: %s ", time.time() - sel_mut_start_t)

  # EVALUATION #
  emitter, repertoire, state = eval_and_add_fn(emitter, training_state, params, key_es_eval)
  logging.debug("ES Epoch Time: %s",time.time()-epoch_start_t)

  # UPDATE METRICS #
  #metrics = jax.lax.cond((epoch+1)%log_frequency == 0 , update_metrics, lambda x:x[0],
(training_state.metrics, epoch//log_frequency+1, repertoire))
  logging.debug("ES Start metrics:")
  metrics = update_metrics_fn(training_state.metrics, epoch, repertoire)
  logging.debug("ES Metrics Time: %s",time.time()-epoch_start_t)

  return SimpleTrainingState(key = key, repertoire = repertoire, metrics = metrics, state =
state), emitter

def train(
    configuration: Configuration,
    emitter,
    emitter_static_settings,
    objective_fn,
    bds_fn,
    experiment_name: str,
    result_path: str,
    key = None,
    progress_fn: Optional[Callable[[int, Dict[str, Any]], None]] = None,
    save_results=True
):
  # Extract QD Parameters
  qd_params = configuration.qd_params
  use_dqd = int(str(qd_params['use_dqd']))
  sols_dim = int(qd_params["sols_dim"])
  # Extract Execution Condifurations
  num_epochs = configuration.num_epochs
  episode_length = configuration.episode_length
  action_repeat = configuration.action_repeat
  max_devices_per_host = configuration.max_devices_per_host
  population_size = configuration.population_size
  seed = configuration.seed
  log_frequency = configuration.log_frequency

  timings = Timings(log_frequency = log_frequency, num_epochs = num_epochs)
  start_t = time.time()
  framework_t = time.time()
  # INIT FRAMEWORK #
  # Initialization of env parameters and devices #
  num_envs = population_size
  process_count = jax.process_count()
  process_id = jax.process_index()
  local_device_count = jax.local_device_count()
  local_devices_to_use = local_device_count

  if max_devices_per_host:
    local_devices_to_use = min(local_devices_to_use, max_devices_per_host)
  logging.info(
      'Device count: %d, process count: %d (id %d), local device count: %d, '
      'devices to be used count: %d',
      jax.device_count(), process_count, process_id, local_device_count,
      local_devices_to_use)
  logging.info("Local devices to use: %d ", local_devices_to_use)
  logging.info("Batch size on 1 device for env: %d", num_envs // local_devices_to_use //
process_count)

  # Initialize keys for random processes - need to handle for jax.
  if key is None:
    key = jax.random.PRNGKey(seed)
  key, key_params, key_env = jax.random.split(key, 3) #key for main training state, policy
model init and train environment

  timings.init_framework = time.time() - framework_t

  # Core training environment
  env_t = time.time() # NOTE: this timing doesnt work anymore at the moment - environment is
initialized outside the train_fn
```

```
  # Calculate initial/first states, one per local acceleration device
  key_envs = jax.random.split(key_env, local_devices_to_use)
  # The inital states are just one dimensional ndarrays
  first_state = jax.vmap(lambda x:
jax.random.normal(x,shape=(population_size,sols_dim)),0,0)(key_envs)

  logging.info("Initialize env time: %s ", time.time() - env_t)
  timings.init_env = time.time() - env_t

  # Initialize model/policy #
  policy_t = time.time()

  # Initialize archive
  min_bd = configuration.min_bd
  max_bd = configuration.max_bd
  grid_shape = configuration.grid_shape
  repertoire = grid_archive.Repertoire.create(jax.random.normal(key_params,shape=(sols_dim,)),
min=min_bd, max=max_bd, grid_shape=grid_shape)

  # ============= METRICS UPDATE FN ============ #
  update_metrics_fn = functools.partial(
    _update_metrics,
    log_frequency,
  )

  # ============ ENVIRONMENT EVAL FUNCTIONS AND ARCHIVE ADDITION ============#
  eval_and_add_simple_fn = jax.jit(functools.partial(
    _eval_and_add_simple,
    local_devices_to_use,
    objective_fn,
    bds_fn,
    population_size,
    jax.jit(repertoire.add_to_archive),
  ))
  # ============ ENVIRONMENT EVAL FUNCTIONS AND ARCHIVE ADDITION ============#
  # Based on the parameter use_dqd choose whether to use gradients to extend solutions

  # Simple evaluation and addition of solutions to the archive
  eval_and_add_fn = jax.jit(functools.partial(
    _eval_and_add,
    local_devices_to_use,
    objective_fn,
    bds_fn,
    population_size,
    emitter_static_settings
  ))
```

```
if use_dqd > 0:
  # Evaluation and addition of solutions to the archive using gradients
  eval_and_add_grad_fn = jax.jit(functools.partial(
    _eval_and_add_grad,
    local_devices_to_use,
    objective_fn,
    bds_fn,
    population_size,
    emitter_static_settings
  ))

# ========== INIT REPERTOIRE BY RANDOM POLICIES =============== #
init_phase_fn = functools.partial(
  _init_phase,
  population_size,
  sols_dim,
  eval_and_add_simple_fn,
  update_metrics_fn,
)

# ========== ONE GENERATION/EPOCH OF ALGORITHM FN ===============#
es_one_epoch_fn = None
if use_dqd == 0:
  es_one_epoch_fn = jax.jit(functools.partial(
    _es_one_epoch,
    eval_and_add_fn,
    update_metrics_fn,
    emitter_static_settings,
  ))
else:
  es_one_epoch_fn = jax.jit(functools.partial(
    _es_one_epoch_grad,
    eval_and_add_fn,
    eval_and_add_grad_fn,
    update_metrics_fn,
    emitter_static_settings,
  ))

key_debug = jax.random.PRNGKey(seed + 777)
timings.init_policies = time.time() - policy_t

# ================= MAIN QD ALGORITHM LOOP =================== #
logging.info("######### START QD ALGORITHM ############")
qd_t = time.time()

# INIT TRAINING STATE #
training_state = SimpleTrainingState(
    key=key,
    repertoire = repertoire,
    metrics = Metrics.create(log_frequency =
        log_frequency, num_epochs = num_epochs,
        grid_shape = repertoire.grid_shape),
    state = first_state
  )
  # INIT REPERTOIRE #
  training_state = init_phase_fn(training_state)
  timings.init_QD = time.time() - qd_t

  logging.info('Starting Main QD Loop')

  for i in range (1, num_epochs+1):
    epoch_t = time.time()
    # print('EPOCH {}'.format(i))
    training_state, emitter = es_one_epoch_fn(emitter, i, training_state)
    # log timings
    epoch_duration = time.time() - epoch_t
    logging.debug("epoch loop Time: %s ", epoch_duration)
    epoch_runtime = time.time() - start_t
    # Average epoch time in seconds
    timings.avg_epoch = ((i-1) * timings.avg_epoch + (epoch_duration)) / float(i)
    use_dqd_weight = 1
    if use_dqd > 0:
      use_dqd_weight = 2
    # Average number of evaluations per second
    # @TO-BE-INVESTIGATED in DQD, do we consider two evaluations per normal evaluation or
twice?
    # Because we have the variation of the normal algorithm and the variation with the
gradients
    # So far, we do consider it as two evaluations per single iteration (i.e. when batch size
is 1)
    timings.avg_eval_per_sec = ((i-1) * timings.avg_eval_per_sec + (use_dqd_weight *
population_size)/epoch_duration ) / float(i)
    # Find the index of the record based on the current iteration/epoch and the
    # frequency of keeping records/logging
    index = jnp.ceil(i / log_frequency).astype(int)
    timings.epoch_runtime = timings.epoch_runtime.at[index, 0].set(epoch_runtime)
    #print("Index: ",index, epoch_runtime)

  timings.full_training = time.time() - start_t
  logging.info(timings)
  #training_state.repertoire.fitness.block_until_ready()
  logging.debug("Total main loop Time: %s ", time.time() - start_t)
  logging.info("Total main loop Time: %s ", time.time() - start_t)
  logging.info("Repertoire size: %d ", training_state.repertoire.num_indivs)
  logging.info("Scores [epoch, num_indivs, best fitness, QD score]:\n %s ",
training_state.metrics.scores)

  # ===== SAVE RESULTS AND CONFIGS ==== #
  if save_results:
    res_dir = make_results_folder(result_path, experiment_name, configuration)
    logging.info("Saving results in %s ", res_dir)
    configuration.save_to_json(folder=res_dir)
    timings.save(folder=res_dir)
    training_state.save(folder=res_dir)
    training_state.metrics.save(folder=res_dir)

  if(progress_fn):
    metrics = dict(
          **dict({
             'train/generation': num_epochs+1,
             'repertoire/repertoire_size': training_state.repertoire.num_indivs,
          }))
    progress_fn(metrics, training_state.repertoire)

  return training_state
```

Code Snippet A.24: Main Module for running the QD and DQD Algorithms on GPUs (qd_loop_simple.py)

A.8.3 Module for scheduling the execution of QD and DQD Algorithms

```
import argparse
import os
from absl import logging,flags
from run_qd_args_parsers import add_args_for_qd_run, check_validity_args

QD_EMITTERS_SUPPORTED = ["map_elites", "map_elites_line", "cma_me_imp",
                        "cma_me_rd", "cma_me_opt"]
DQD_EMITTERS_SUPPORTED = ["og_map_elites_iso", "og_map_elites_line",
                          "omg_mega_iso", "omg_mega_line","omg_mega",
                          "cma_mega", "cma_mega_adam"]
ALL_EMITTERS_SUPPORTED = QD_EMITTERS_SUPPORTED + DQD_EMITTERS_SUPPORTED

ENV_SUPPORTED = ['rastrigin-distorted', 'rastrigin', 'arm']

# from jax.config import config
# config.update("jax_enable_x64", True)
# import sys
# jax.numpy.set_printoptions(threshold=sys.maxsize)
args = None

def process_args():
    parser = argparse.ArgumentParser(formatter_class=argparse.RawTextHelpFormatter)
    add_args_for_qd_run(parser, ALL_EMITTERS_SUPPORTED, env_supported=ENV_SUPPORTED)
    parsed_arguments = parser.parse_args()
    check_validity_args(parser, parsed_arguments)
    return parsed_arguments

# Set the environmental variables for Hardware and
#  Jax before importing Jax Library
if __name__ == "__main__":
  try:
    args = process_args()
    # Change environmental variables based on
    # whether to use only cpu or not
    if not args.use_cpu != 0:
      os.environ["CUDA_VISIBLE_DEVICES"] = args.devices
      # Just uncomment the line below if you want to enforce it
      # os.environ["XLA_FLAGS"] = "--
xla_force_host_platform_device_count="+str(args.device_count)
    else:
      os.environ["JAX_PLATFORM_NAME"] = "cpu"
      # Just uncomment the line below if you want to enforce it
      # os.environ["XLA_FLAGS"] = "--
xla_force_host_platform_device_count="+str(args.device_count)

  except Exception as e:
    logging.fatal(e, exc_info=True)

# Then import ll the rest necessary libraries
import jax
from jax.lib import xla_bridge
from training.configuration import Configuration
from training import qd_loop_simple as qd_simple
from run_qd_utils import get_num_epochs_and_evaluations, get_emitter, get_env_info
import datetime
import random
import sys

def main(parsed_arguments):
  if parsed_arguments.log_folder:
    if not os.path.exists(parsed_arguments.log_folder):
      raise FileNotFoundError("Folder {} not found.".format(parsed_arguments.log_folder))
    else:
      datetime_now = datetime.datetime.now().strftime("%Y-%m-%d_T_%H-%M-%S")
    logging.get_absl_handler().use_absl_log_file('log_{0}.log'.format(datetime_now),
                        parsed_arguments.log_folder)
    flags.FLAGS.mark_as_parsed()
  results_saving_folder = parsed_arguments.directory
  # logging.get_absl_handler().setFormatter(None)

  if not os.path.exists(results_saving_folder):
    raise FileNotFoundError(f"Folder {results_saving_folder} not found.")

  levels = {'fatal': logging.FATAL,
            'error': logging.ERROR,
            'warning': logging.WARNING,
            'info': logging.INFO,
            'debug': logging.DEBUG}

  logging.set_verbosity(levels[parsed_arguments.log_level])
  population_size = parsed_arguments.batch_size
  local_device_count = jax.local_device_count()
  local_devices_to_use = local_device_count
  process_count = jax.process_count()

  logging.info(f"Hardware Details:\n"
               f"\t Platform: {xla_bridge.get_backend().platform}\n"
               f"\t Local_device_count: {local_device_count}\n"
               f"\t process_count: {process_count}\n")

  seed = ( random.randrange(sys.maxsize) if parsed_arguments.seed
                              is None else parsed_arguments.seed
          )
  key = jax.random.PRNGKey(seed)
  key, key_emitter_init = jax.random.split(key, 2)

  qd_params = dict()
  emitter, static_settings, qd_params = get_emitter(parsed_arguments, qd_params,
key_emitter_init)

  num_epochs, num_evaluations =
get_num_epochs_and_evaluations(num_epochs=parsed_arguments.num_epochs,
                                                               num_evaluations=parsed_argument
s.num_evaluations,
                                                               population_size=parsed_argument
s.batch_size,
                                                               use_dqd=qd_params['use_dqd'])

  qd_params['device'] = xla_bridge.get_backend().platform # 'cpu' if args.use_cpu == 1 else
'gpu'
  qd_params['sols_dim'] = parsed_arguments.sols_dim
  logging.info(f"Options:\n"
               f"\t Log_level:{parsed_arguments.log_level}\n"
               f"\t Seed: {seed}\n"
               f"\t Batch_size:{parsed_arguments.batch_size}\n"
               f"\t Num_epochs:{num_epochs}\n"
               f"\t Num_evaluations:{num_evaluations}\n"
               f"\t Episode_length:{parsed_arguments.episode_length}\n"
               f"\t Log_frequency:{parsed_arguments.log_frequency}\n")

  archive_bounds, _, calc_objs, calc_bds = get_env_info(parsed_arguments)

  configuration = Configuration(args.env_name,
                              num_epochs,
                              parsed_arguments.episode_length,
                              action_repeat=1,
                              population_size=parsed_arguments.batch_size,
                              seed=seed,
                              log_frequency=parsed_arguments.log_frequency,
                              qd_params=qd_params,
                              min_bd=float(archive_bounds[0,0]), #
float(args.min_max_bd[0]),
                              max_bd=float(archive_bounds[0,1]), #
float(args.min_max_bd[1]),
                              grid_shape=tuple(parsed_arguments.grid_shape),
                              max_devices_per_host=None,
                              )

  qd_simple.train(
    emitter=emitter,
    emitter_static_settings = static_settings,
    objective_fn=calc_objs,
    bds_fn=calc_bds,
    configuration=configuration,
    progress_fn=None,
    experiment_name=parsed_arguments.exp_name,
    key=key,
    result_path=results_saving_folder,
    save_results=True
  )

if __name__ == "__main__":
  try:
    main(args)
  except Exception as e:
    logging.fatal(e, exc_info=True)
```

Code Snippet A.25: Module for scheduling the execution of QD and DQD Algorithms (run_qd_loop_simple.py)

A.8.4 Module for providing command line input parsers for the different functionalities

```python
import os
import argparse

def add_args_for_qd_run(parser: argparse.ArgumentParser, emitters_supported,
env_supported=['rastrigin']):
    parser.add_argument('--seed', default=None, type=int)
    # Options for algorithm execution space
    parser.add_argument('--episode_length', default=100, type=int)
    parser.add_argument('--num_evaluations', default=None, type=int)
    parser.add_argument('--batch_size', default=2048, type=int)
    parser.add_argument('--num_epochs', default=None, type=int)
    parser.add_argument('--sols_dim', default=10, type=int)
    # Emitter Options
    parser.add_argument('--emitter',
                        default=emitters_supported[0],
                        choices=emitters_supported)
    # Covariance Matrix Adaptation Options - CMA (Used only if the emitter is of CMA type)
    parser.add_argument('--selection_rule', default=None, type=int)
    parser.add_argument('--restart_rule', default=None, type=int)
    parser.add_argument('--weight_rule', default=None, type=int)
    parser.add_argument('--stepsize', default=None, type=float)
    parser.add_argument('--norm_grad', default=None, type=int)
    # Algorithm Grid shape of archive
    parser.add_argument('--grid_shape', nargs='+', type=int, required=True)
    # Standard deviations for algorithm
    parser.add_argument('--sigma0', default=None, type=float)
    parser.add_argument('--sigma1', default=None, type=float)
    parser.add_argument('--sigma_g', default=None, type=float,
                        help='Standard Deviation for the noise of gradients.')
    # Name of the environment to use
    parser.add_argument('--env_name', type=str, required=True,
                        help='Name of the environment to use', choices=env_supported)
    # Logging Options
    parser.add_argument('--log_level',
                        default='info',
                        choices=['fatal', 'error', 'warning', 'info', 'debug'])
    parser.add_argument('--log_folder', default=None, type=str)
    parser.add_argument('--log_frequency', default=1, type=int)
    # Other details
    parser.add_argument('--exp_name', type=str, default="qd")
    parser.add_argument('-d', '--directory', type=str, default=os.curdir)
    # Hardware Options
    parser.add_argument('--use_cpu', default=0, type=int,help='Use CPU only?')
    parser.add_argument('--devices', default="0", type=str,
                help='Which GPU(s) (identified by their Device ids) to use?'+
                            'E.g. "0,2" use device 0 and 2')
    parser.add_argument('--device_count', default=1, type=str,
                help='Number of cores available to use/devices')
    return parser

def add_args_for_finding_best_params(parser: argparse.ArgumentParser):
    parser.add_argument('-n', '--number_replications', type=int, required=True)
    parser.add_argument('--sols_dim_list',type=int ,nargs='+',default=None) # [16, 64, 256,
1024]

    # Options for algorithm execution space
    parser.add_argument('--episode_len_list',type=int ,nargs='+',default=None)
    parser.add_argument('--num_evaluations', default=None, type=int)
    parser.add_argument('--batch_size', default=2048, type=int)
    parser.add_argument('--num_epochs', default=None, type=int)

    # Covariance Matrix Adaptation Options - CMA (Used only if the emitter is of CMA type)
    parser.add_argument('--stepsize_list', default=None, type=float)
    parser.add_argument('--norm_grad', default=None, type=int)
    # Algorithm Grid shape of archive
    parser.add_argument('--grid_shape', nargs='+', type=int, required=True)
    # Standard deviations for algorithm
    parser.add_argument('--sigma0_list', nargs='+', default=None, type=float)
    # Standard deviations for algorithm
    parser.add_argument('--sigma1_list', nargs='+', default=None, type=float)
    # parser.add_argument('--sigma1', default=0, type=int)
    parser.add_argument('--sigma_g_list',nargs='+', default=None, type=float,
                    help='Standard Deviation for the noise of gradients.')
    # Name of the environment to use
    parser.add_argument('--env_name', type=str, required=True,
                    help='Name of the environment to use')
                    # help='Name of the environment to use', choices=['rastrigin'])
    # Logging Options
    parser.add_argument('--log_level',
                    default='info',
                    choices=['fatal', 'error', 'warning', 'info', 'debug'])
    parser.add_argument('--log_folder', default=None, type=str)
    parser.add_argument('--log_frequency', default=1, type=int)
    # Other details
    parser.add_argument('--exp_name', type=str, default="qd")
    parser.add_argument('-d', '--directory', type=str, default=os.curdir)
    parser.add_argument('--results_f', type=str, default='scores')
    # Hardware Options
    parser.add_argument('--use_cpu', default=0, type=int,help='Use CPU only?')
    parser.add_argument('--devices', default="0", type=str,
                    help='Which GPU(s) (identified by their Device ids) to use?'+
                                'E.g. "0,2" use device 0 and 2')
    parser.add_argument('--device_count', default=1, type=str,
                    help='Number of cores available to use/devices')
    return parser.parse_args()

def add_args_for_batch_sizes_comp(parser: argparse.ArgumentParser, emitters_supported):
    parser.add_argument('-n', '--number_replications', type=int, required=True)
    parser.add_argument('--sols_dim_list',type=int ,nargs='+',default=None) # [16, 64, 256,
1024]
    parser.add_argument('--batch_size_list',type=int ,nargs='+',default=None) # [16, 64, 256,
1024]

    parser.add_argument('--emitter',
                        default=emitters_supported[0],
                        choices=emitters_supported)

    # Options for algorithm execution space
    parser.add_argument('--episode_len_list',type=int ,nargs='+',default=None)
    parser.add_argument('--num_evaluations', default=None, type=int)
    parser.add_argument('--num_epochs', default=None, type=int)

    # Covariance Matrix Adaptation Options - CMA (Used only if the emitter is of CMA type)
    parser.add_argument('--selection_rule', default=None, type=int)
    parser.add_argument('--restart_rule', default=None, type=int)
    parser.add_argument('--weight_rule', default=None, type=int)
    parser.add_argument('--stepsize', default=None, type=float)
    parser.add_argument('--norm_grad', default=None, type=int)
    # Algorithm Grid shape of archive
    parser.add_argument('--grid_shape', nargs='+', type=int, required=True)
    parser.add_argument('--sigma0', default=None, type=float)
    parser.add_argument('--sigma1', default=None, type=float)
    parser.add_argument('--sigma_g', default=None, type=float,
                        help='Standard Deviation for the noise of gradients.')
    # Name of the environment to use
    parser.add_argument('--env_name', type=str, required=True,
                        help='Name of the environment to use')
                        # help='Name of the environment to use', choices=['rastrigin'])
    # Logging Options
    parser.add_argument('--log_level',
                        default='info',
                        choices=['fatal', 'error', 'warning', 'info', 'debug'])
    parser.add_argument('--log_folder', default=None, type=str)
    parser.add_argument('--log_frequency', default=1, type=int)
    # Other details
    parser.add_argument('--exp_name', type=str, default="qd")
    parser.add_argument('-d', '--directory', type=str, default=os.curdir)
    parser.add_argument('--results_f', type=str, default='scores')
    # Hardware Options
    parser.add_argument('--use_cpu', default=0, type=int,help='Use CPU only?')
    parser.add_argument('--devices', default="0", type=str,
                    help='Which GPU(s) (identified by their Device ids) to use?'+
                          'E.g. "0,2" use device 0 and 2')
    parser.add_argument('--device_count', default=1, type=str,
                    help='Number of cores available to use/devices')
    return parser.parse_args()

def check_validity_args(parser: argparse.ArgumentParser,
                        parsed_arguments):
  num_epochs = parsed_arguments.num_epochs
  num_evaluations = parsed_arguments.num_evaluations

  if num_epochs is None and num_evaluations is None:
    parser.error("One (and only one) of the following arguments should be set: --num-epochs or
--num-evaluations")
  elif num_epochs is not None and num_evaluations is not None:
    parser.error("One (and only one) of the following arguments should be set: --num-epochs or
--num-evaluations")
```

Code Snippet A.26: Module for provides command line input parsers for allowing the use of the library's functionalities via command line (run_qd_args_parsers.py)

A.8.5 Module for finding the best parameters for a QD/DQD Emitter for a specific problem size

```
import argparse
import os
from absl import logging,flags
from run_qd_args_parsers import add_args_for_finding_best_params, check_validity_args

from scipy import stats

# QD_EMITTERS_SUPPORTED = ["cma_me_imp", "cma_me_rd", "cma_me_opt"]
# DQD_EMITTERS_SUPPORTED = ["cma_mega", "cma_mega_adam"]
# ALL_EMITTERS_SUPPORTED = QD_EMITTERS_SUPPORTED + DQD_EMITTERS_SUPPORTED

# from jax.config import config
# config.update("jax_enable_x64", True)
# import sys
# jax.numpy.set_printoptions(threshold=sys.maxsize)
args = None
```

```
def process_args():
    parser = argparse.ArgumentParser(formatter_class=argparse.RawTextHelpFormatter)
    add_args_for_finding_best_params(parser)
    parsed_arguments = parser.parse_args()
    check_validity_args(parser, parsed_arguments)
    return parsed_arguments

# Set the environmental variables for Hardware and
#  Jax before importing Jax Library
if __name__ == "__main__":
  try:
    args = process_args()
    # Change environmental variables based on
    # whether to use only cpu or not
    if not args.use_cpu != 0:
      os.environ["XLA_PYTHON_CLIENT_PREALLOCATE"] = "false"
      os.environ["XLA_PYTHON_CLIENT_MEM_FRACTION"] = ".5"
      os.environ["CUDA_VISIBLE_DEVICES"] = args.devices
      # Just uncomment the line below if you want to enforce it
      # os.environ["XLA_FLAGS"] = "--
xla_force_host_platform_device_count="+str(args.device_count)
    else:
      os.environ["JAX_PLATFORM_NAME"] = "cpu"
      # Just uncomment the line below if you want to enforce it
      # os.environ["XLA_FLAGS"] = "--
xla_force_host_platform_device_count="+str(args.device_count)

  except Exception as e:
    logging.fatal(e, exc_info=True)

# Then import ll the rest necessary libraries
import jax
import jax.numpy as jnp
import numpy as np
from jax.lib import xla_bridge
from qdax.training.configuration import Configuration
from qdax.training import qd_loop_simple as qd_simple
from run_qd_utils import get_num_epochs_and_evaluations, get_emitter, get_env_info
import datetime
import random
import sys
import pandas as pd
import json
import csv

def get_score_entries(scores, with_mad=True):
  score_entries = list()

  for emitter, emitter_dict in scores.items():
    for sol_dim, sol_dim_dict in emitter_dict.items():
      scores = list()
      local_scores = list()
      local_params = list()
      local_indicators = list()
      for p, score_stats in sol_dim_dict.items():
        if with_mad:
          local_scores.append(float(score_stats['mean']))
          plus_minus_symbol  ='\u00b1'
          local_indicators.append(str(score_stats['mean']) + " " + plus_minus_symbol
                                  + " " + str(score_stats['mad']))
          local_params.append(p)
        else:
          local_scores.append(float(score_stats['mean']))
          local_indicators = local_scores
          local_params.append(p)

      max_idx = np.argmax(np.array(local_scores))
      score_entry = [emitter, sol_dim, local_params[max_idx], local_indicators[max_idx]]
      score_entries.append(score_entry)

  return score_entries

def save_as_csv(score_entries, filename):
  with open(filename, "w", newline="") as f:
      writer = csv.writer(f)
      writer.writerows(score_entries)
      # pd.DataFrame(np.array(score_entries)).to_csv(filename)

def save_scores_json(scores_dict, filename):
  with open(filename, 'w') as fp:
      json.dump(scores_dict, fp, sort_keys=True, indent=4)

def run_qd(emitter_args, results_saving_folder):
  # Random seed
  seed = random.randrange(sys.maxsize)
  key = jax.random.PRNGKey(seed)
  key, key_emitter_init = jax.random.split(key, 2)
  qd_params = dict()
  # Define arguments of emitter and environemnt
  emitter, static_settings, qd_params = get_emitter(emitter_args, qd_params, key_emitter_init)
  num_epochs, num_evaluations =
get_num_epochs_and_evaluations(num_epochs=emitter_args.num_epochs,
                                                            num_evaluations=emitter_args.num
_evaluations,
                                                            population_size=emitter_args.bat
ch_size,
                                                            use_dqd=qd_params['use_dqd'])

  logging.info(f"Emitter Details:\n"
               f"\t Emitter: {emitter_args.emitter}\n"
               f"\t sigma0: {emitter_args.sigma0}\n"
               f"\t sigma1: {emitter_args.sigma1}\n"
               f"\t sigma_g: {emitter_args.sigma_g}\n"
               f"\t stepsize: {emitter_args.stepsize}\n")

  qd_params['device'] = xla_bridge.get_backend().platform # 'cpu' if args.use_cpu == 1 else
'gpu'
  qd_params['sols_dim'] = emitter_args.sols_dim
  archive_bounds, _, calc_objs, calc_bds = get_env_info(emitter_args)
  configuration = Configuration(args.env_name,
                                num_epochs,
                                0,
                                action_repeat=1,
                                population_size=emitter_args.batch_size,
                                seed=seed,
                                log_frequency=emitter_args.log_frequency,
                                qd_params=qd_params,
                                min_bd=float(archive_bounds[0,0]), #
float(args.min_max_bd[0]),
                                max_bd=float(archive_bounds[0,1]), #
float(args.min_max_bd[1]),
                                grid_shape=tuple(emitter_args.grid_shape),
                                max_devices_per_host=None,
                                )

  training_state = qd_simple.train(
        emitter=emitter,
        emitter_static_settings = static_settings,
        objective_fn=calc_objs,
        bds_fn=calc_bds,
        configuration=configuration,
        progress_fn=None,
        experiment_name=emitter_args.exp_name,
        key=key,
        result_path=results_saving_folder,
        save_results=True
      )
  return training_state
```

```python
def save_scores(results_f, scores_dict):
  score_entries = get_score_entries(scores_dict)
  current_time = datetime.datetime.now().strftime("%Y-%m-%d_T_%H-%M-%S")
  csv_filename =  '{0}_{1}_best_scores.csv'.format(results_f, str(current_time))
  save_as_csv(score_entries, csv_filename)
  json_filename =  '{0}_{1}_scores.json'.format(results_f, str(current_time))
  save_scores_json(scores_dict, json_filename)

def main(parsed_arguments):
  if parsed_arguments.log_folder:
    if not os.path.exists(parsed_arguments.log_folder):
      raise FileNotFoundError("Folder {} not found.".format(parsed_arguments.log_folder))
    else:
      datetime_now = datetime.datetime.now().strftime("%Y-%m-%d_T_%H-%M-%S")
      logging.get_absl_handler().use_absl_log_file('log_{0}.log'.format(datetime_now),
                                                   parsed_arguments.log_folder)
      flags.FLAGS.mark_as_parsed()
  results_saving_folder = parsed_arguments.directory
  # logging.get_absl_handler().setFormatter(None)

  if not os.path.exists(results_saving_folder):
    raise FileNotFoundError(f"Folder {results_saving_folder} not found.")

  levels = {'fatal': logging.FATAL,
            'error': logging.ERROR,
            'warning': logging.WARNING,
            'info': logging.INFO,
            'debug': logging.DEBUG}

  logging.set_verbosity(levels[parsed_arguments.log_level])
  population_size = parsed_arguments.batch_size
  local_device_count = jax.local_device_count()
  local_devices_to_use = local_device_count
  process_count = jax.process_count()

  logging.info(f"Hardware Details:\n"
               f"\t Platform: {xla_bridge.get_backend().platform}\n"
               f"\t Local_device_count: {local_device_count}\n"
               f"\t process_count: {process_count}\n")

  sols_dim_list = parsed_arguments.sols_dim_list
  sols_dim_list = sols_dim_list if sols_dim_list else [1024]
  sigma0_list =  parsed_arguments.sigma0_list
  sigma0_list = sigma0_list if sigma0_list else [0.005, 0.01, 0.02, 0.03, 0.04, 0.05,
                                                 0.06, 0.07, 0.08, 0.09, 0.1, 0.15,
                                                  0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8,
                                                   0.9, 1, 2,3,4, 5, 7, 10, 15, 20]
  sigma1_list = parsed_arguments.sigma1_list
  sigma1_list = sigma1_list if sigma1_list else [0.01, 0.05, 0.1, 0.2, 0.5, 1]
  sigma_g_list = parsed_arguments.sigma_g_list
  sigma_g_list = sigma_g_list if sigma_g_list else [0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1, 5, 10]
  stepsize_list = parsed_arguments.stepsize_list
  stepsize_list = stepsize_list if stepsize_list else [0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1, 5,
10]
  number_replications = parsed_arguments.number_replications

  scores_dict = dict()

  # for emitter_name in ["cma_mega", "cma_mega_adam"]:
  #   scores_dict[emitter_name] = dict()
  #   emitter_scores = scores_dict[emitter_name]
  #   for sols_dim in sols_dim_list:
  #     emitter_scores[sols_dim] = dict()
  #     sol_dim_scores = emitter_scores[sols_dim]
  #     for sigma_g in sigma_g_list:
  #       for stepsize in stepsize_list:
  #         args_dict = vars(parsed_arguments)
  #         emitter_args = argparse.Namespace(**args_dict)
  #         emitter_args.emitter = emitter_name
  #         emitter_args.sigma0 = None
  #         emitter_args.sigma1 = None
  #         emitter_args.sigma_g = sigma_g
  #         emitter_args.selection_rule = None
  #         emitter_args.restart_rule = None
  #         emitter_args.weight_rule = None
  #         emitter_args.sols_dim = sols_dim
  #         emitter_args.stepsize = stepsize
  #         scores = list()

  #         for _ in range(number_replications):
  #           training_state = run_qd(emitter_args, results_saving_folder)
  #           epoch, archive_size, best_fit, qd_score = training_state.metrics.scores[-1,:]
  #           scores.append(qd_score)

  #         scores = np.array(scores)
  #         score = np.median(scores)
  #         # Median Absolute Deviation
  #         mad = stats.median_absolute_deviation(scores, axis=None)
  #         key = 'sigma_g={0}-stepsize={1}'.format(sigma_g,stepsize)
  #         sol_dim_scores[key] = dict()
  #         sol_dim_scores[key]['mean'] = str(score)
  #         sol_dim_scores[key]['mad'] = str(mad)

  # results_f = parsed_arguments.results_f
  # save_scores(results_f, scores_dict)

  for emitter_name in ["cma_me_imp", "cma_me_rd", "cma_me_opt"]: # "map_elites",
    scores_dict[emitter_name] = dict()
    emitter_scores = scores_dict[emitter_name]
    for sols_dim in sols_dim_list:
      emitter_scores[sols_dim] = dict()
      sol_dim_scores = emitter_scores[sols_dim]
      for sigma0 in sigma0_list:
          args_dict = vars(args)
          emitter_args = argparse.Namespace(**args_dict)
          emitter_args.emitter = emitter_name
          emitter_args.sigma0 = sigma0
          emitter_args.sigma1 = None
          emitter_args.sigma_g = None
          emitter_args.selection_rule = None
          emitter_args.restart_rule = None
          emitter_args.weight_rule = None
          emitter_args.sols_dim = sols_dim
          emitter_args.stepsize = None
          scores = list()

          for _ in range(number_replications):
            training_state = run_qd(emitter_args, results_saving_folder)
            epoch, archive_size, best_fit, qd_score = training_state.metrics.scores[-1,:]
            scores.append(qd_score)

          scores = np.array(scores)
          score = np.median(scores)
          # Median Absolute Deviation
          mad = stats.median_absolute_deviation(scores, axis=None)
          key = 'sigma0={0}'.format(sigma0)
          sol_dim_scores[key] = dict()
          sol_dim_scores[key]['mean'] = str(score)
          sol_dim_scores[key]['mad'] = str(mad)

  results_f = parsed_arguments.results_f
  save_scores(results_f, scores_dict)

  return

  for emitter_name in ["map_elites_iso"]:
    scores_dict[emitter_name] = dict()
    emitter_scores = scores_dict[emitter_name]
    for sols_dim in sols_dim_list:
      emitter_scores[sols_dim] = dict()
      sol_dim_scores = emitter_scores[sols_dim]
      for sigma0 in sigma0_list:
        for sigma1 in sigma1_list:
          args_dict = vars(args)
          emitter_args = argparse.Namespace(**args_dict)
          emitter_args.emitter = emitter_name
          emitter_args.sigma0 = sigma0
          emitter_args.sigma1 = sigma1
          emitter_args.sigma_g = None
          emitter_args.selection_rule = None
          emitter_args.restart_rule = None
          emitter_args.weight_rule = None
          emitter_args.sols_dim = sols_dim
          emitter_args.stepsize = None
          scores = list()

          for _ in range(number_replications):
            training_state = run_qd(emitter_args, results_saving_folder)
            epoch, archive_size, best_fit, qd_score = training_state.metrics.scores[-1,:]
            scores.append(qd_score)

          scores = np.array(scores)
          score = np.median(scores)
          # Median Absolute Deviation
          mad = stats.median_absolute_deviation(scores, axis=None)
          key = 'sigma0={0},sigma1={1}'.format(sigma0, sigma1)
          sol_dim_scores[key] = dict()
          sol_dim_scores[key]['mean'] = str(score)
          sol_dim_scores[key]['mad'] = str(mad)

  results_f = parsed_arguments.results_f
  save_scores(results_f, scores_dict)

  for emitter_name in ["og_map_elites_line", "omg_mega_line"]:
    scores_dict[emitter_name] = dict()
    emitter_scores = scores_dict[emitter_name]
    for sols_dim in sols_dim_list:
      emitter_scores[sols_dim] = dict()
      sol_dim_scores = emitter_scores[sols_dim]
      for sigma0 in sigma0_list:
        for sigma1 in sigma1_list:
          for sigma_g in sigma_g_list:
            args_dict = vars(args)
            emitter_args = argparse.Namespace(**args_dict)
            emitter_args.emitter = emitter_name
            emitter_args.sigma0 = sigma0
            emitter_args.sigma1 = sigma1
            emitter_args.sigma_g = sigma_g
            emitter_args.selection_rule = None
            emitter_args.restart_rule = None
            emitter_args.weight_rule = None
            emitter_args.sols_dim = sols_dim
            emitter_args.stepsize = None
            scores = list()

            for _ in range(number_replications):
              training_state = run_qd(emitter_args, results_saving_folder)
              epoch, archive_size, best_fit, qd_score = training_state.metrics.scores[-1,:]
              scores.append(qd_score)

            scores = np.array(scores)
            score = np.median(scores)
            # Median Absolute Deviation
            mad = stats.median_absolute_deviation(scores, axis=None)
            key = 'sigma0={0},sigma1={1},sigma_g={2}'.format(sigma0, sigma1, sigma_g)
            sol_dim_scores[key] = dict()
            sol_dim_scores[key]['mean'] = str(score)
            sol_dim_scores[key]['mad'] = str(mad)

  results_f = parsed_arguments.results_f
  save_scores(results_f, scores_dict)

  # for emitter_name in ["og_map_elites", "omg_mega"]:
  #   scores_dict[emitter_name] = dict()
  #   emitter_scores = scores_dict[emitter_name]
  #   for sols_dim in sols_dim_list:
  #     emitter_scores[sols_dim] = dict()
  #     sol_dim_scores = emitter_scores[sols_dim]
  #     for sigma0 in sigma0_list:
  #       for sigma_g in sigma_g_list:
  #         args_dict = vars(args)
  #         emitter_args = argparse.Namespace(**args_dict)
  #         emitter_args.emitter = emitter_name
  #         emitter_args.sigma0 = sigma0
  #         emitter_args.sigma1 = None
  #         emitter_args.sigma_g = sigma_g
  #         emitter_args.selection_rule = None
  #         emitter_args.restart_rule = None
  #         emitter_args.weight_rule = None
  #         emitter_args.sols_dim = sols_dim
  #         emitter_args.stepsize = None
  #         scores = list()

  #         for _ in range(number_replications):
  #           training_state = run_qd(emitter_args, results_saving_folder)
```

```
  #           epoch, archive_size, best_fit, qd_score = training_state.metrics.scores[-1,:]
  #           scores.append(qd_score)

  #         scores = np.array(scores)
  #         score = np.median(scores)
  #         # Median Absolute Deviation
  #         mad = stats.median_absolute_deviation(scores, axis=None)
  #         key = 'sigma0={0},sigma_g={1}'.format(sigma0, sigma_g)
  #         sol_dim_scores[key] = dict()
  #         sol_dim_scores[key]['mean'] = str(score)
  #         sol_dim_scores[key]['mad'] = str(mad)

  # results_f = parsed_arguments.results_f
  # save_scores(results_f, scores_dict)


if __name__ == "__main__":
  try:
    main(args)
  except Exception as e:
    logging.fatal(e, exc_info=True)
```

Code Snippet A.27: Module used for identifying the parameters of QD and DQD algorithms that give the best QD Scores (find_best_params.py)

A.8.6 Module to execute a QD or DQD Emitter for different combinations of batch sizes and problem sizes

```
import argparse
import os
from absl import logging,flags
from run_qd_args_parsers import add_args_for_batch_sizes_comp, check_validity_args

from scipy import stats

QD_EMITTERS_SUPPORTED = ["map_elites", "map_elites_iso", "cma_me_imp",
                        "cma_me_rd", "cma_me_opt"]
DQD_EMITTERS_SUPPORTED = ["og_map_elites", "og_map_elites_line",
                          "omg_mega", "cma_mega", "cma_mega_adam", "omg_mega_line"]
ALL_EMITTERS_SUPPORTED = QD_EMITTERS_SUPPORTED + DQD_EMITTERS_SUPPORTED

# from jax.config import config
# config.update("jax_enable_x64", True)
# import sys
# jax.numpy.set_printoptions(threshold=sys.maxsize)
```

```python
args = None

def process_args():
    parser = argparse.ArgumentParser(formatter_class=argparse.RawTextHelpFormatter)
    add_args_for_batch_sizes_comp(parser, ALL_EMITTERS_SUPPORTED)
    parsed_arguments = parser.parse_args()
    check_validity_args(parser, parsed_arguments)
    return parsed_arguments

# Set the environmental variables for Hardware and
#  Jax before importing Jax Library
if __name__ == "__main__":
  try:
    args = process_args()
    # Change environmental variables based on
    # whether to use only cpu or not
    if not args.use_cpu != 0:
      os.environ["XLA_PYTHON_CLIENT_PREALLOCATE"] = "false"
      os.environ["XLA_PYTHON_CLIENT_MEM_FRACTION"] = ".5"
      os.environ["CUDA_VISIBLE_DEVICES"] = args.devices
      # Just uncomment the line below if you want to enforce it
      # os.environ["XLA_FLAGS"] = "--
xla_force_host_platform_device_count="+str(args.device_count)
    else:
      os.environ["JAX_PLATFORM_NAME"] = "cpu"
      # Just uncomment the line below if you want to enforce it
      # os.environ["XLA_FLAGS"] = "--
xla_force_host_platform_device_count="+str(args.device_count)

  except Exception as e:
    logging.fatal(e, exc_info=True)

# Then import ll the rest necessary libraries
import jax
import jax.numpy as jnp
import numpy as np
from jax.lib import xla_bridge
from qdax.training.configuration import Configuration
from qdax.training import qd_loop_simple as qd_simple
from run_qd_utils import get_num_epochs_and_evaluations, get_emitter, get_env_info
import datetime
import random
import sys
```

```python
def run_qd(emitter_args, results_saving_folder, local_devices_to_use, process_count):
  # Random seed
  seed = random.randrange(sys.maxsize)
  key = jax.random.PRNGKey(seed)
  key, key_emitter_init = jax.random.split(key, 2)
  qd_params = dict()
  # Define arguments of emitter and environemnt
  emitter, static_settings, qd_params = get_emitter(emitter_args, qd_params, key_emitter_init)
  num_epochs, num_evaluations =
get_num_epochs_and_evaluations(num_epochs=emitter_args.num_epochs,
                                                              num_evaluations=emitter_args.num
_evaluations,
                                                              population_size=emitter_args.bat
ch_size,
                                                              use_dqd=qd_params['use_dqd'])

  logging.info(f"Emitter Details:\n"
               f"\t Emitter: {emitter_args.emitter}\n"
               f"\t sigma0: {emitter_args.sigma0}\n"
               f"\t sigma1: {emitter_args.sigma1}\n"
               f"\t sigma_g: {emitter_args.sigma_g}\n"
               f"\t stepsize: {emitter_args.stepsize}\n")

  qd_params['device'] = xla_bridge.get_backend().platform # 'cpu' if args.use_cpu == 1 else
'gpu'
  qd_params['sols_dim'] = emitter_args.sols_dim
  archive_bounds, _, calc_objs, calc_bds = get_env_info(emitter_args)
  configuration = Configuration(args.env_name,
                                num_epochs,
                                0,
                                action_repeat=1,
                                population_size=emitter_args.batch_size,
                                seed=seed,
                                log_frequency=emitter_args.log_frequency,
                                qd_params=qd_params,
                                min_bd=float(archive_bounds[0,0]), #
float(args.min_max_bd[0]),
                                max_bd=float(archive_bounds[0,1]), #
float(args.min_max_bd[1]),
                                grid_shape=tuple(emitter_args.grid_shape),
                                max_devices_per_host=None,
                                )

  training_state = qd_simple.train(
        emitter=emitter,
        emitter_static_settings = static_settings,
        objective_fn=calc_objs,
        bds_fn=calc_bds,
        configuration=configuration,
        progress_fn=None,
        experiment_name=emitter_args.exp_name,
        key=key,
        result_path=results_saving_folder,
        save_results=True
      )
  return training_state


def main(parsed_arguments):
  if parsed_arguments.log_folder:
    if not os.path.exists(parsed_arguments.log_folder):
      raise FileNotFoundError("Folder {} not found.".format(parsed_arguments.log_folder))
    else:
      datetime_now = datetime.datetime.now().strftime("%Y-%m-%d_T_%H-%M-%S")
      logging.get_absl_handler().use_absl_log_file('log_{0}.log'.format(datetime_now),
                                                    parsed_arguments.log_folder)
      flags.FLAGS.mark_as_parsed()
  results_saving_folder = parsed_arguments.directory
  # logging.get_absl_handler().setFormatter(None)

  if not os.path.exists(results_saving_folder):
    raise FileNotFoundError(f"Folder {results_saving_folder} not found.")

  levels = {'fatal': logging.FATAL,
            'error': logging.ERROR,
            'warning': logging.WARNING,
            'info': logging.INFO,
            'debug': logging.DEBUG}

  logging.set_verbosity(levels[parsed_arguments.log_level])
  local_device_count = jax.local_device_count()
  local_devices_to_use = local_device_count
  process_count = jax.process_count()

  logging.info(f"Hardware Details:\n"
               f"\t Platform: {xla_bridge.get_backend().platform}\n"
               f"\t Local_device_count: {local_device_count}\n"
               f"\t process_count: {process_count}\n")

  sols_dim_list = parsed_arguments.sols_dim_list
  sols_dim_list = sols_dim_list if sols_dim_list else [128, 256, 512, 1024, 2048]
  batch_size_list = parsed_arguments.batch_size_list
  batch_size_list = batch_size_list if batch_size_list else [512, 2048, 8192, 16384]
  number_replications = parsed_arguments.number_replications

  for sols_dim in sols_dim_list:
    for batch_size in batch_size_list:
        args_dict = vars(args)
        emitter_args = argparse.Namespace(**args_dict)
        emitter_args.sols_dim = sols_dim
        emitter_args.batch_size = batch_size
        scores = list()
        for _ in range(number_replications):
          run_qd(emitter_args, results_saving_folder, local_devices_to_use, process_count)

if __name__ == "__main__":
  try:
    main(args)
  except Exception as e:
    logging.fatal(e, exc_info=True)
```

Code Snippet A.28: Module for executing different QD and DQD Algorithms with different combinations of batch sizes and problem sizes (run_batch_and_problem_sizes.py)

A.8.7 Module for the Training State of a QD or DQD Algorithm

```
import os
import pickle

import flax.struct

from brax.training.types import PRNGKey
from qd_utils import grid_archive
from stats.metrics import Metrics
import stats.saving_loading_utils as saving_loading_utils
import jax.numpy as jnp

'''
This class represents a simple training state as opposed to the actual TrainingState class.
Simplicity occurs at the attributes of this class. Currently, the only difference is the
state attribute which instead of envs.State it's just a one dimensional ndarray
'''

@flax.struct.dataclass
class SimpleTrainingState:
  """Contains training state for the learner."""
  key: PRNGKey
  repertoire: grid_archive.Repertoire
  metrics: Metrics
  state: jnp.ndarray

  def save(self,
           folder: str = os.curdir,
           name_file: str = "training_state.pkl",
           ) -> None:
    saving_loading_utils.save_dataclass(
      dataclass_object=self,
      folder=folder,
      name_file=name_file,
    )
```

Code Snippet A.29: Module for storing the progress of a QD or DQD Algorithm (training_state_simple.py)

A.8.8 Module for storing the general configurations of a QD or DQD Algorithm

```
import json
import os
from typing import Dict, Any, Optional, Tuple

import dataclasses
from dataclasses import dataclass


@dataclass
class Configuration:
  env_name: str
  num_epochs: int
  episode_length: int
  action_repeat: int
  population_size: int
  seed: int
  log_frequency: int
  qd_params: Dict[str, Any]
  min_bd: float  # Assume  each BD dimension has same bounds
  max_bd: float
  grid_shape: tuple
  max_devices_per_host: Optional[int] = None

  def save_to_json(self,
                   folder: str = os.curdir,
                   name_file: str = "configuration.json",
                   ):
```

```
    path_file = os.path.join(folder, name_file)
    with open(path_file, "w") as json_file:
      json.dump(dataclasses.asdict(self),
                json_file,
                indent=1,
                )

  @staticmethod
  def get_evaluations_num(self):
    if self.qd_params.get('use_dqd') >= 1:
      return self.num_epochs*self.population_size*2
    else:
      return self.num_epochs*self.population_size


  @classmethod
  def load_from_json(cls,
                     path_file: str
                     ) -> 'Configuration':
    with open(path_file, "r") as json_file:
      configuration_dictionary = json.load(json_file)

    return cls(**configuration_dictionary)

  def get_results_folder(self, experiment_name):
    current_results_dict = dataclasses.asdict(self)
    current_results_dict.pop("seed")
    current_results_dict.pop("log_frequency")
    current_results_dict.pop("max_devices_per_host")
    current_results_dict.pop("min_bd")
    current_results_dict.pop("max_bd")
    current_results_dict.pop("action_repeat")

    return
f"{experiment_name}{self.fix_name_folder(self.get_all_variables_str_from_dict(current_results_
dict))}"

  @staticmethod
  def get_all_variables_str_from_dict(dictionary):
    all_variables_str = ""

    for variable_str, value in dictionary.items():
      # print("Variable str: ",variable_str, "value: ", value)
      if isinstance(value, int) or isinstance(value, float) or isinstance(value, str):
        all_variables_str = all_variables_str + f"_{variable_str}-{value}"
      elif isinstance(value, dict):
        all_variables_str += Configuration.get_all_variables_str_from_dict(value)
      elif isinstance(value, tuple):
        all_variables_str = all_variables_str + f"_{variable_str}-{'-'.join(map(str, value))}"
      else:
        print(f"WARNING: Unexpected type encountered when computing results folder name, for
variable {variable_str}")

    return Configuration.fix_name_folder(all_variables_str)

  @staticmethod
  def fix_name_folder(name_folder):
    return name_folder\
      .strip()\
      .lower()\
      .replace(' ', '')\
      .replace('.', '-')\
      .replace('=', '-')
```

Code Snippet A.30: Module for storing the general QD or DQD configurations of a single run (configurations.py)

A.8.9 Module for storing metrics for QD or DQD Execution

```
import os
from typing import Any

import flax.struct
import numpy as np
from dataclasses import dataclass
from jax import numpy as jnp

from stats import saving_loading_utils

Array = Any


@dataclass
class MetricsData:
  archives: np.ndarray
  scores: np.ndarray

  def save(self,
           folder: str = os.curdir,
           name_file: str = "metrics.pkl",
           ):
    saving_loading_utils.save_dataclass(
      dataclass_object=self,
      folder=folder,
      name_file=name_file,
    )

  @classmethod
  def from_metrics(cls,
                   metrics: 'Metrics'
                   ) -> 'MetricsData':
    return cls(
      archives=np.asarray(metrics.archives),
      scores=np.asarray(metrics.scores)
    )


@flax.struct.dataclass
class Metrics:
  archives: Array
  scores: Array

  @classmethod
  def create(cls, num_epochs, log_frequency, grid_shape):
    log_size = jnp.ceil(num_epochs / log_frequency).astype(int) + 1
    archives = jnp.zeros(tuple(jnp.append(jnp.array([log_size]), grid_shape)))
    scores = jnp.zeros([log_size, 4])  # epoch, archive size, best fit, QD score
    return Metrics(archives=archives, scores=scores)

  def save(self,
           folder: str = os.curdir,
           name_file: str = "metrics.pkl",
           ):
    saving_loading_utils.save_dataclass(
      dataclass_object=self,
      folder=folder,
      name_file=name_file,
    )
```

Code Snippet A.31: Module for storing the metrics of a QD or DQD execution (metrics.py)

A.8.10 Utilities module for saving and loading a QD's or DQD's execution data

```
import datetime
import os
import os.path as osp
import pickle
import uuid
from typing import Any

from training.configuration import Configuration


def save_dataclass(
    dataclass_object,
    folder: str,
    name_file: str,
) -> None:
  path_save_file = os.path.join(folder, name_file)
  with open(path_save_file, "wb") as file_to_save:
    pickle.dump(dataclass_object, file_to_save)


def load_dataclass(
    path_file_to_load: str,
) -> Any:
  with open(path_file_to_load, "rb") as file_to_load:
    return pickle.load(file_to_load)


# Saving metrics and timings
def make_results_folder(result_path,
                        experiment_name,
                        configuration: Configuration):

  path_folder_replication = osp.join(
    result_path,
    configuration.get_results_folder(experiment_name),
    f"{datetime.datetime.now().strftime('%Y-%m-%d_%H-%M-%S')}_{uuid.uuid4()}",
  )

  os.makedirs(path_folder_replication)

  return path_folder_replication
```

Code Snippet A.32: Utilities module for saving and loading a QD’s or DQD’s execution data
(saving_loading_utils.py)

Appendix B - Algorithms Parameters

B.1.1 MAP-Elites (Isotropic Gaussian)

- Rastrigin function with simple encoding
  - Sigma_0 = 0.01
- Arm Repertoire
  - Sigma_0 = 0.05
- Rastrigin function with distorted behavior space
  - Sigma_0 = 1

B.1.2 MAP-Elites (Iso+lineDD)

- Rastrigin function with simple encoding
  - Sigma_0 (iso sigma) = 0.01
  - Sigma_1 (line sigma)= 0.2
- Arm Repertoire
  - Sigma_0 (iso sigma) = 0.01
  - Sigma_1 (line sigma)= 0.5
- Rastrigin function with distorted behavior space
  - Sigma_0 (iso sigma) = 0.5
  - Sigma_1 (line sigma)= 0.5

B.1.3 CMA-ME (Improvement)

- Rastrigin function with simple encoding
  - Sigma_0 = 0.05
- Arm Repertoire
  - Sigma_0 = 0.02
- Rastrigin function with distorted behavior space
  - Sigma_0 = 2

For all the cases above we also used the following global parameters:

- Selection Rule = FILTER
- Restart Rule = No Improvement

- Weight Rule = Truncation
- Normalize Gradients = True

B.1.4 CMA-ME (Optimizing)

- Rastrigin function with simple encoding
    - Sigma_0 = 0.05
- Arm Repertoire
    - Sigma_0 = 0.05
- Rastrigin function with distorted behavior space
    - Sigma_0 = 2

For all the cases above we also used the following global parameters:

- Selection Rule = FILTER
- Restart Rule = No Improvement
- Weight Rule = Truncation
- Normalize Gradients = True

B.1.5 CMA-ME (Random Direction)

- Rastrigin function with simple encoding
    - Sigma_0 = 0.01
- Arm Repertoire
    - Sigma_0 = 0.01
- Rastrigin function with distorted behavior space
    - Sigma_0 = 2

For all the cases above we also used the following global parameters:

- Selection Rule = FILTER
- Restart Rule = No Improvement
- Weight Rule = Truncation
- Normalize Gradients = True

B.1.6 OMG-MEGA (line)

- Rastrigin function with simple encoding
  - Sigma_0 (iso sigma) = 0.01
  - Sigma_1 (line sigma) = 0.5
  - Sigma_g (gradient sigma) = 10
- Arm Repertoire
  - Sigma_0 (iso sigma) = 0.01
  - Sigma_1 (line sigma) = 0.1
  - Sigma_g (gradient sigma) = 1
- Rastrigin function with distorted behavior space
  - Sigma_0 (iso sigma) = 0.5
  - Sigma_1 (line sigma) = 0.5
  - Sigma_g (gradient sigma) = 10

B.1.7 OG-MAP-Elites (line)

- Rastrigin function with simple encoding
  - Sigma_0 (iso sigma) = 0.01
  - Sigma_1 (line sigma) = 1
  - Sigma_g (gradient sigma) = 10
- Arm Repertoire
  - Sigma_0 (iso sigma) = 0.01
  - Sigma_1 (line sigma) = 0.2
  - Sigma_g (gradient sigma) = 5
- Rastrigin function with distorted behavior space
  - Sigma_0 (iso sigma) = 0.5
  - Sigma_1 (line sigma) = 0.2
  - Sigma_g (gradient sigma) = 5

B.1.8 CMA-MEGA (Gradient Ascent)

- Rastrigin function with simple encoding
  - Sigma_g (gradient sigma) = 1
  - Stepsize = 0.01

- Arm Repertoire
    - Sigma_g (gradient sigma) = 5
    - Stepsize = 0.05
- Rastrigin function with distorted behavior space
    - Sigma_g (gradient sigma) = 10
    - Stepsize = 0.5

For all the cases above we also used the following global parameters:

- Selection Rule = MU
- Restart Rule = No Improvement
- Weight Rule = Truncation
- Normalize Gradients = True

B.1.9 CMA-MEGA (Adam)

- Rastrigin function with simple encoding
    - Sigma_g (gradient sigma) = 0.5
    - Stepsize = 0.1
- Arm Repertoire
    - Sigma_g (gradient sigma) = 2
    - Stepsize = 1
- Rastrigin function with distorted behavior space
    - Sigma_g (gradient sigma) = 10
    - Stepsize = 0.05

For all the cases above we also used the following global parameters:

- Selection Rule = MU
- Restart Rule = No Improvement
- Weight Rule = Truncation
- Normalize Gradients = True

Appendix C - Results

C.1 Heatmap of QD Scores of algorithms on different problem sizes and batch sizes

All the charts below show the QD Scores (the bar on the right shows the value of the QD Score each color represents) of QD and DQD algorithms for different combinations of batch sizes and problem sizes.

C.1.1 Rastrigin with Distorted Behavior Space

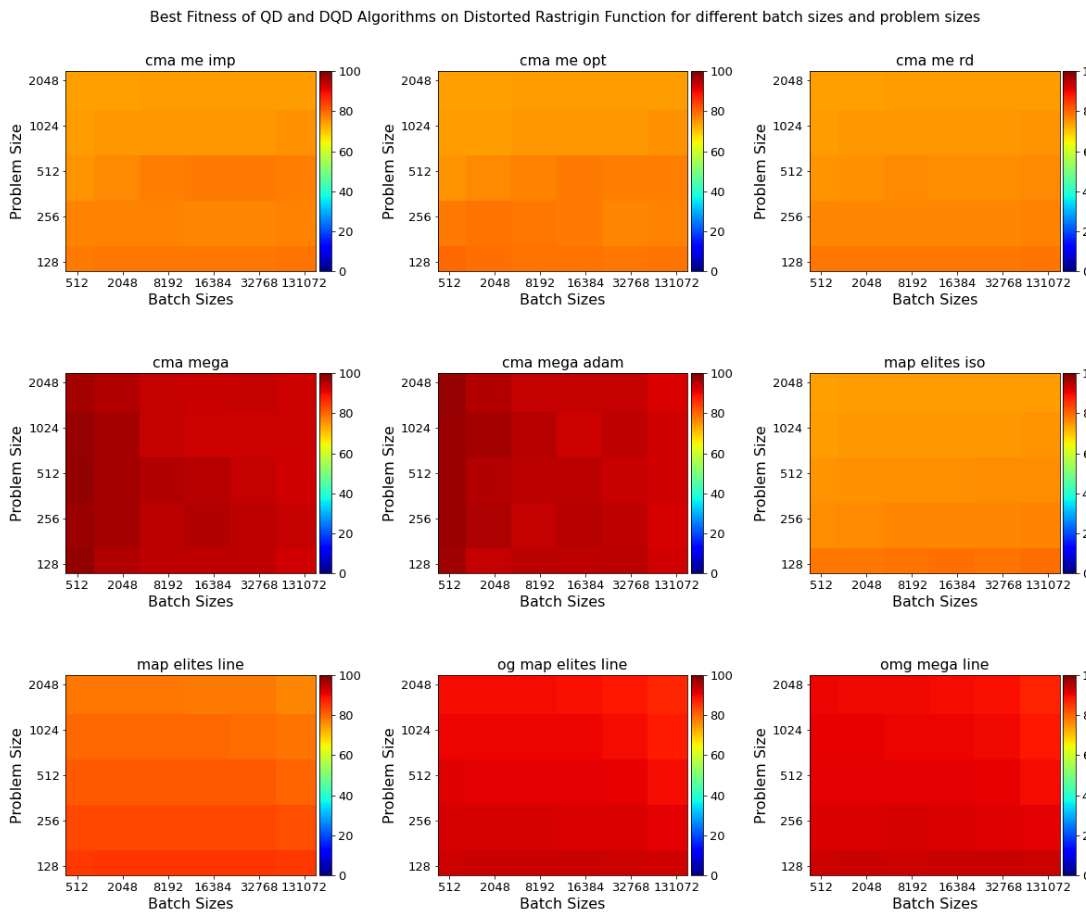


Fig C.1: Best Fitness of QD and DQD Algorithms on the Rastrigin with Distorted Behavior Space problem for different batch sizes and problem sizes

C.1.2 Arm Repertoire

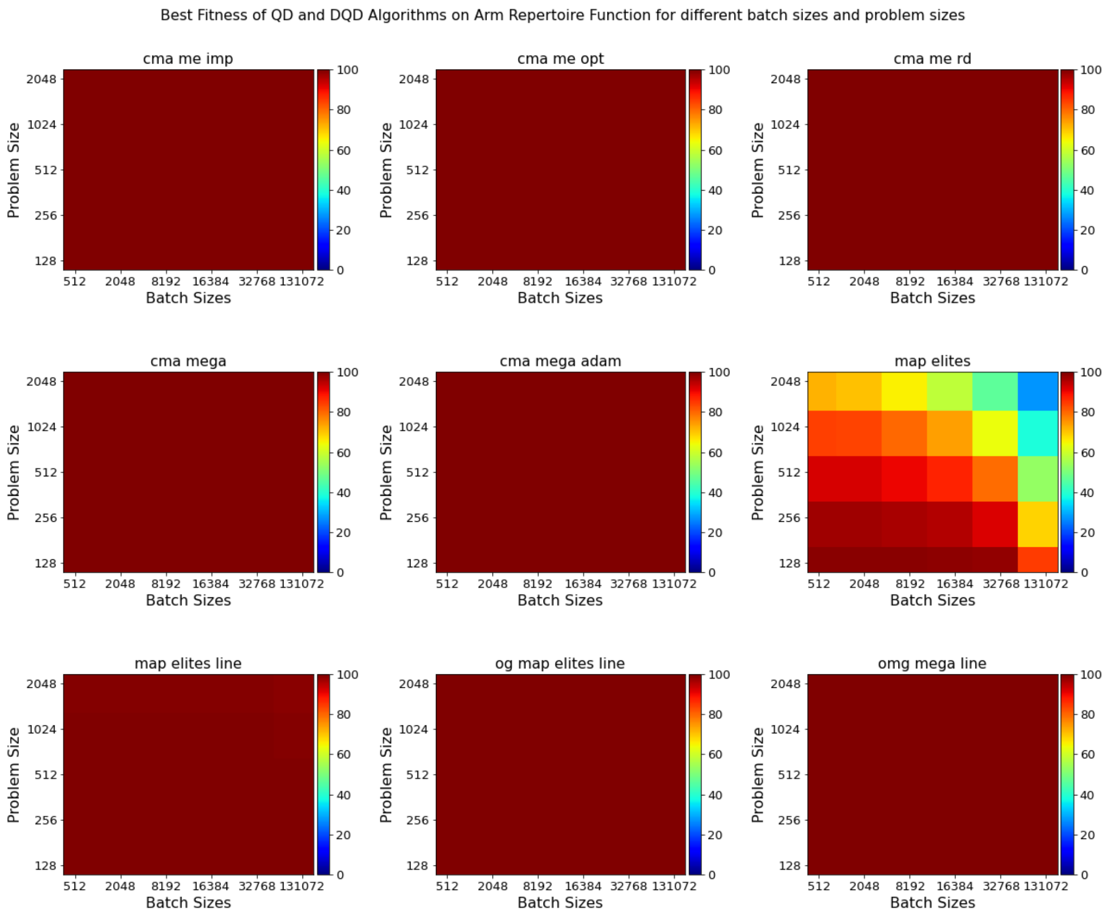


Fig C.2: Best Fitness of QD and DQD Algorithms on the Arm Repertoire problem for different batch sizes and problem sizes

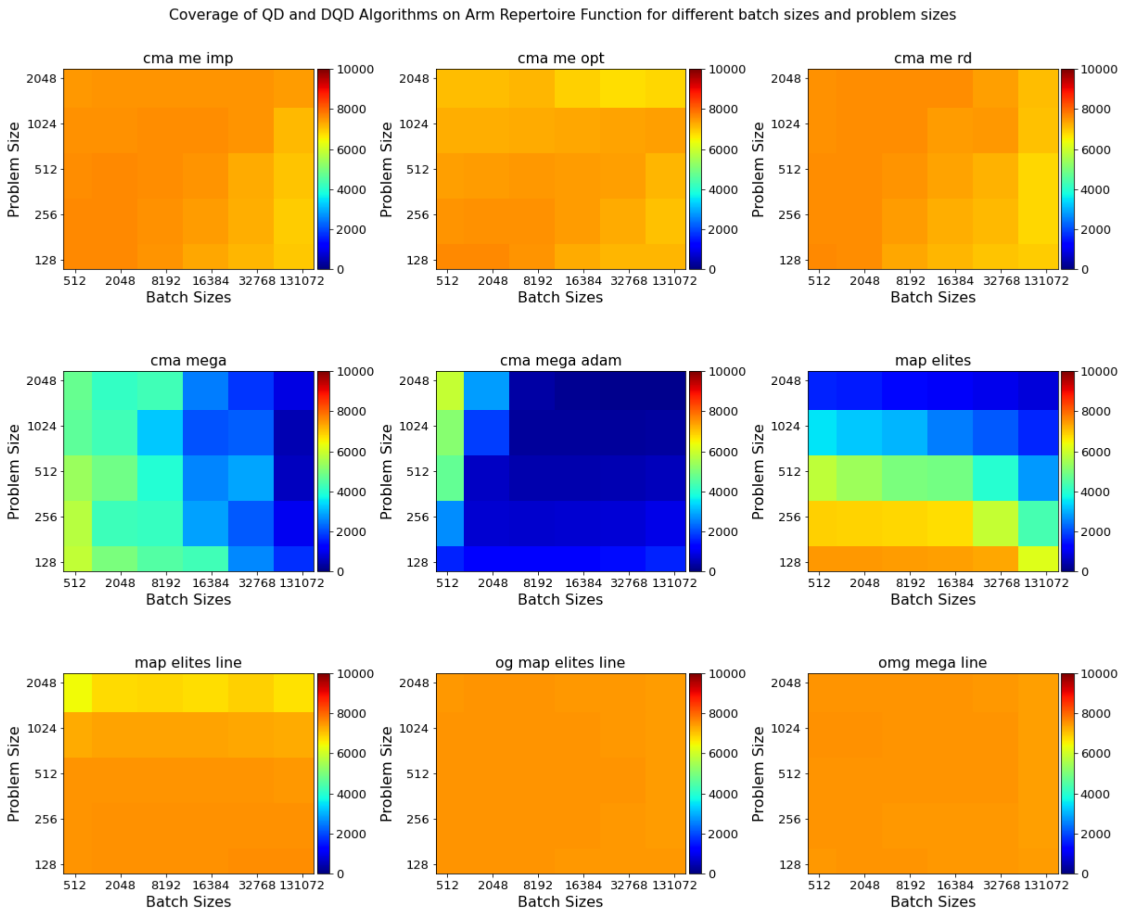


Fig C.3: Coverage of QD and DQD Algorithms on the Arm Repertoire problem for different batch sizes and problem sizes

C.1.3 Rastrigin with simple encoding

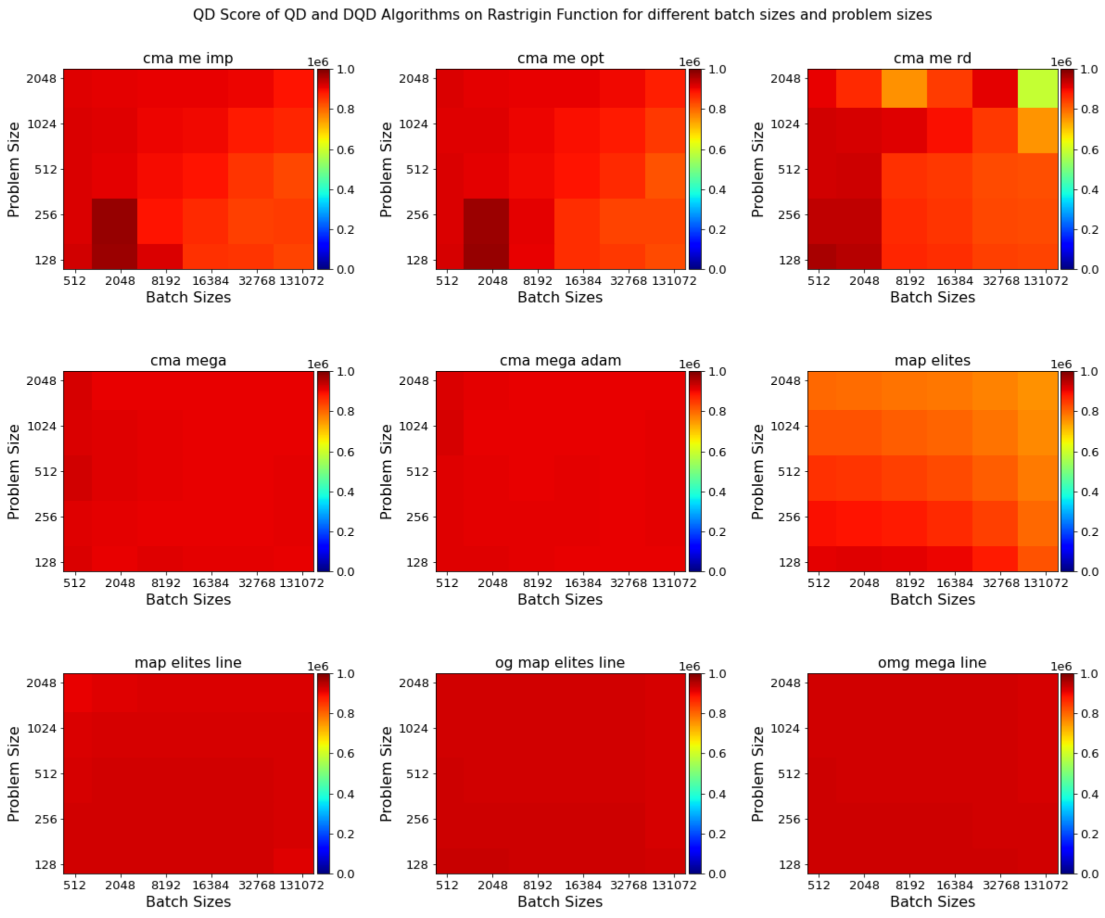


Fig C.4: QD Scores of QD and DQD Algorithms on the Rastrigin with simple encoding problem for different batch sizes and problem sizes

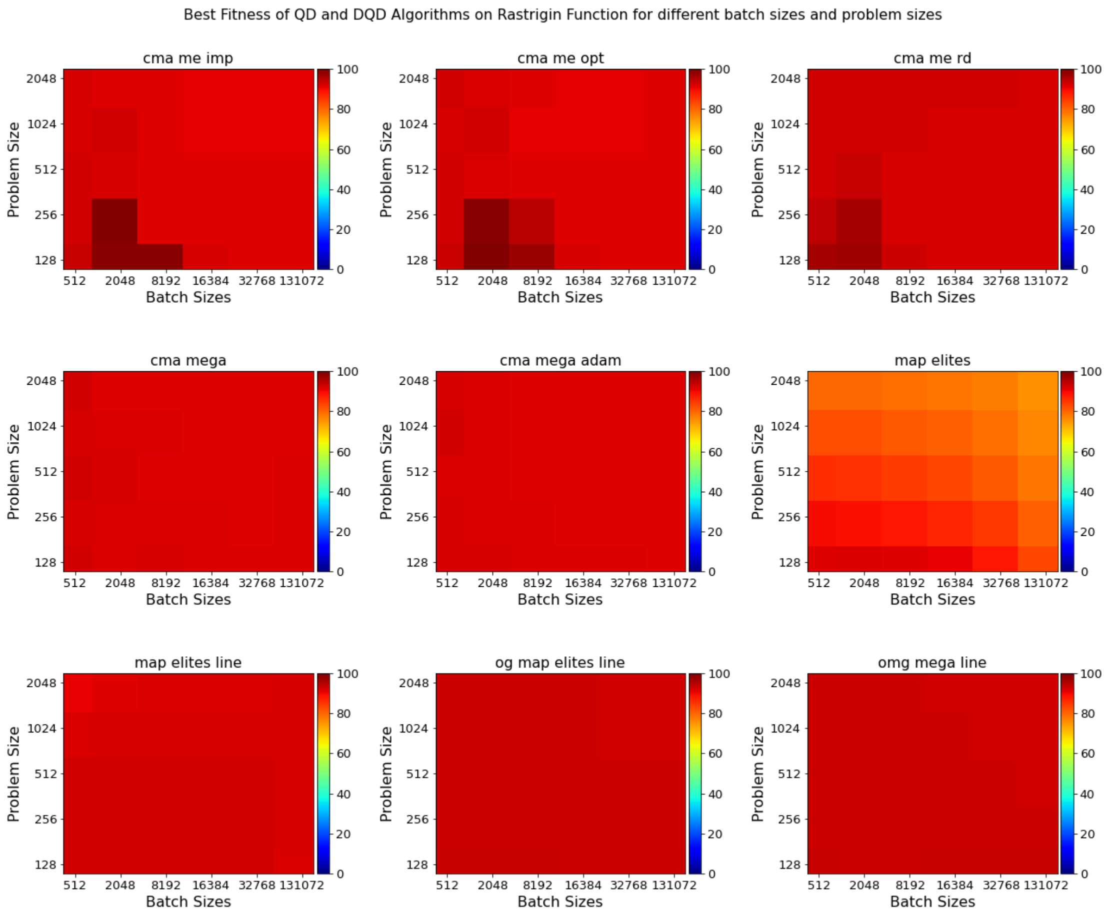


Fig C.5: Best Fitness of QD and DQD Algorithms on the Rastrigin with simple encoding problem for different batch sizes and problem sizes

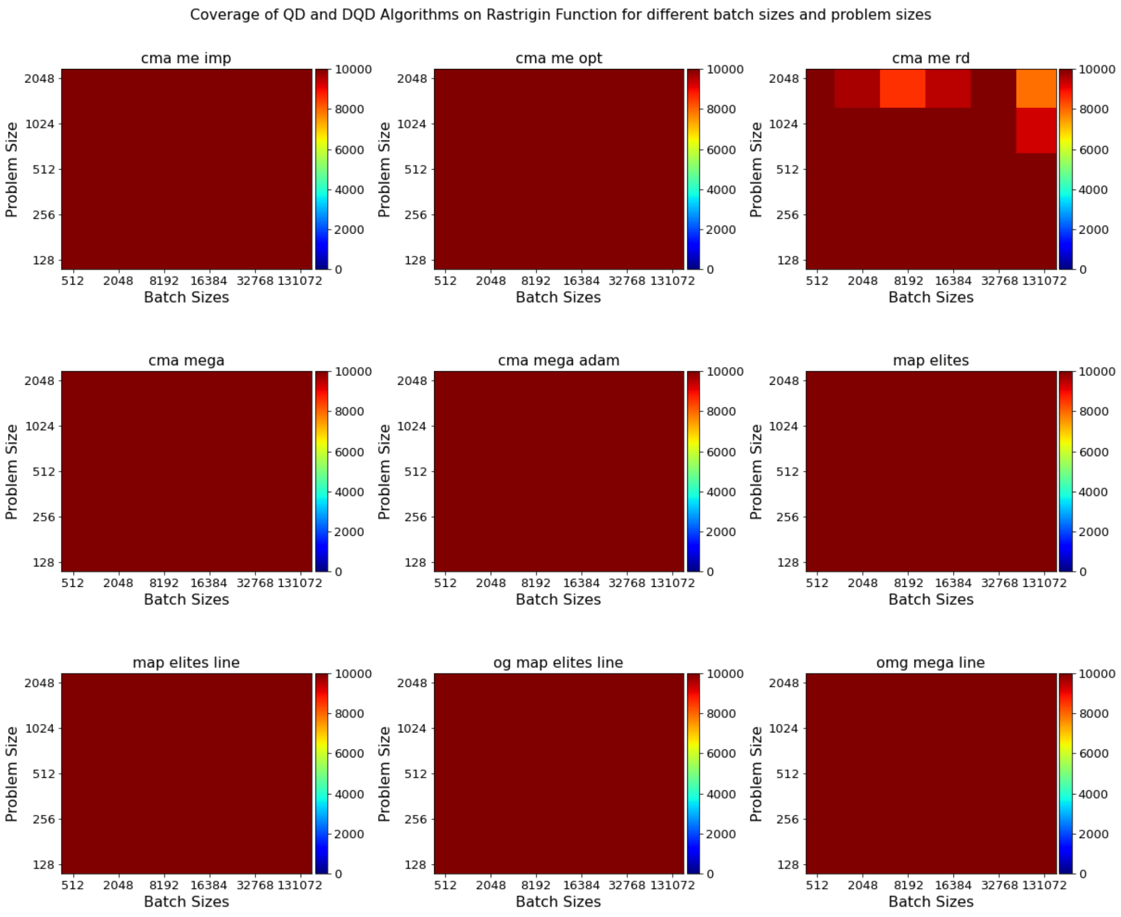


Fig C.6: Coverage of QD and DQD Algorithms on the Rastrigin with simple encoding problem for different batch sizes and problem sizes

C.2 Batch Size Variation Charts

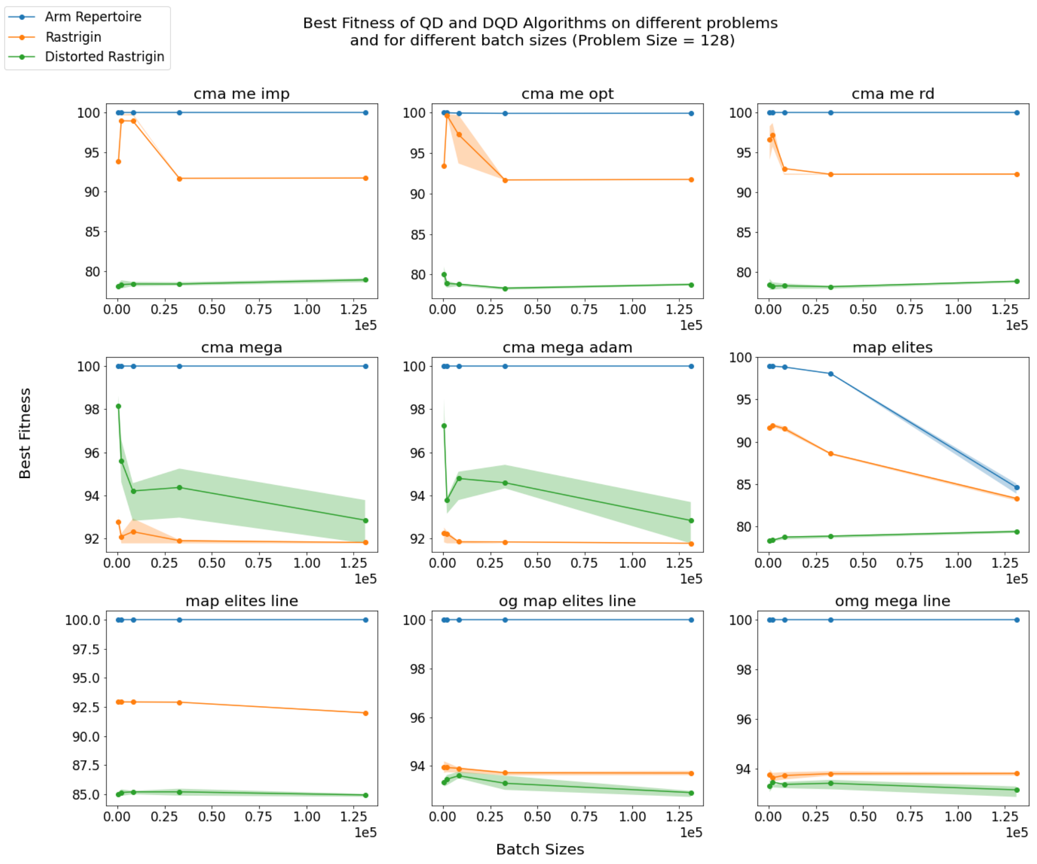


Fig C.7: Best Fitness of QD and DQD Algorithms on all the three problems for different batch sizes and constant problem size equal to 128

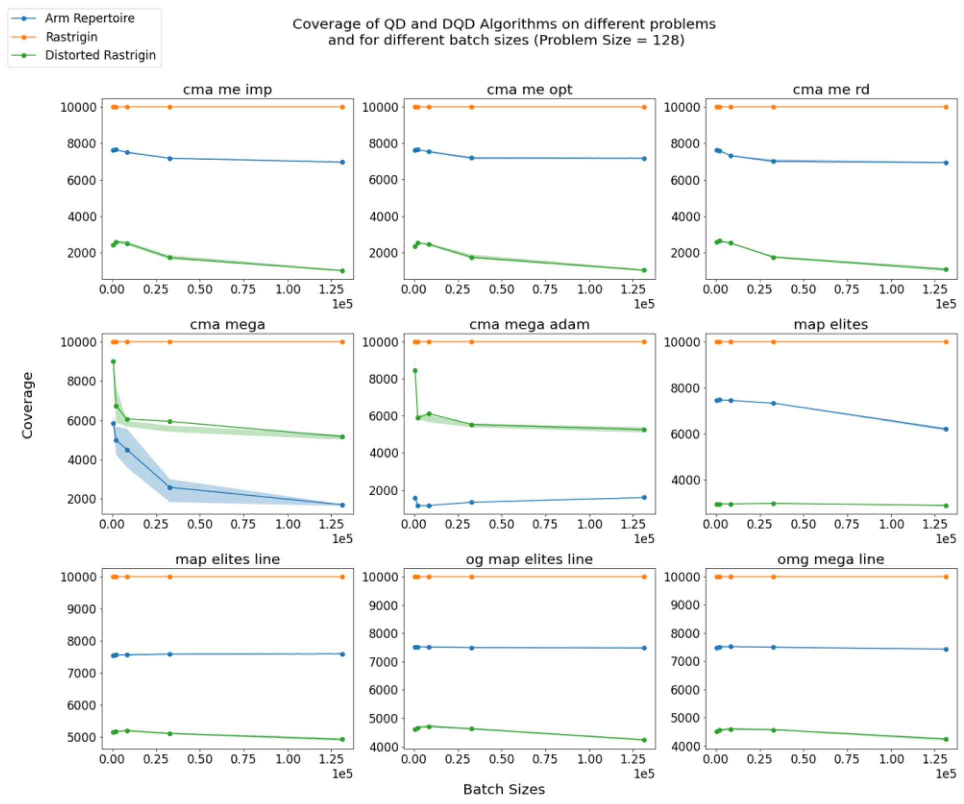


Fig C.8: Coverage of QD and DQD Algorithms on all the three problems for different batch sizes and constant problem size equal to 128

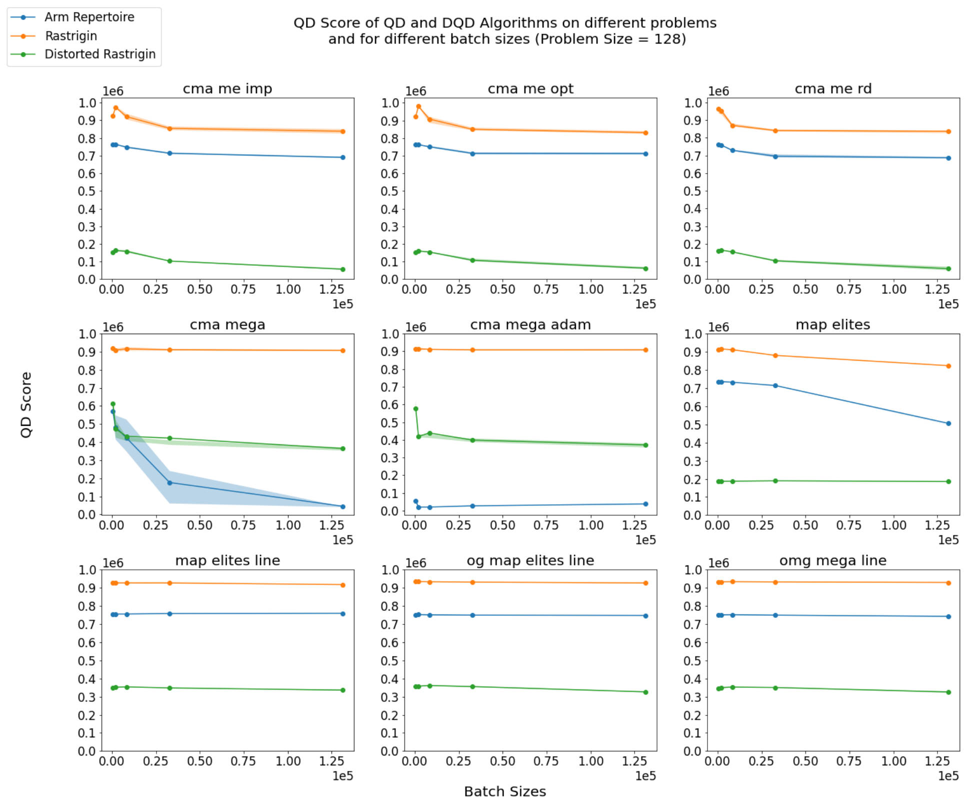


Fig C.9: QD Score of QD and DQD Algorithms on all the three problems for different batch sizes and constant problem size equal to 128

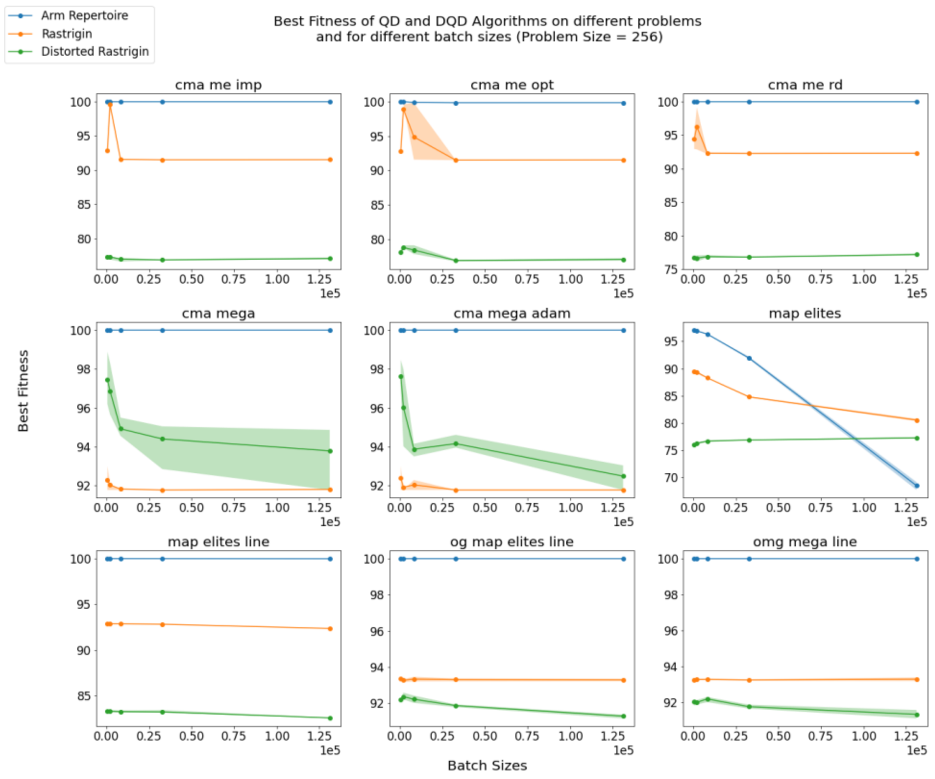


Fig C.10: Best Fitness of QD and DQD Algorithms on all the three problems for different batch sizes and constant problem size equal to 256

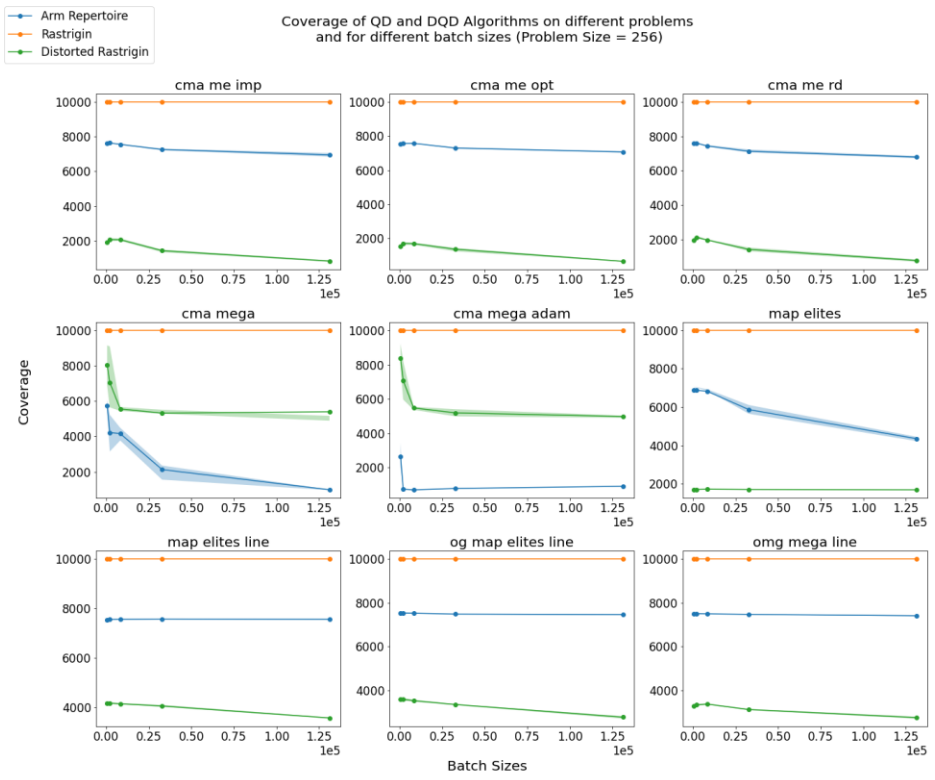


Fig C.11: Coverage of QD and DQD Algorithms on all the three problems for different batch sizes and constant problem size equal to 256

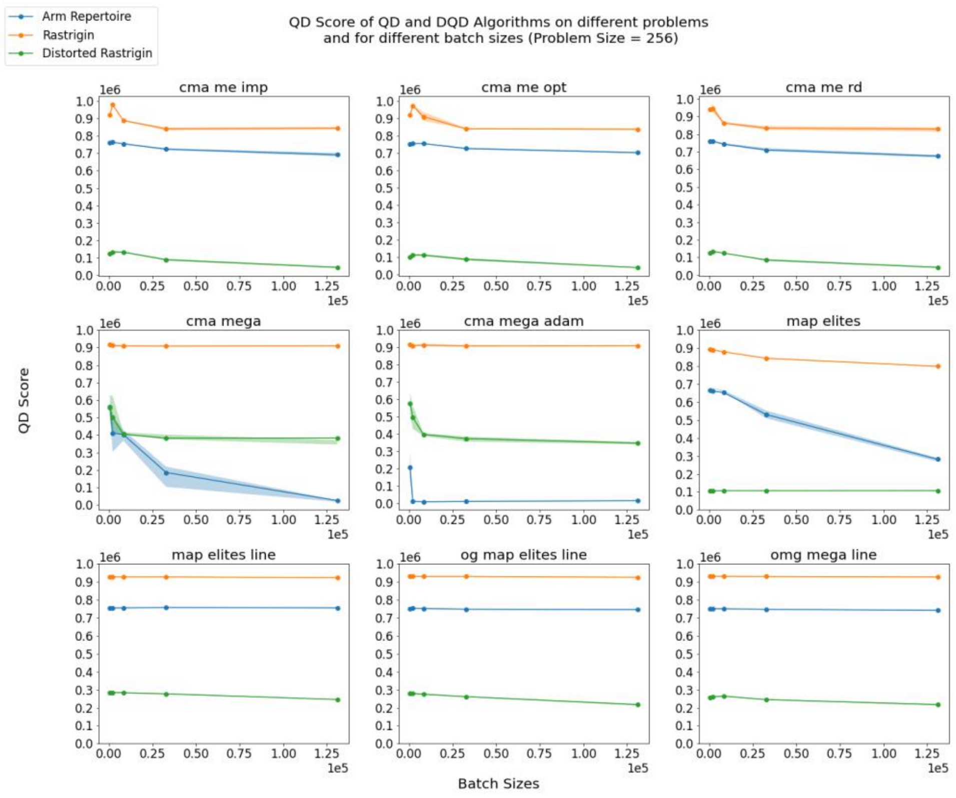


Fig C.12: QD Score of QD and DQD Algorithms on all the three problems for different batch sizes and constant problem size equal to 256

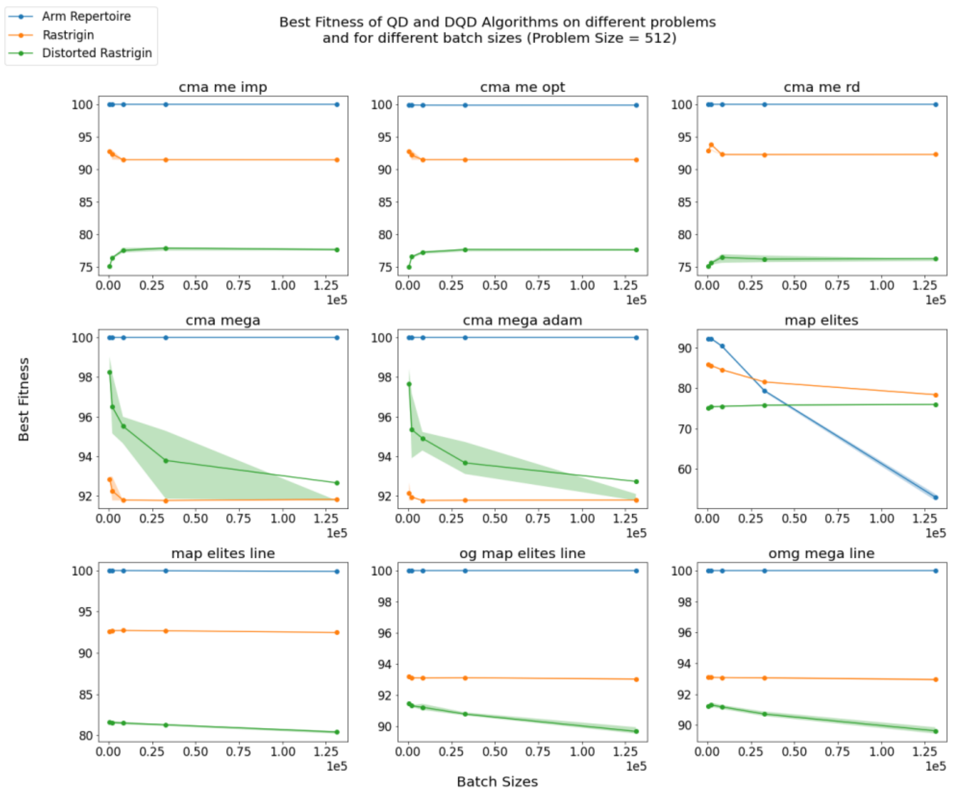


Fig C.13: Best Fitness of QD and DQD Algorithms on all the three problems for different batch sizes and constant problem size equal to 512

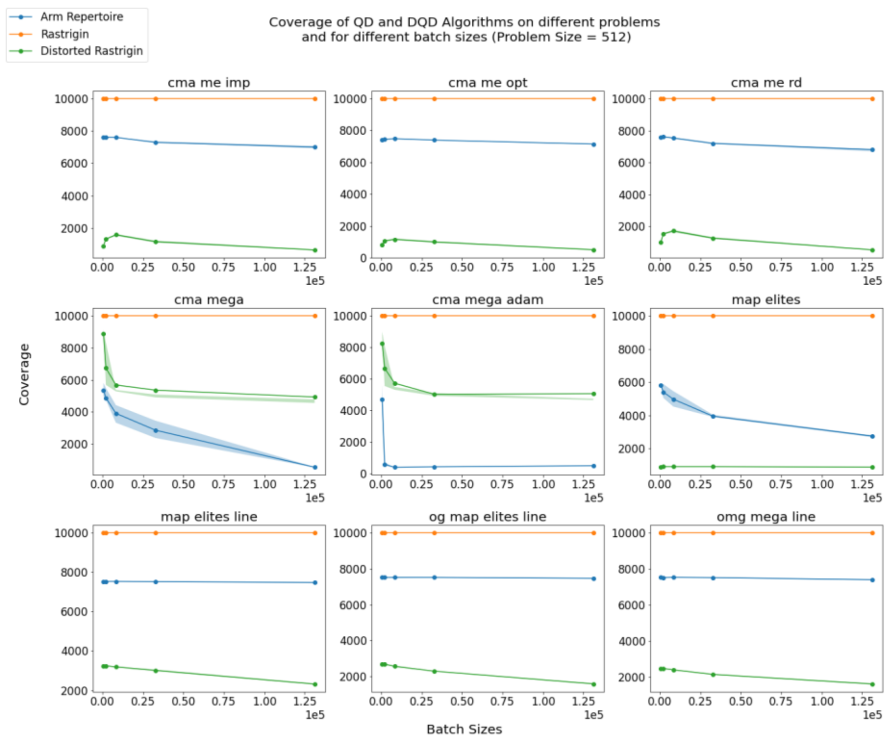


Fig C.14: Coverage of QD and DQD Algorithms on all the three problems for different batch sizes and constant problem size equal to 512

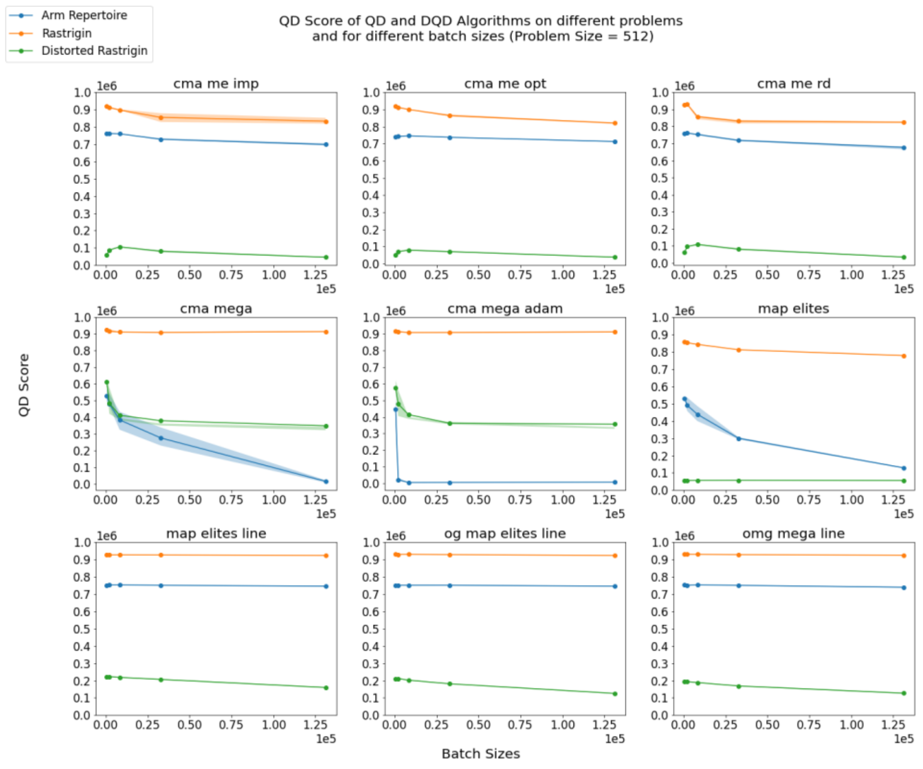


Fig C.15: QD Score of QD and DQD Algorithms on all the three problems for different batch sizes and constant problem size equal to 512

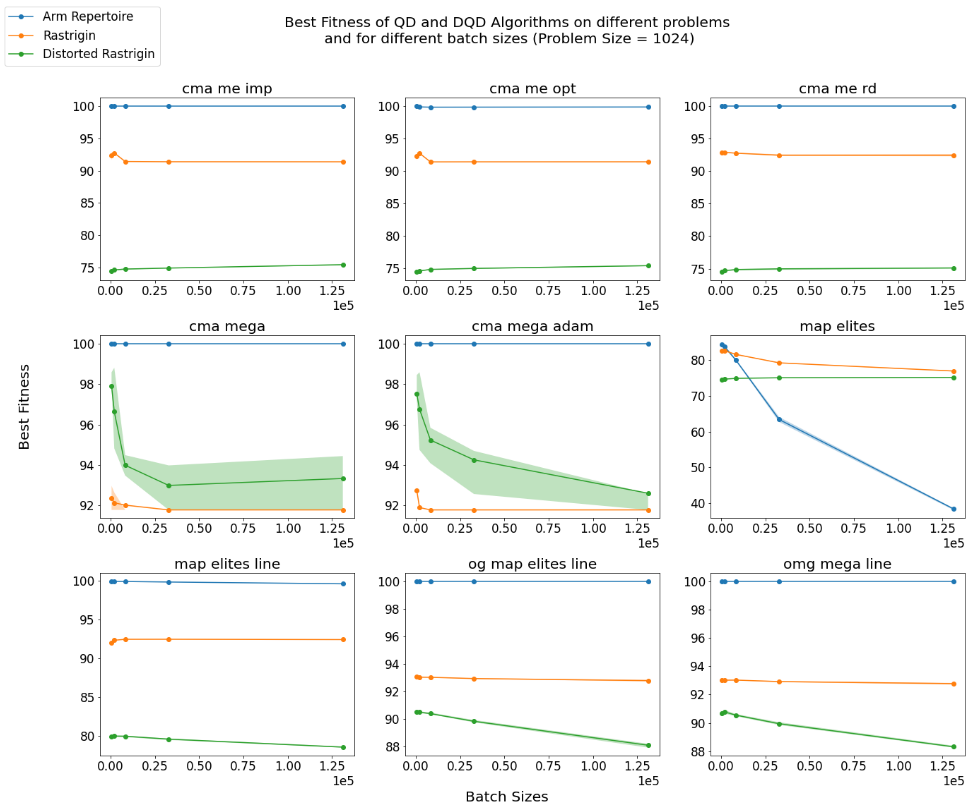


Fig C.16: Best Fitness of QD and DQD Algorithms on all the three problems for different batch sizes and constant problem size equal to 1024

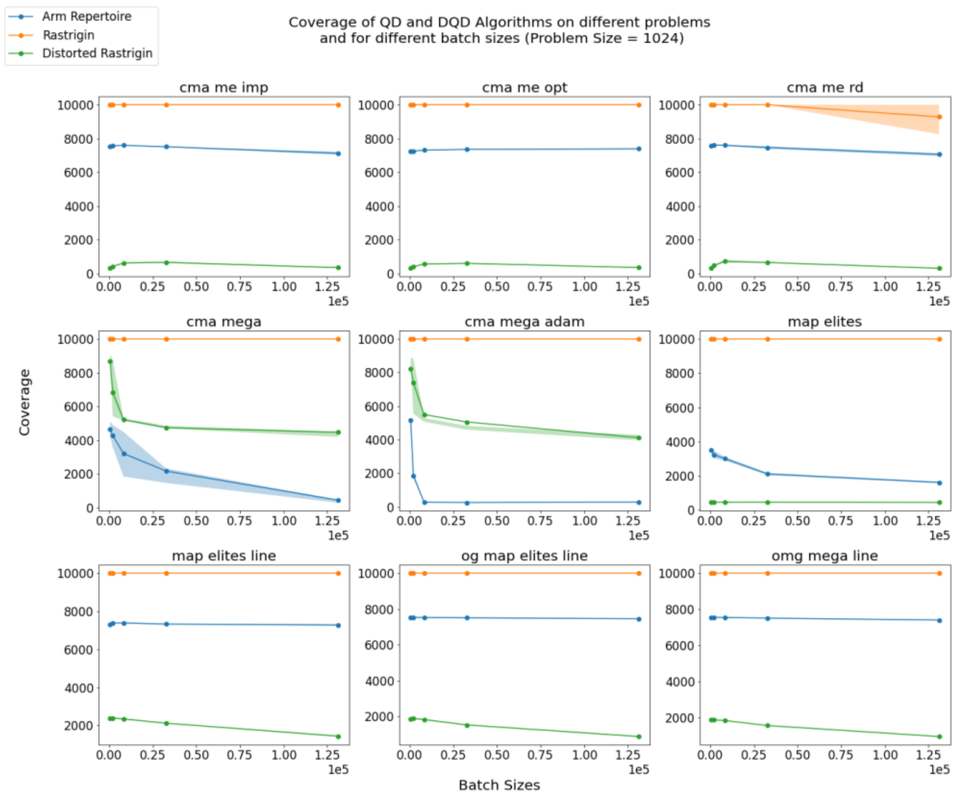


Fig C.17: Coverage of QD and DQD Algorithms on all the three problems for different batch sizes and constant problem size equal to 1024

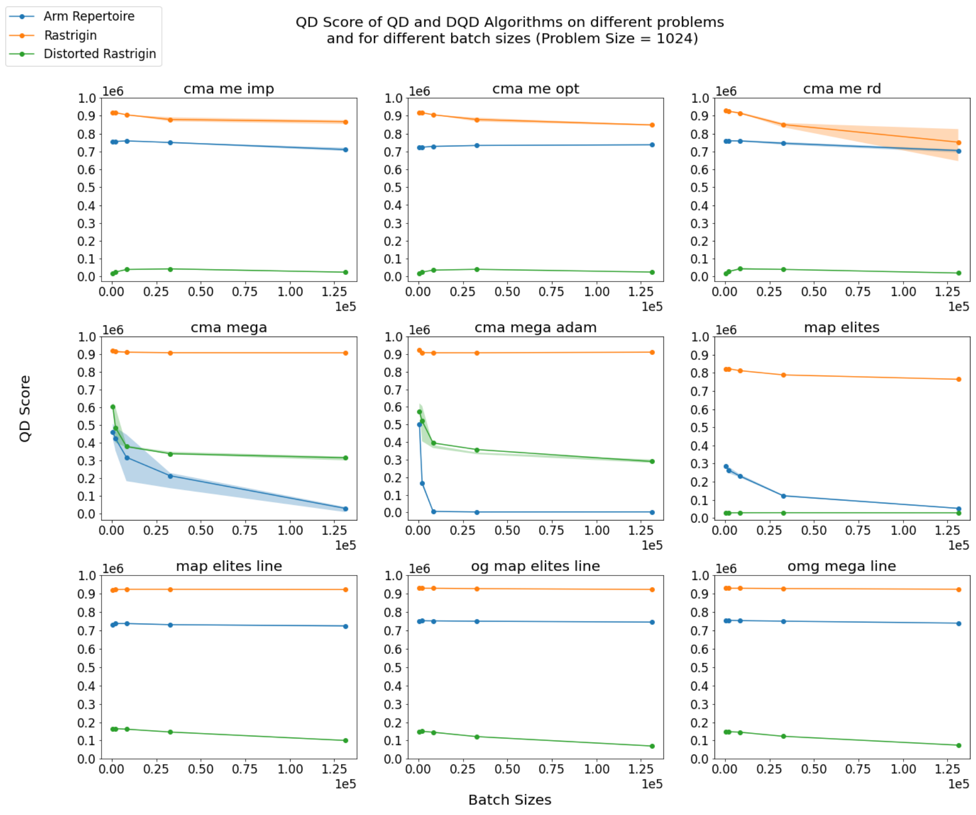


Fig C.18: QD Score of QD and DQD Algorithms on all the three problems for different batch sizes and constant problem size equal to 1024

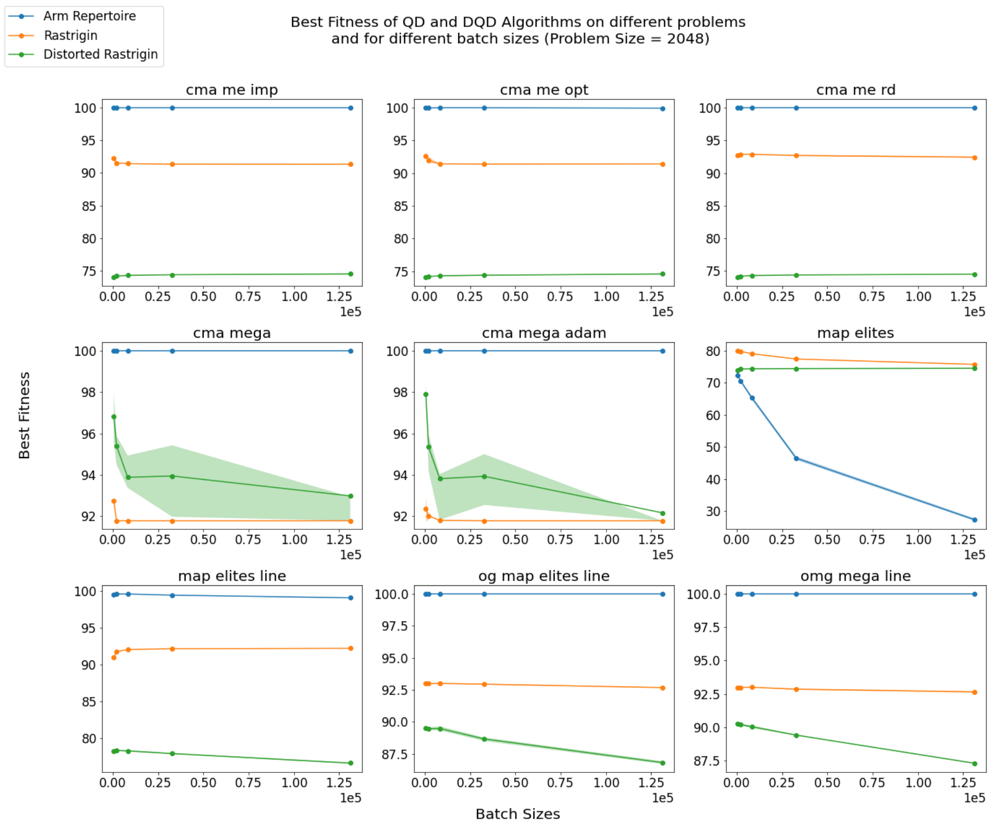


Fig C.19: Best Fitness of QD and DQD Algorithms on all the three problems for different batch sizes and constant problem size equal to 2048

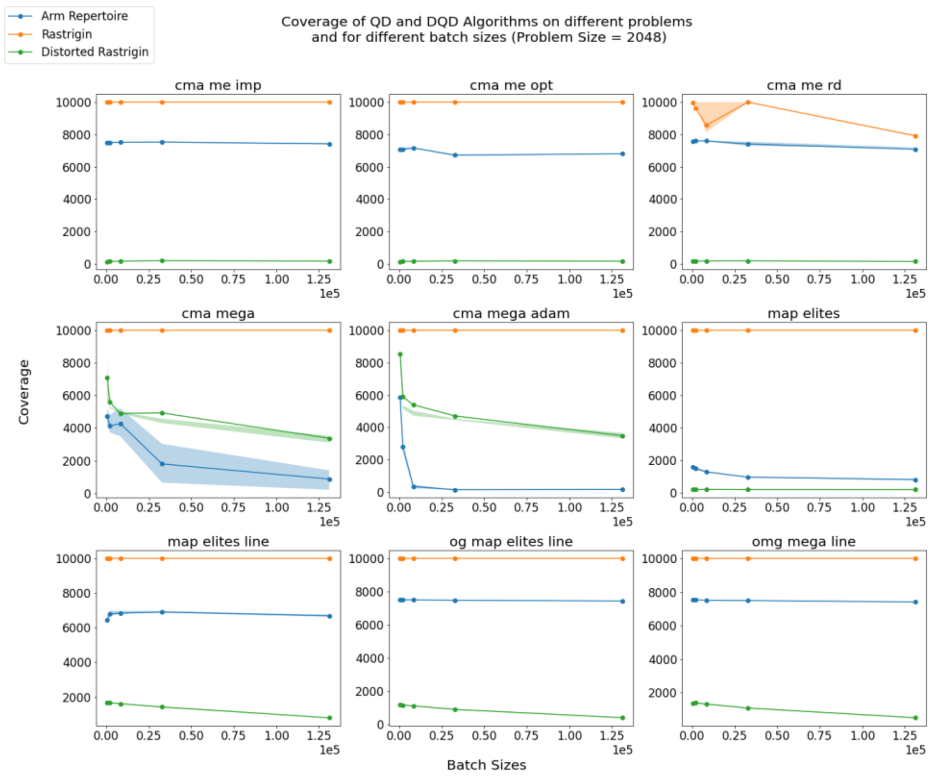


Fig C.20: Coverage of QD and DQD Algorithms on all the three problems for different batch sizes and constant problem size equal to 2048

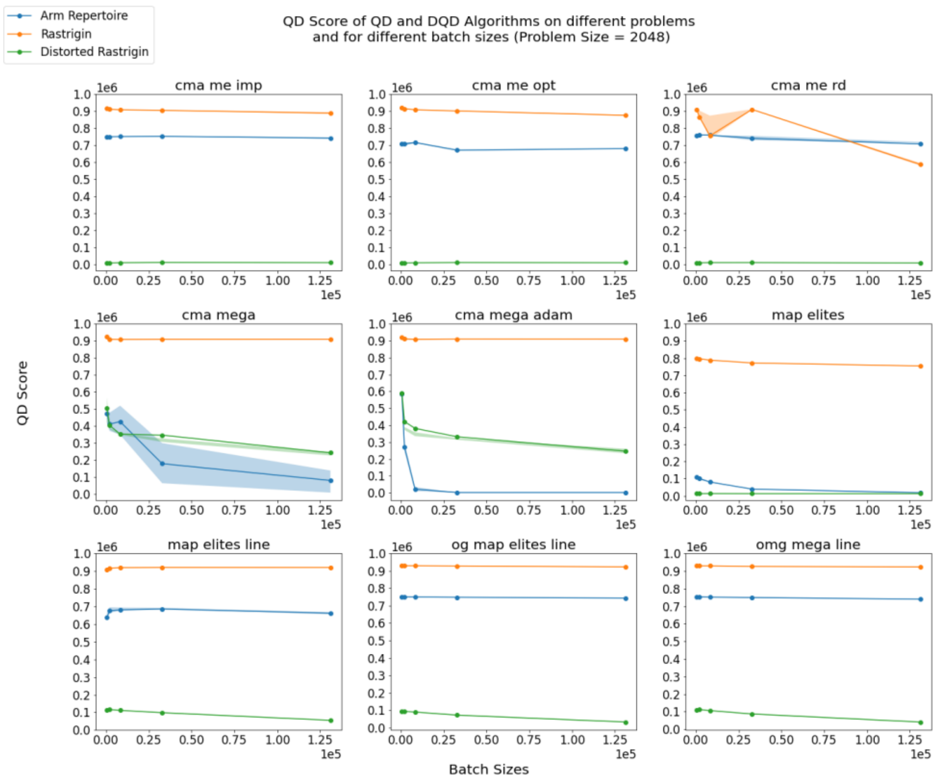


Fig C.21: QD Score of QD and DQD Algorithms on all the three problems for different batch sizes and constant problem size equal to 2048

C.3 Problem Size Variation Charts

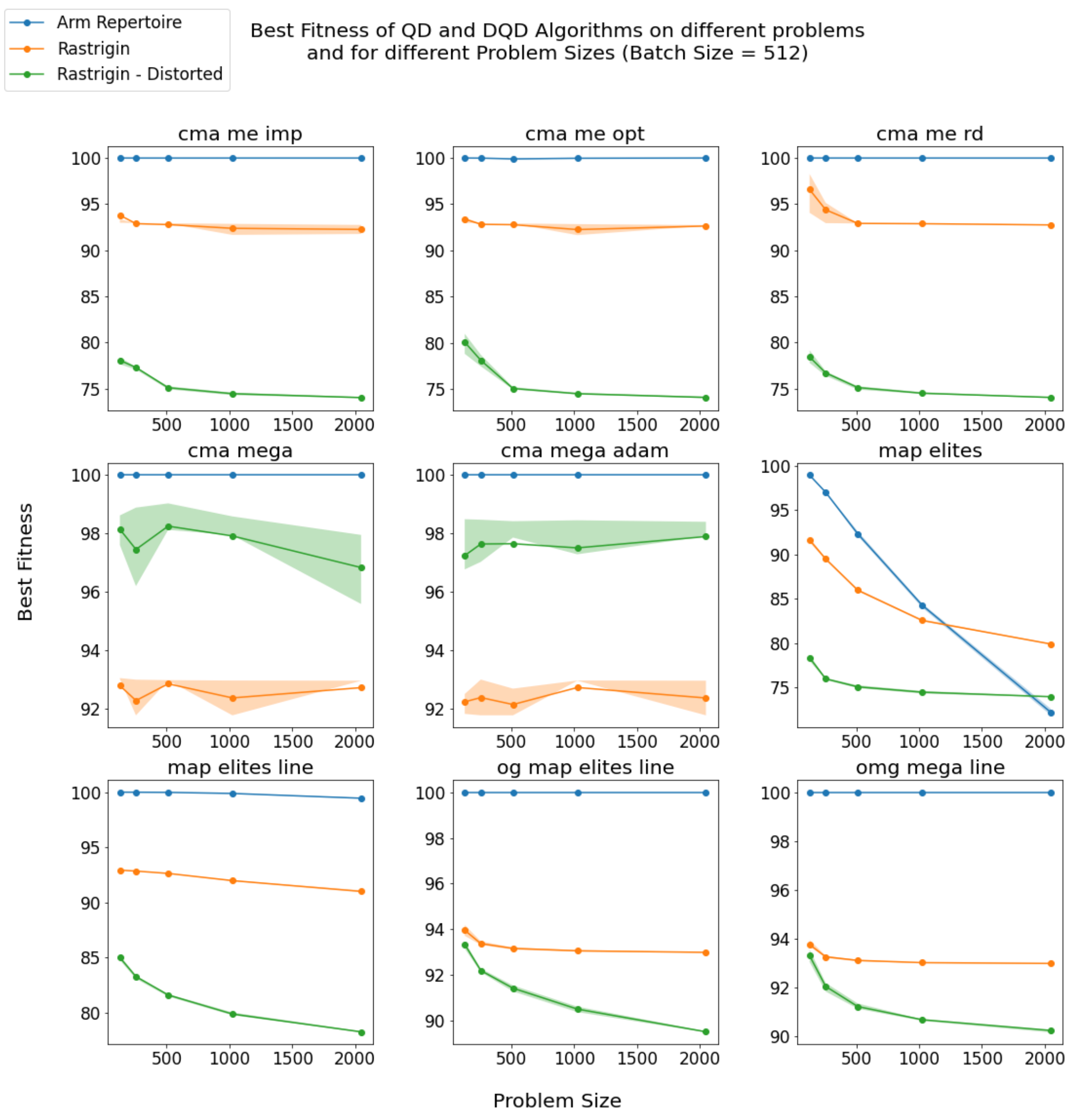


Fig C.22: Best Fitness of QD and DQD Algorithms on all the three problems for different problem sizes and constant batch size equal to 512

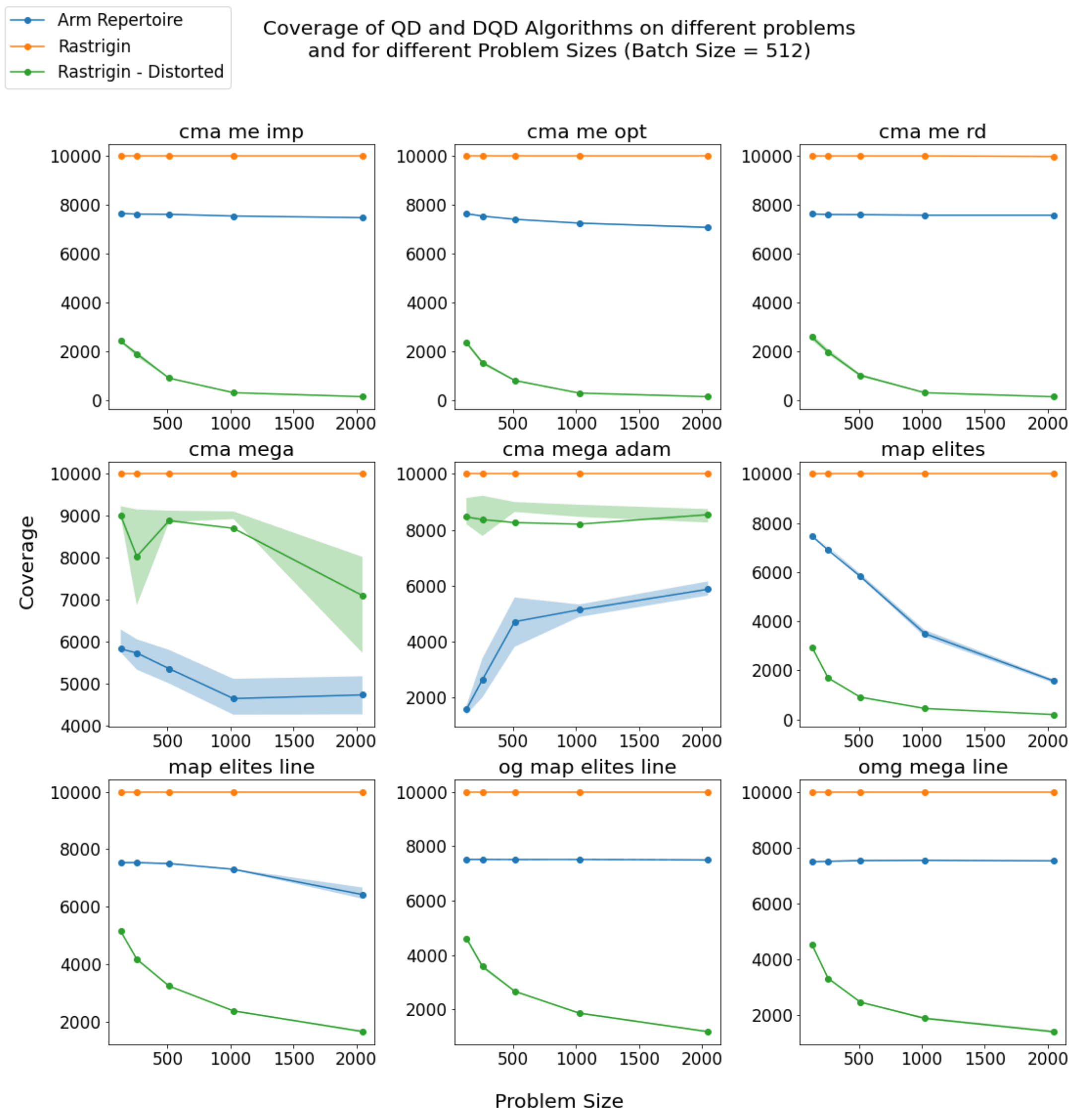


Fig C.23: Coverage of QD and DQD Algorithms on all the three problems for different problem sizes and constant batch size equal to 512

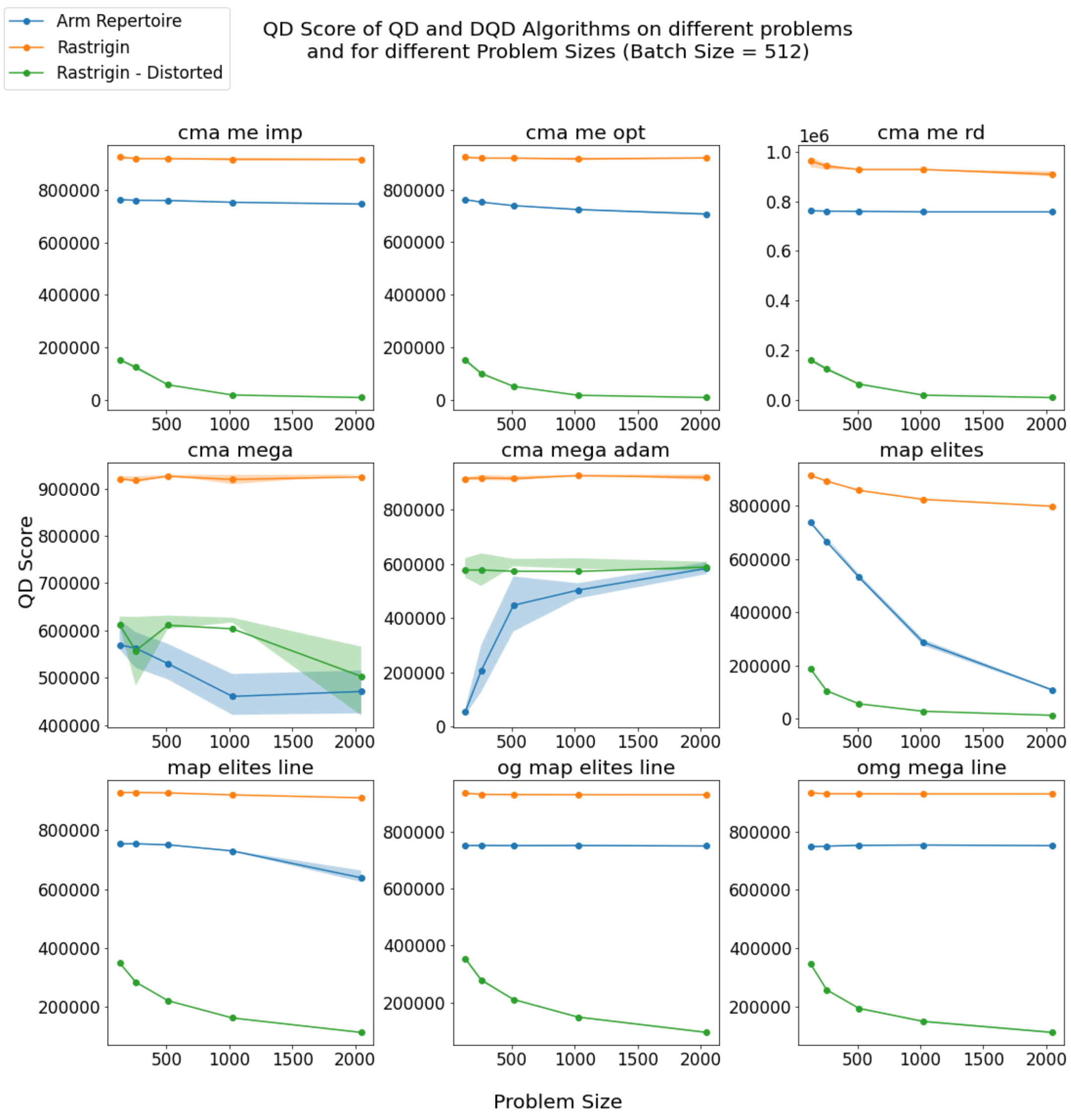


Fig C.24: QD Score of QD and DQD Algorithms on all the three problems for different problem sizes and constant batch size equal to 512

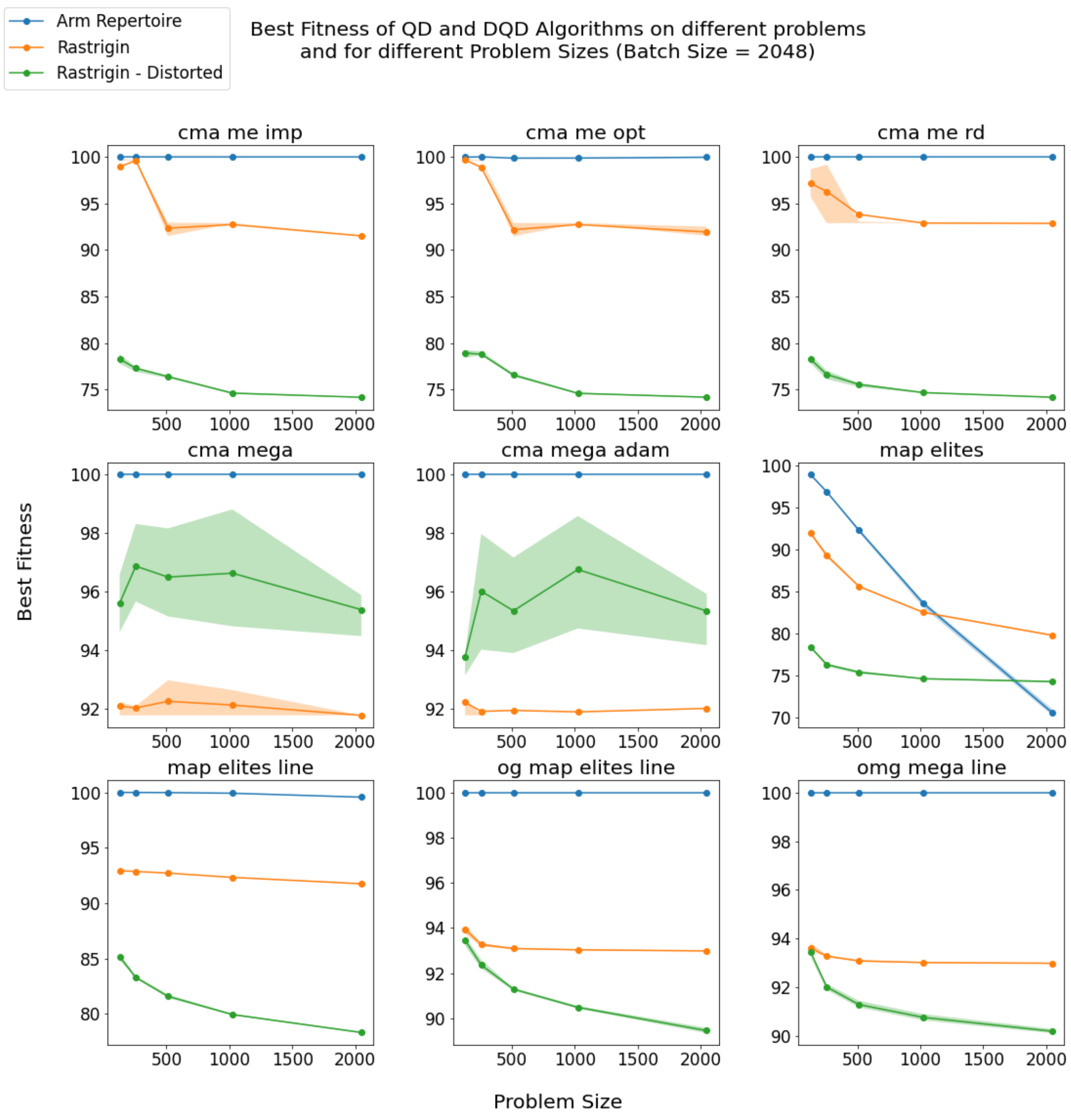


Fig C.25: Best Fitness of QD and DQD Algorithms on all the three problems for different problem sizes and constant batch size equal to 2048

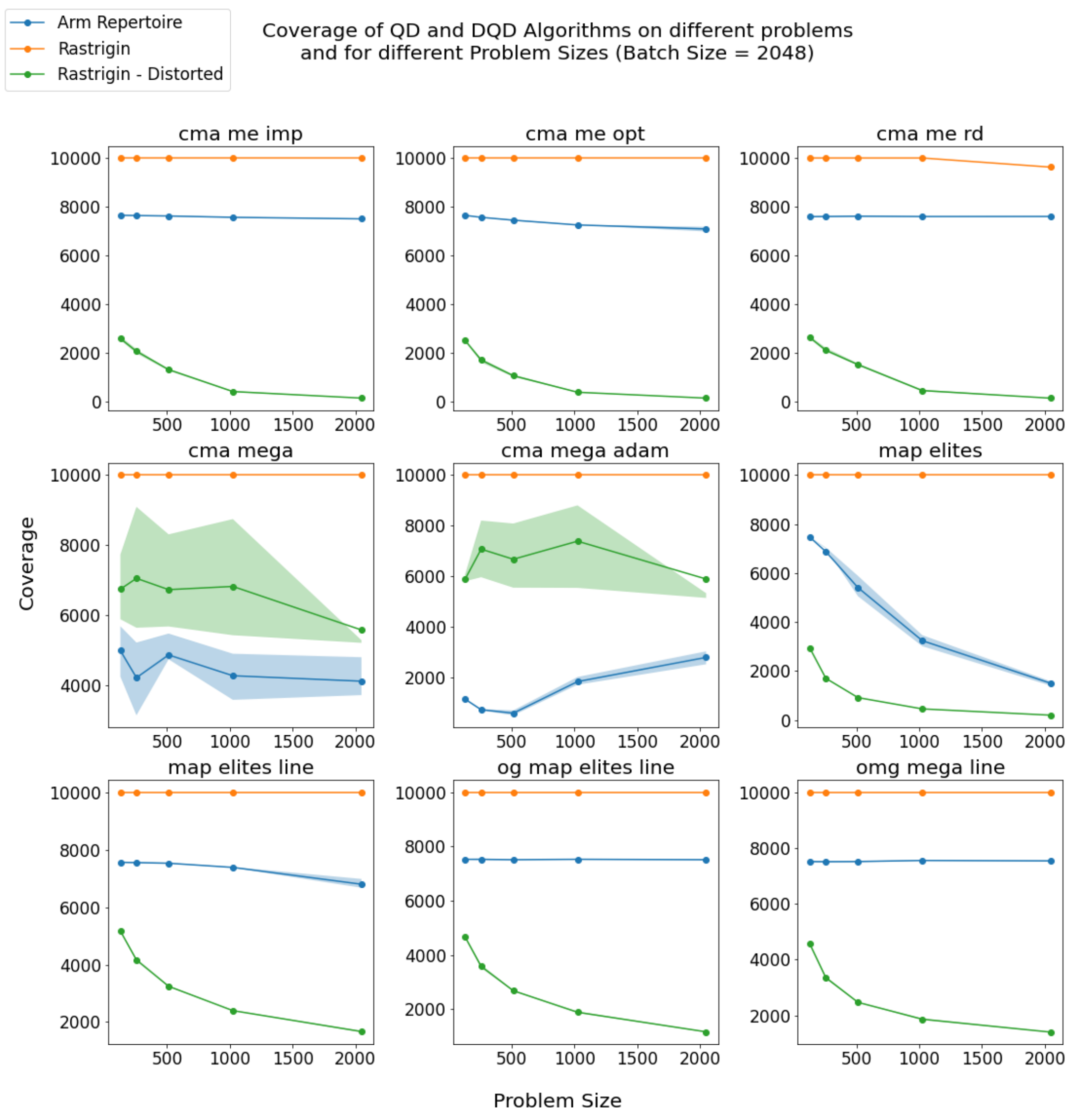


Fig C.26: Coverage of QD and DQD Algorithms on all the three problems for different problem sizes and constant batch size equal to 2048

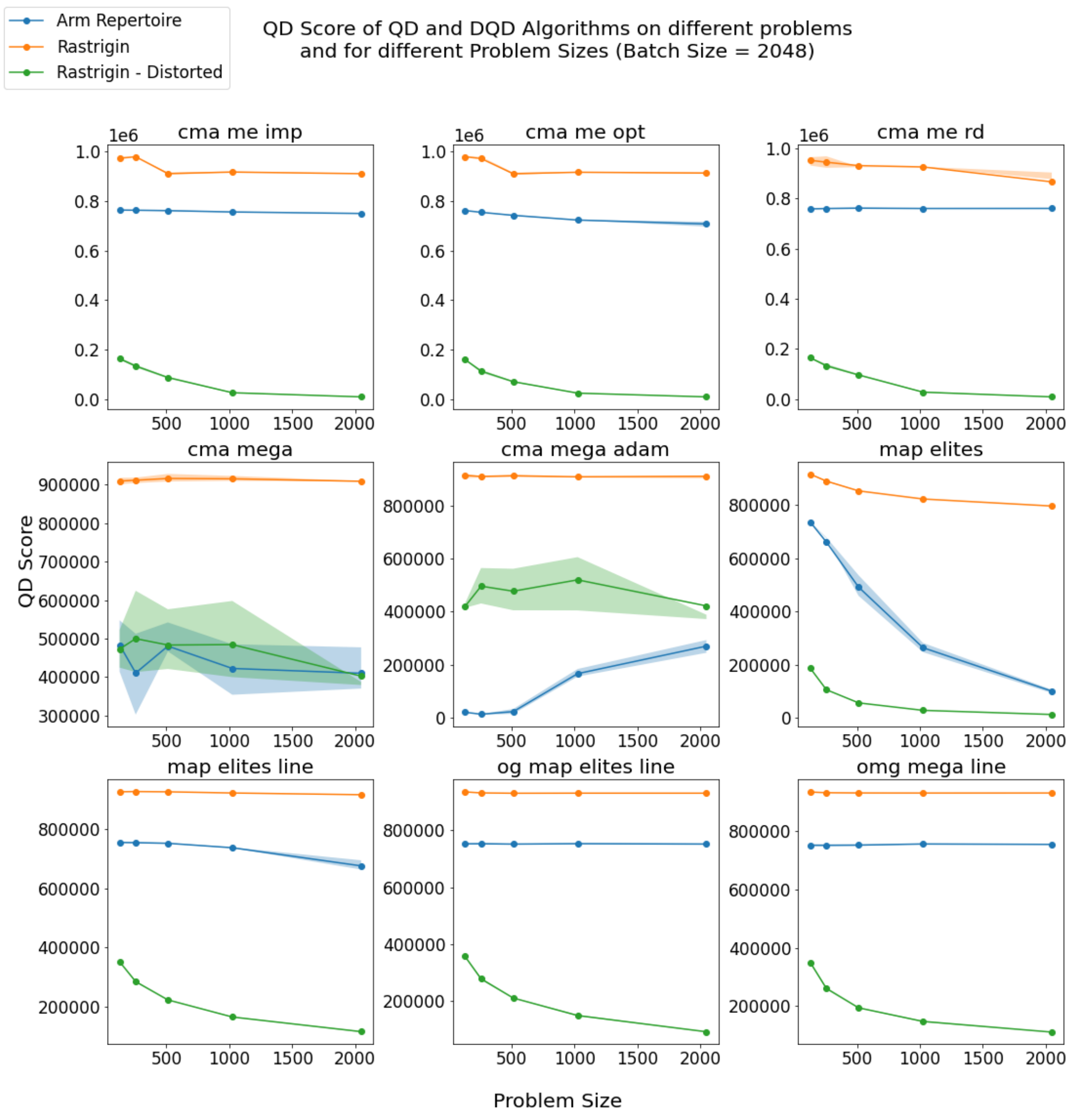


Fig C.27: QD Score of QD and DQD Algorithms on all the three problems for different problem sizes and constant batch size equal to 2048

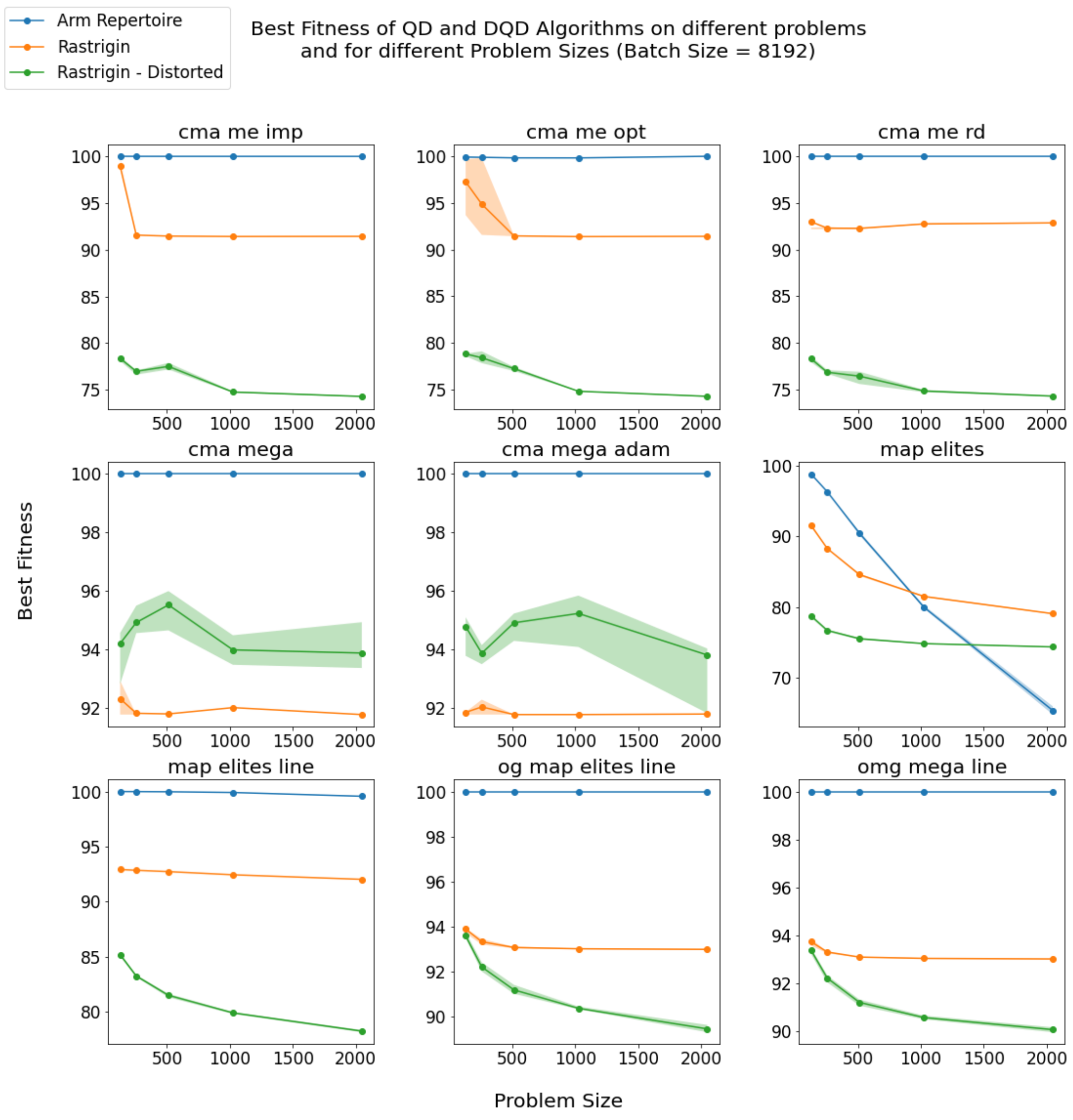


Fig C.28: Best Fitness of QD and DQD Algorithms on all the three problems for different problem sizes and constant batch size equal to 8192

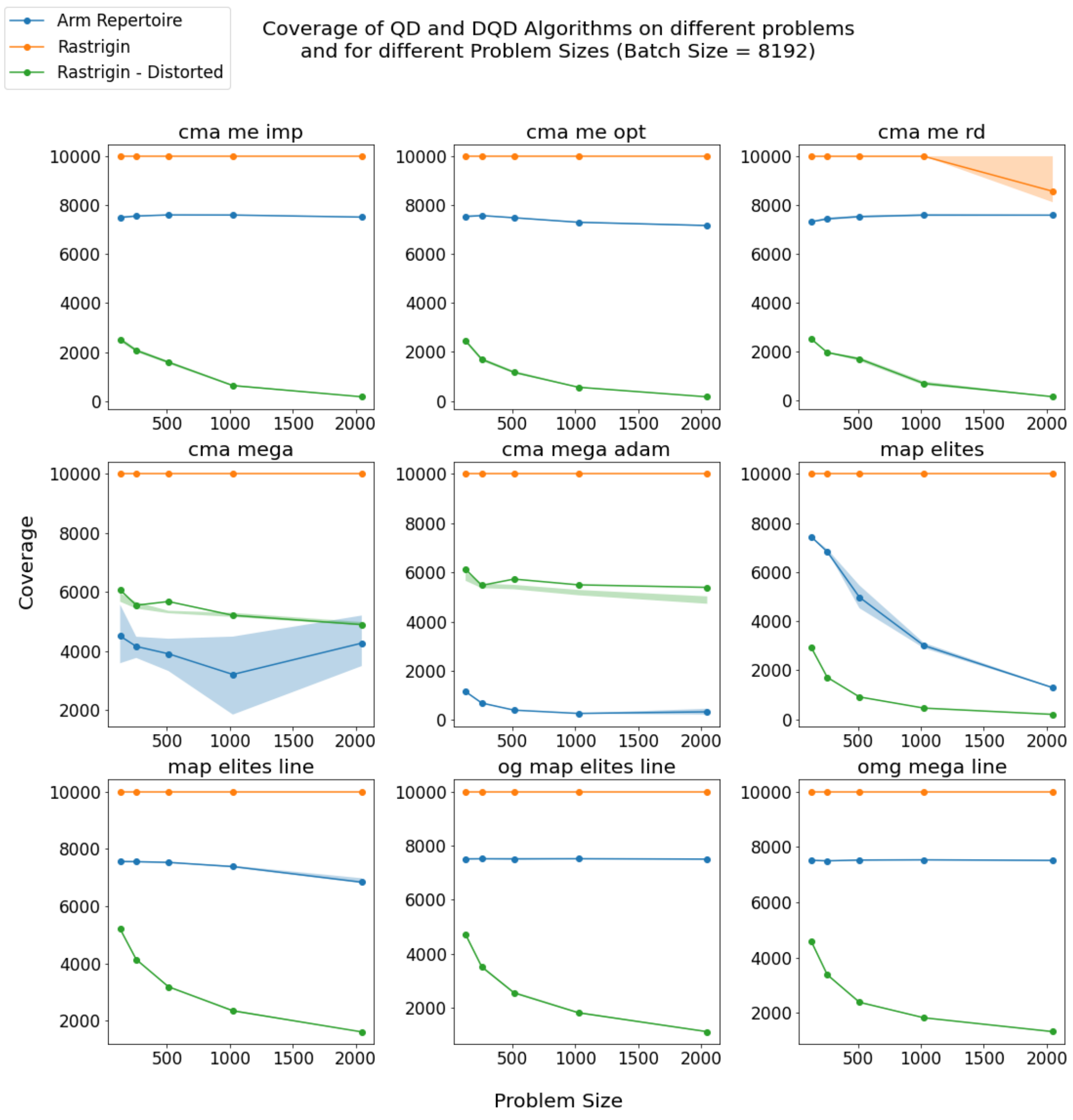


Fig C.29: Coverage of QD and DQD Algorithms on all the three problems for different problem sizes and constant batch size equal to 8192

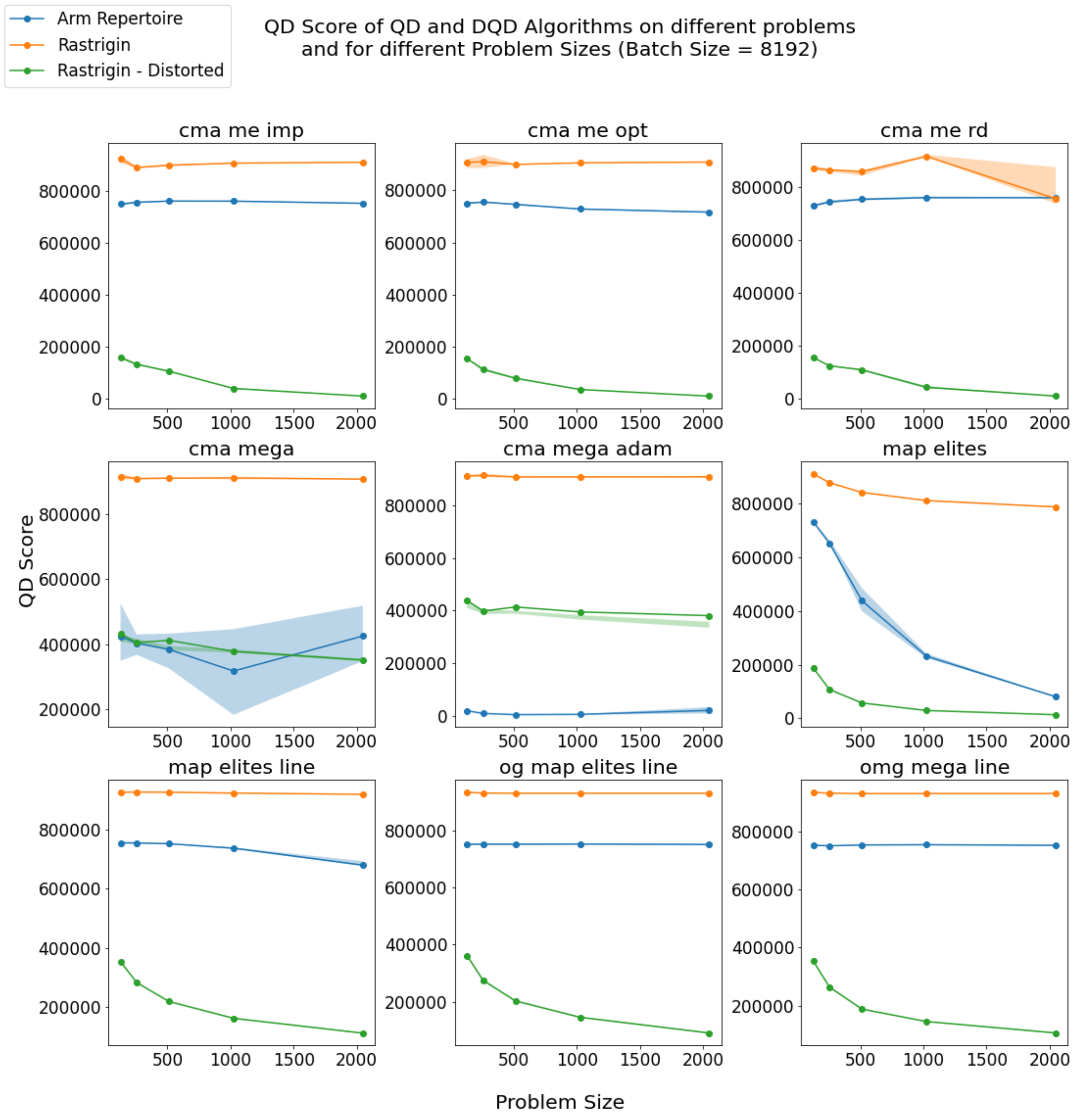


Fig C.30: QD Score of QD and DQD Algorithms on all the three problems for different problem sizes and constant batch size equal to 8192

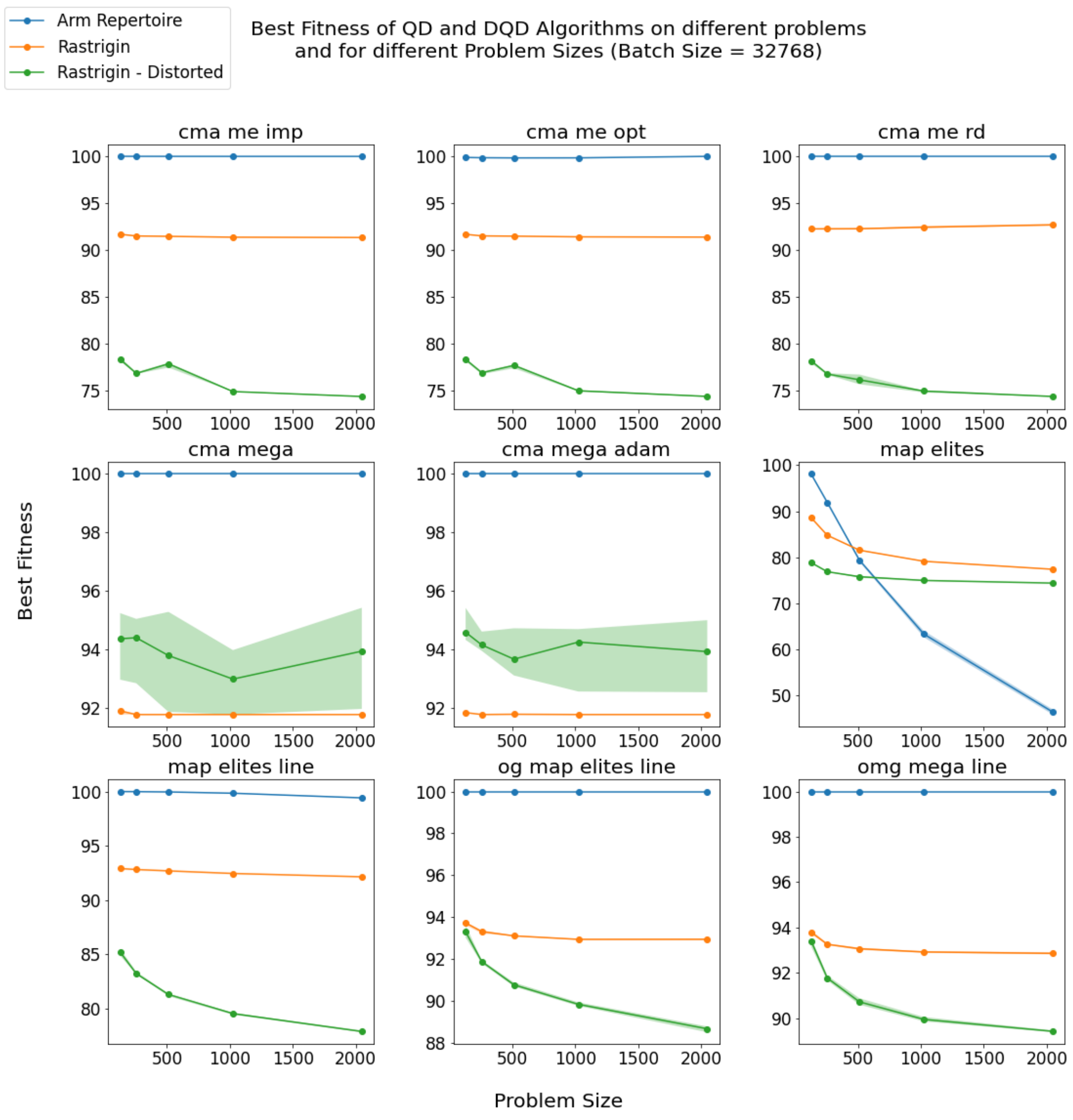


Fig C.31: Best Fitness of QD and DQD Algorithms on all the three problems for different problem sizes and constant batch size equal to 32768

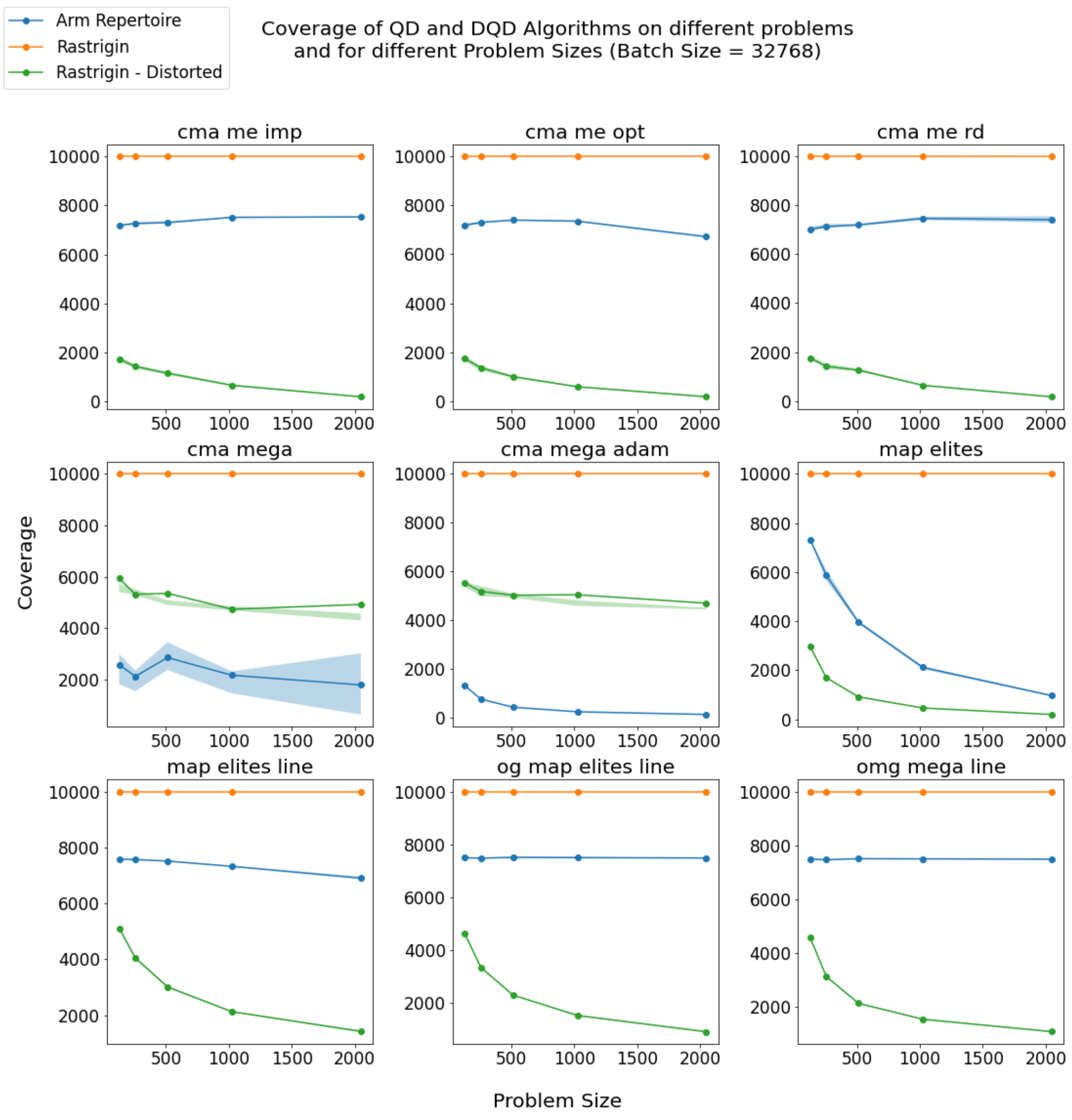


Fig C.32: Coverage of QD and DQD Algorithms on all the three problems for different problem sizes and constant batch size equal to 32768

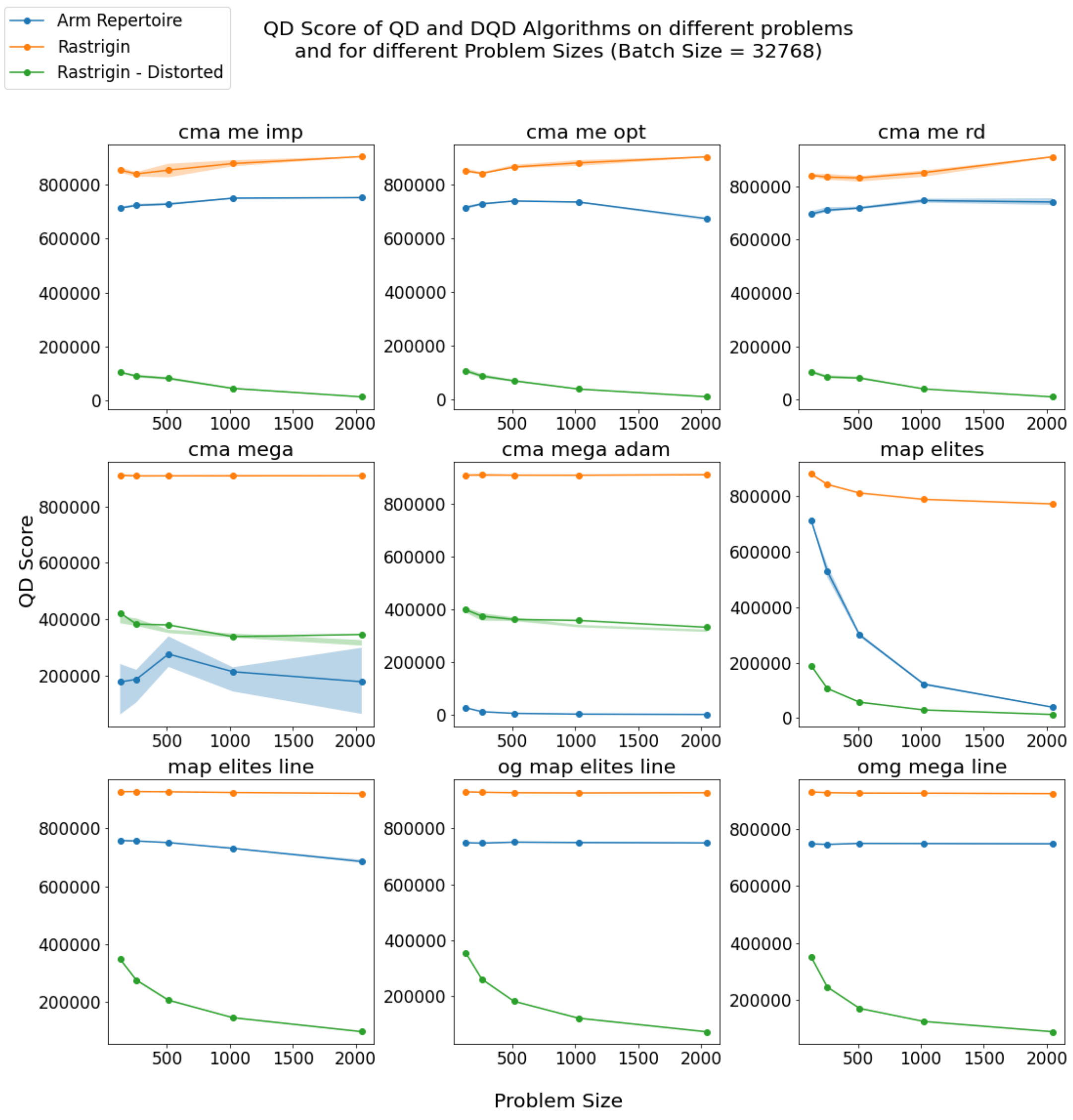


Fig C.33: QD Score of QD and DQD Algorithms on all the three problems for different problem sizes and constant batch size equal to 32768

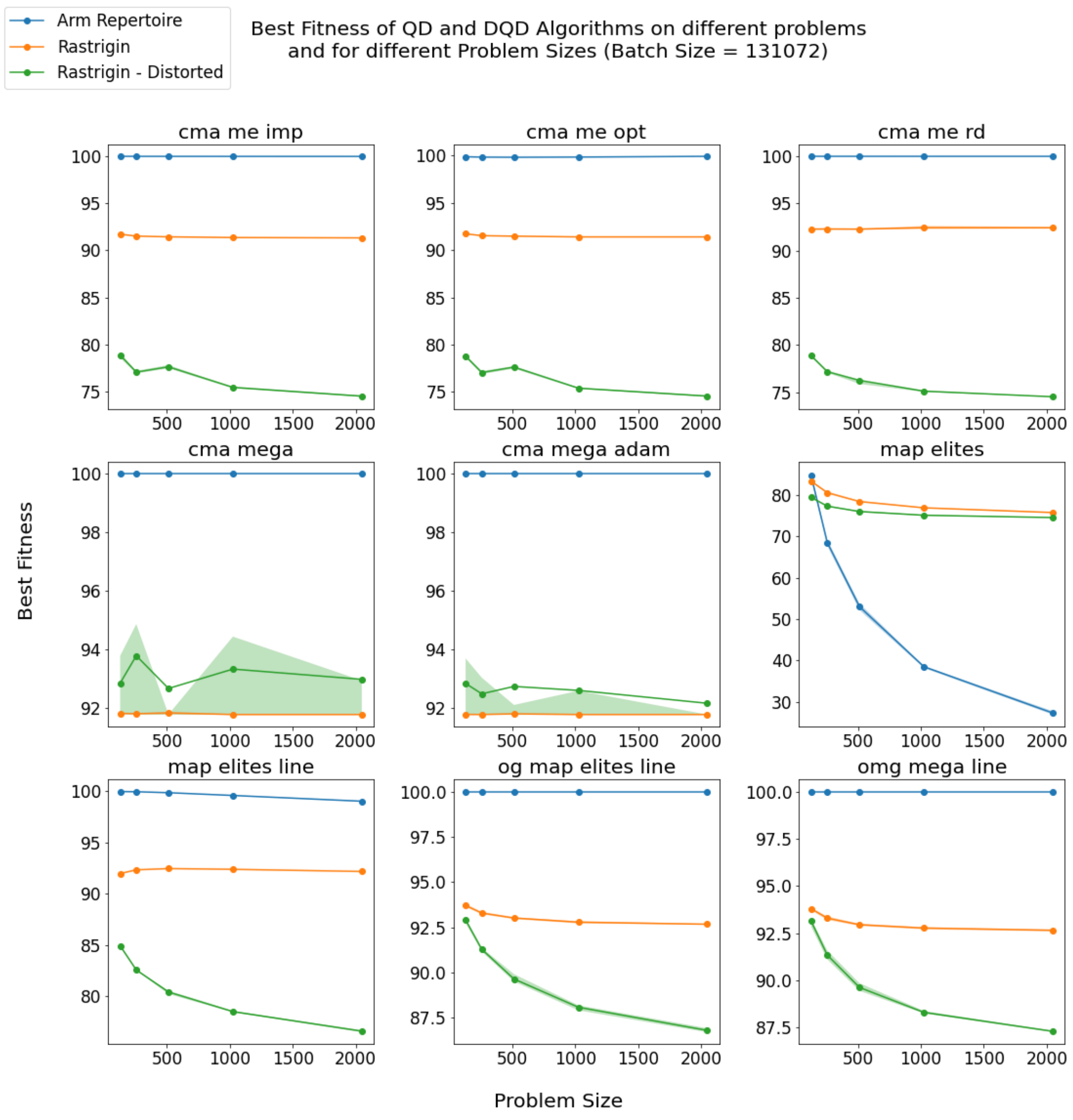


Fig C.34: Best Fitness of QD and DQD Algorithms on all the three problems for different problem sizes and constant batch size equal to 131072

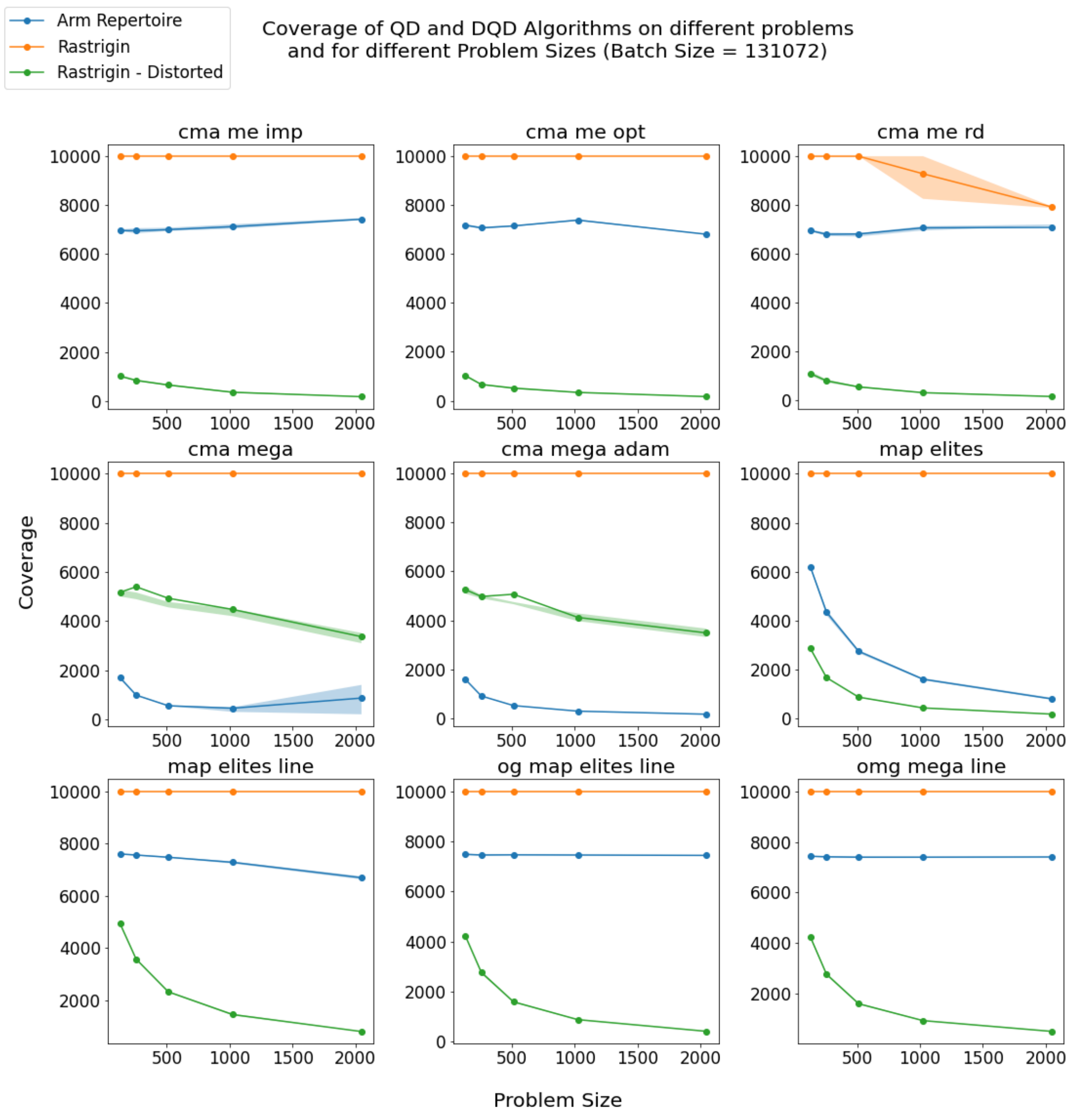


Fig C.35: Coverage of QD and DQD Algorithms on all the three problems for different problem sizes and constant batch size equal to 131072

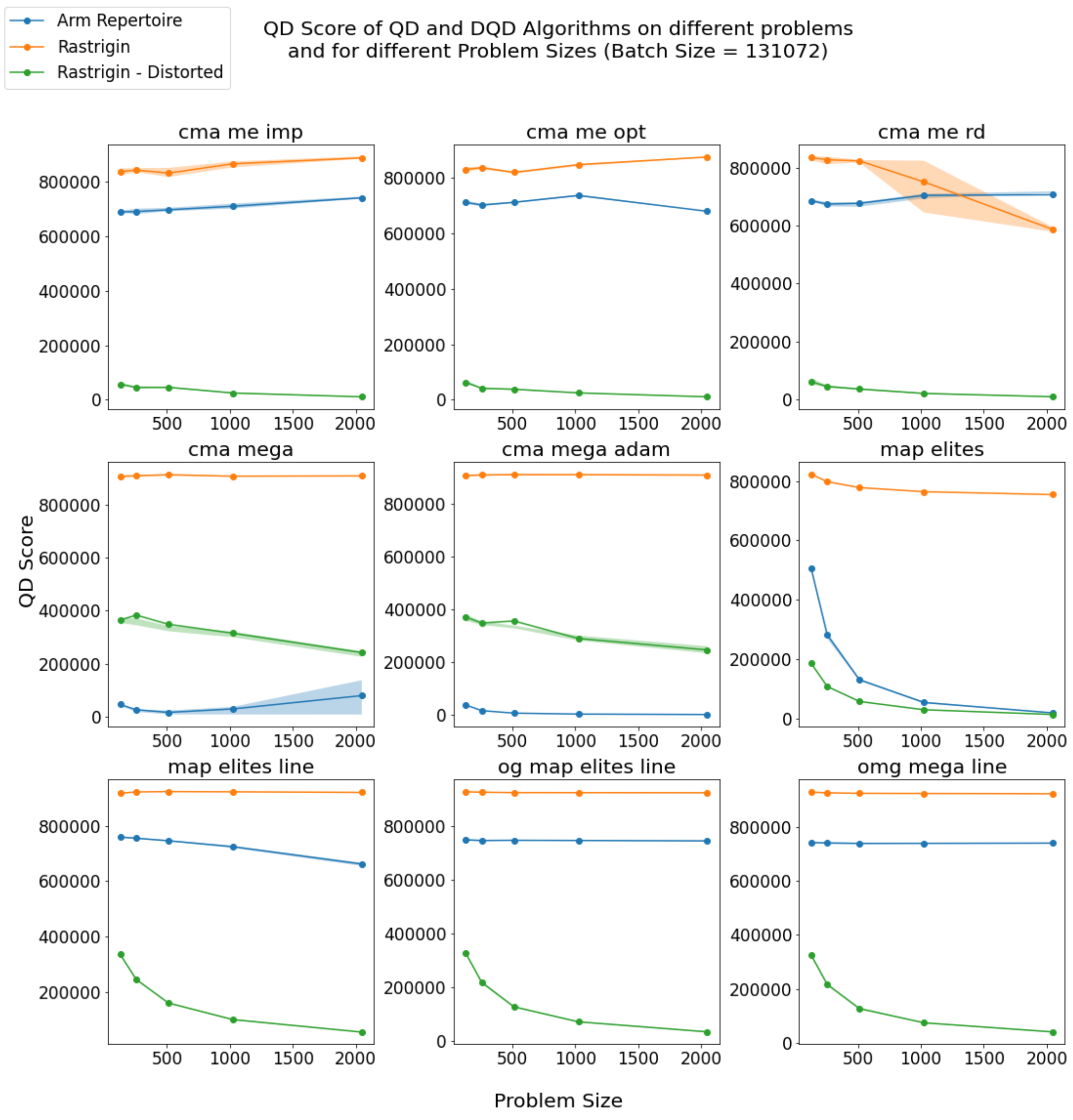


Fig C.36: QD Score of QD and DQD Algorithms on all the three problems for different problem sizes and constant batch size equal to 131072

C.4 Runtime - Problem Size Variation Charts

C.4.1 Arm Repertoire

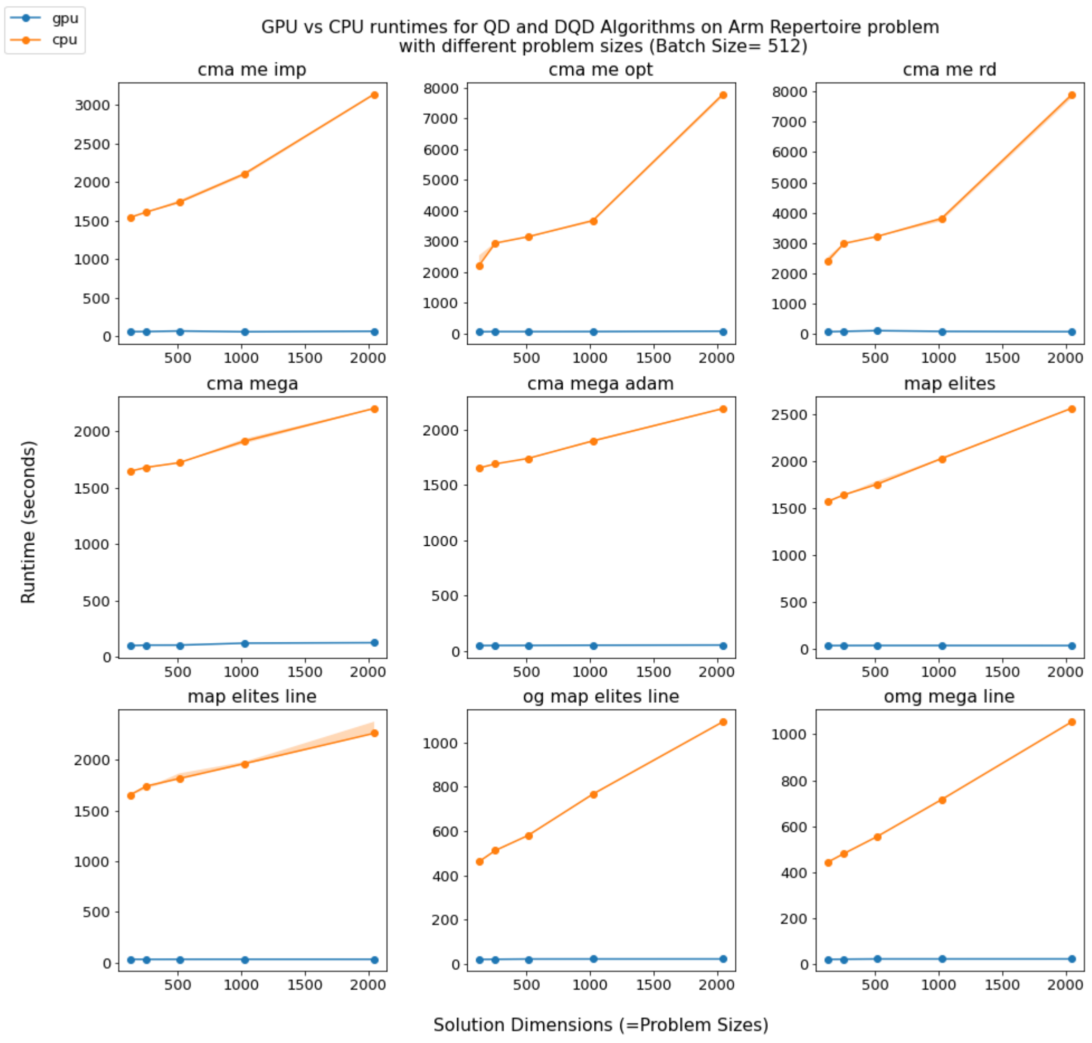


Fig C.37: Runtime of QD and DQD Algorithms on all the three problems for different problem sizes and constant batch size equal to 512

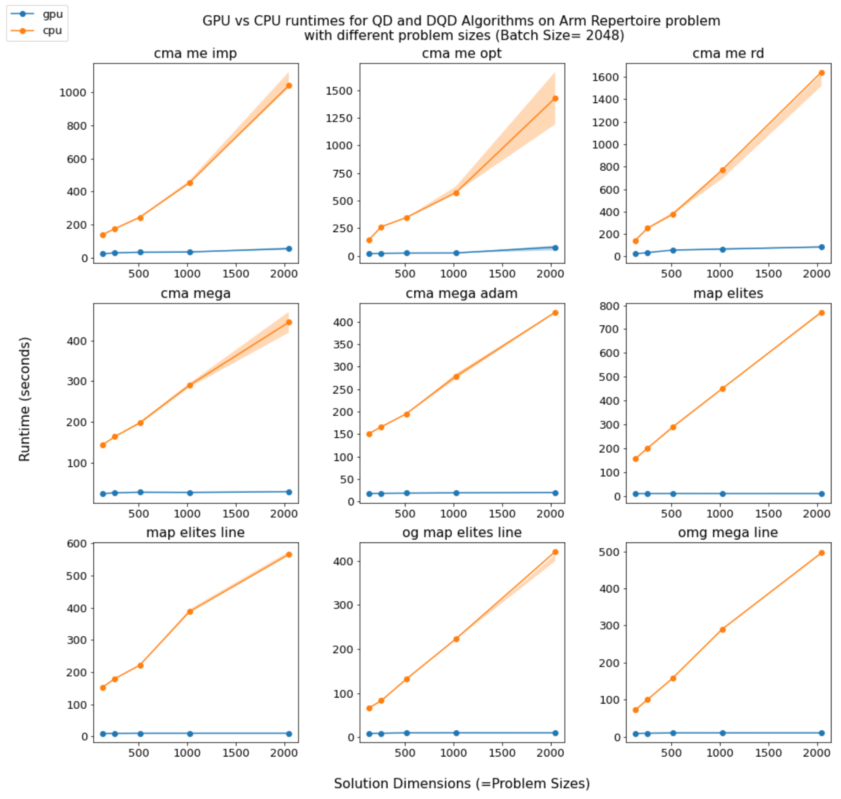


Fig C.38: Runtime of QD and DQD Algorithms on all the three problems for different problem sizes and constant batch size equal to 2048

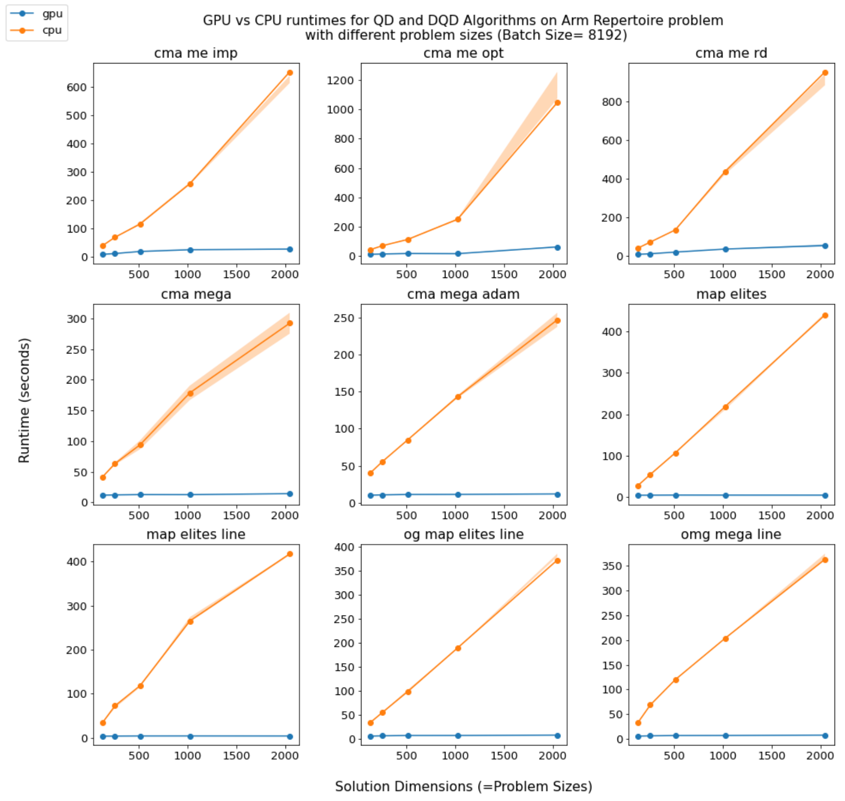


Fig C.39: Runtime of QD and DQD Algorithms on all the three problems for different problem sizes and constant batch size equal to 8192

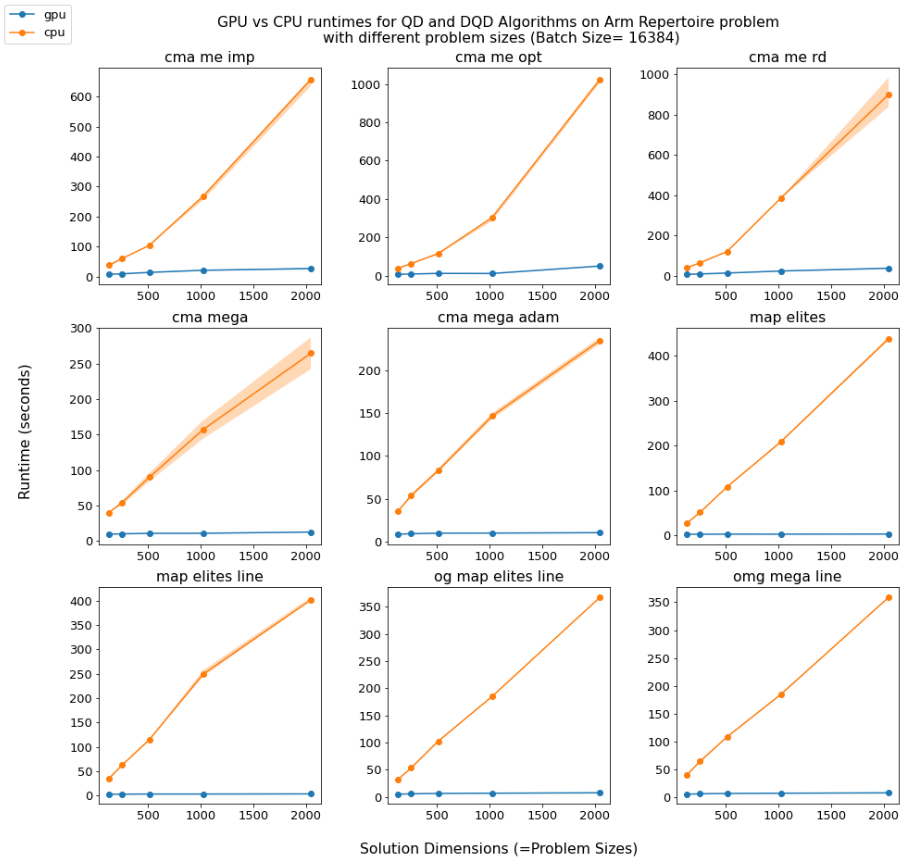


Fig C.40: Runtime of QD and DQD Algorithms on all the three problems for different problem sizes and constant batch size equal to 16384

C.4.2 Rastrigin with Distorted Behavioral Space

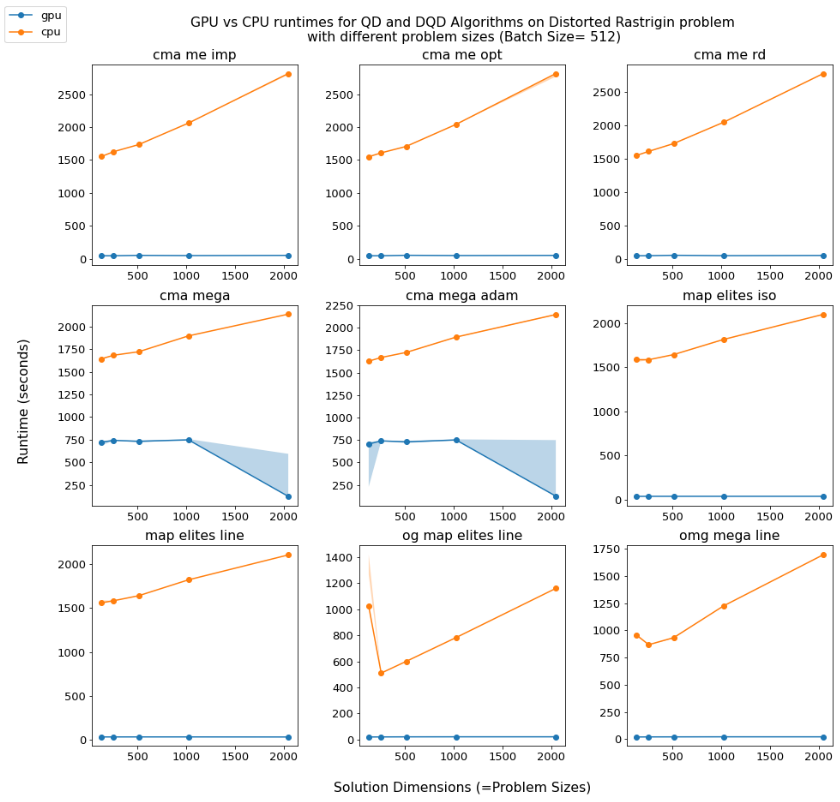


Fig C.41: Runtime of QD and DQD Algorithms on all the three problems for different problem sizes and constant batch size equal to 512

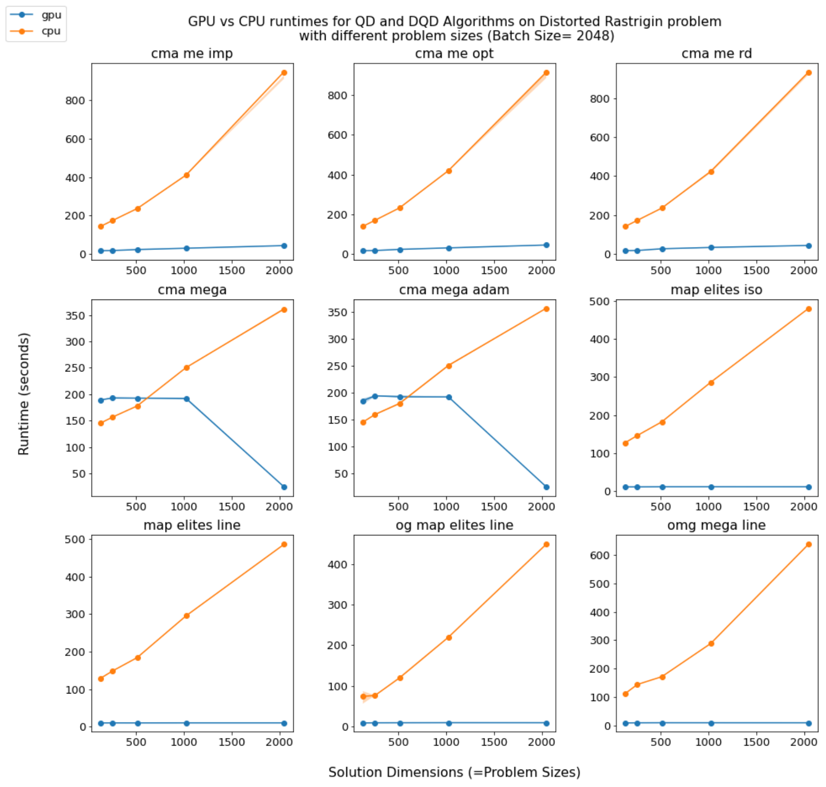


Fig C.42: Runtime of QD and DQD Algorithms on all the three problems for different problem sizes and constant batch size equal to 2048

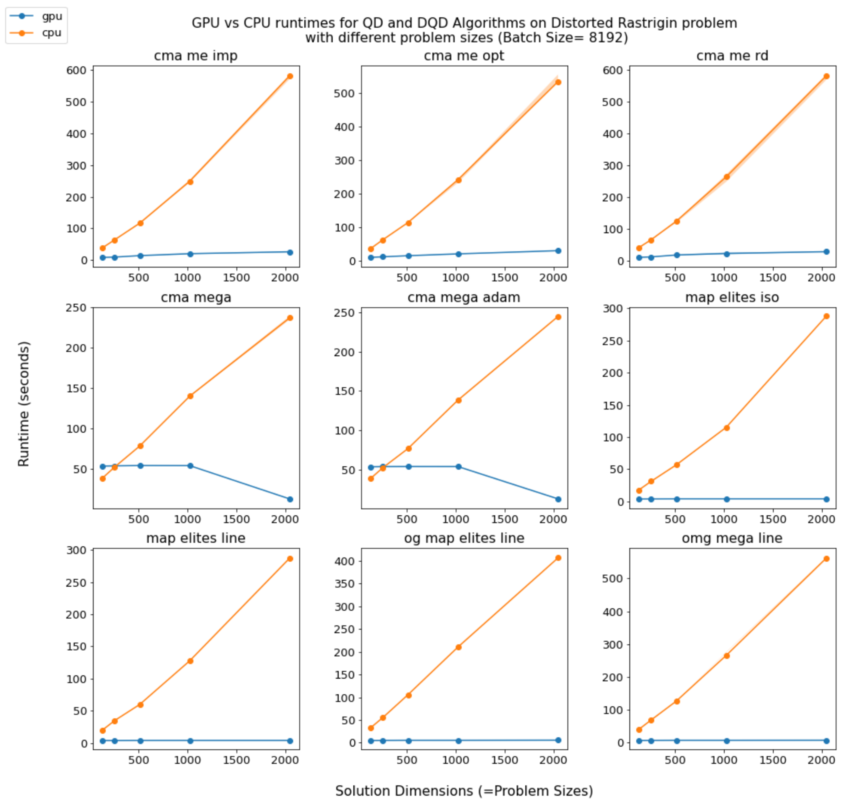


Fig C.43: Runtime of QD and DQD Algorithms on all the three problems for different problem sizes and constant batch size equal to 8192

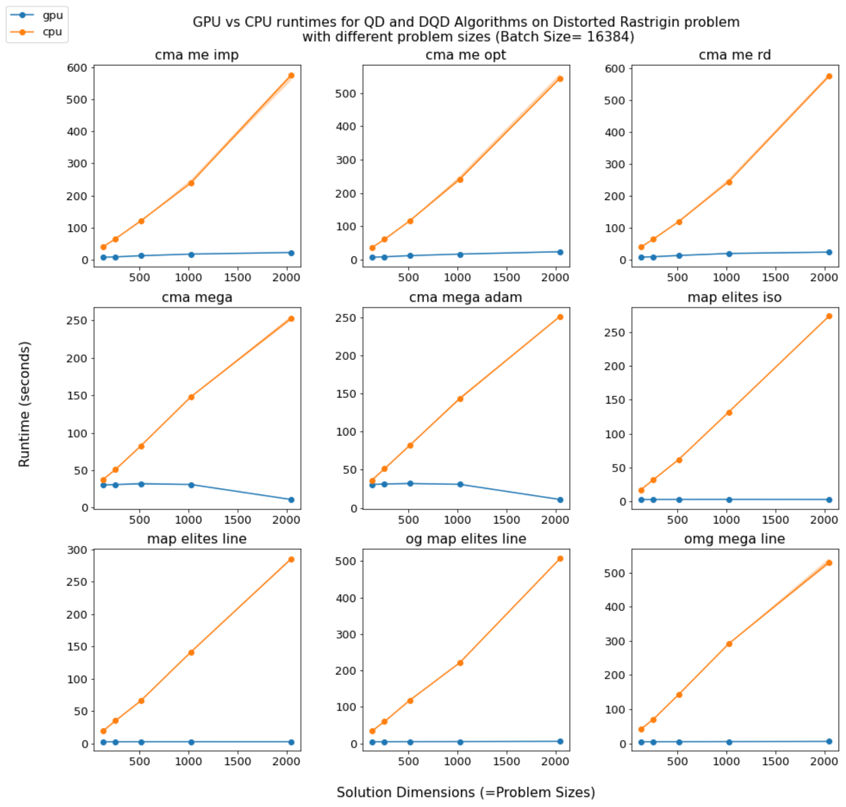


Fig C.44: Runtime of QD and DQD Algorithms on all the three problems for different problem sizes and constant batch size equal to 16384

C.4.3 Rastrigin with simple encoding

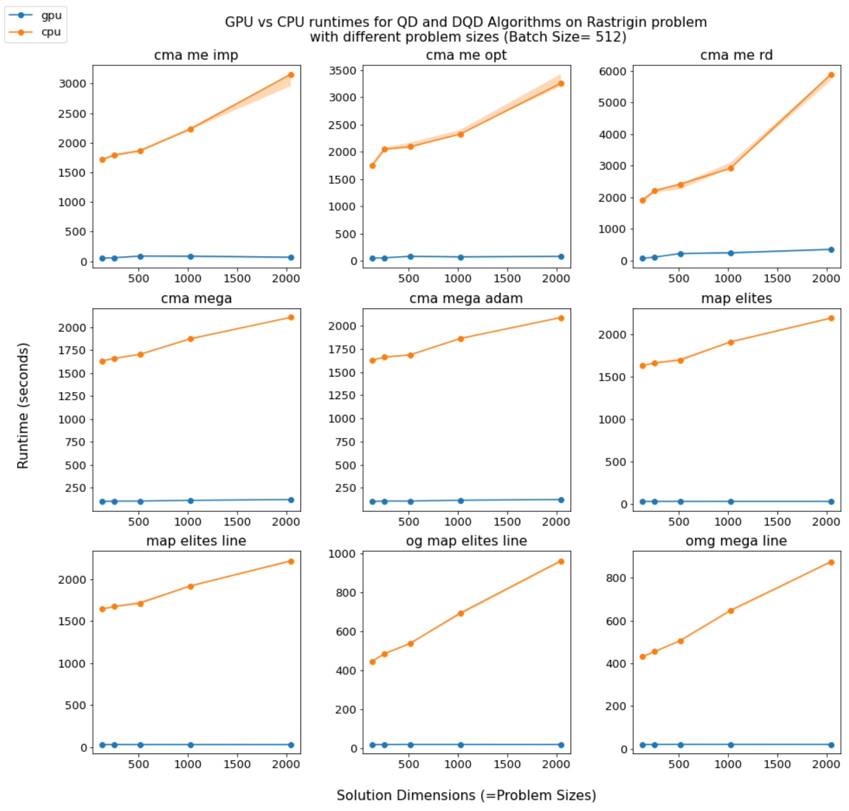


Fig C.45: Runtime of QD and DQD Algorithms on all the three problems for different problem sizes and constant batch size equal to 512

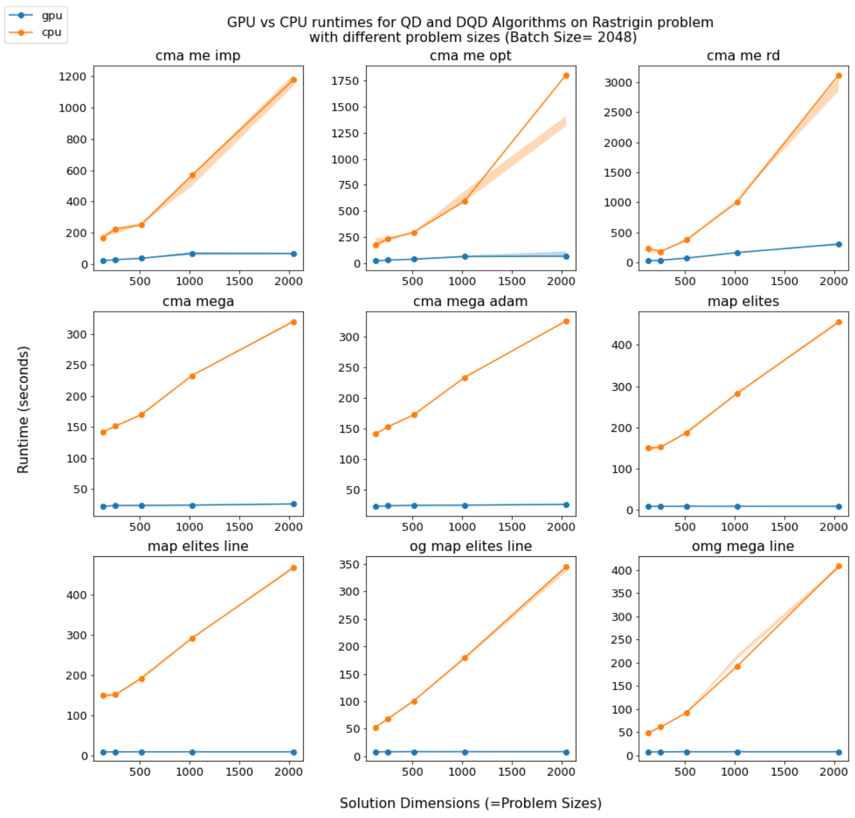


Fig C.46: Runtime of QD and DQD Algorithms on all the three problems for different problem sizes and constant batch size equal to 2048

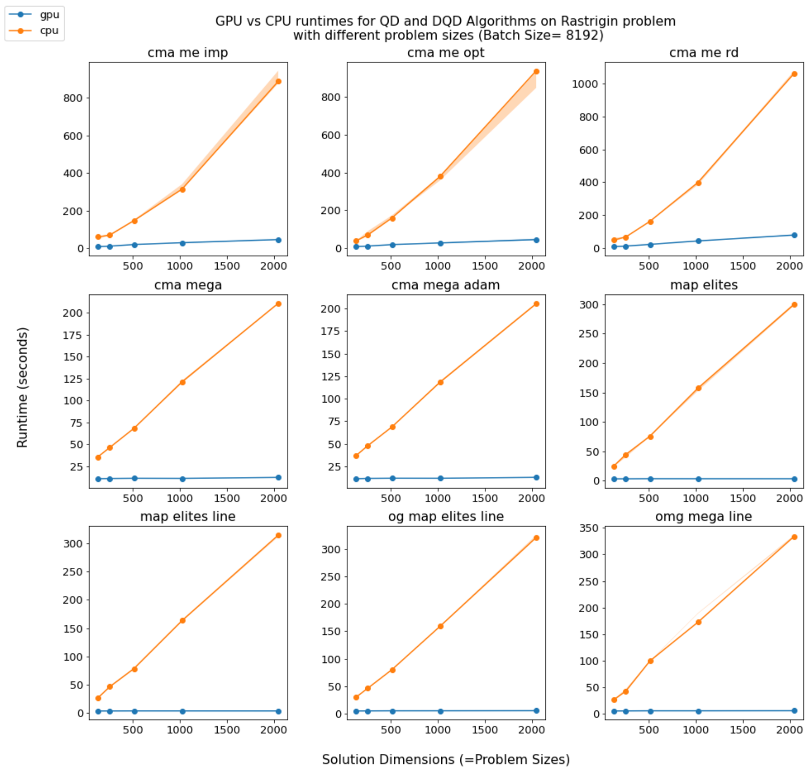


Fig C.47: Runtime of QD and DQD Algorithms on all the three problems for different problem sizes and constant batch size equal to 8192

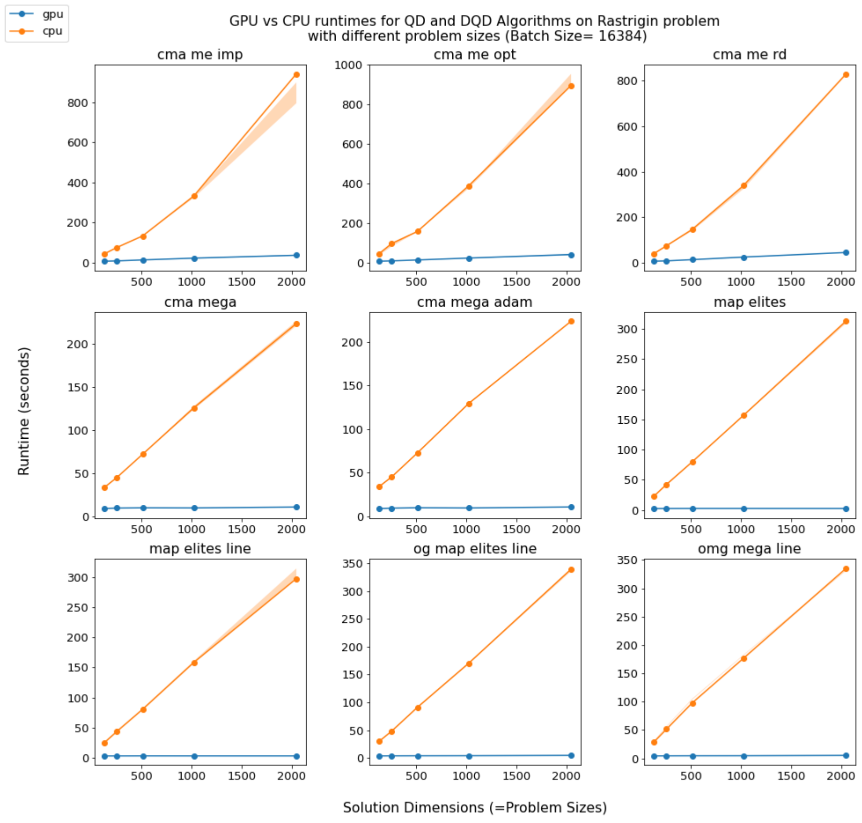


Fig C.48: Runtime of QD and DQD Algorithms on all the three problems for different problem sizes and constant batch size equal to 16384

C.5 Runtime - Batch Size Variation Charts

C.6 Arm Repertoire

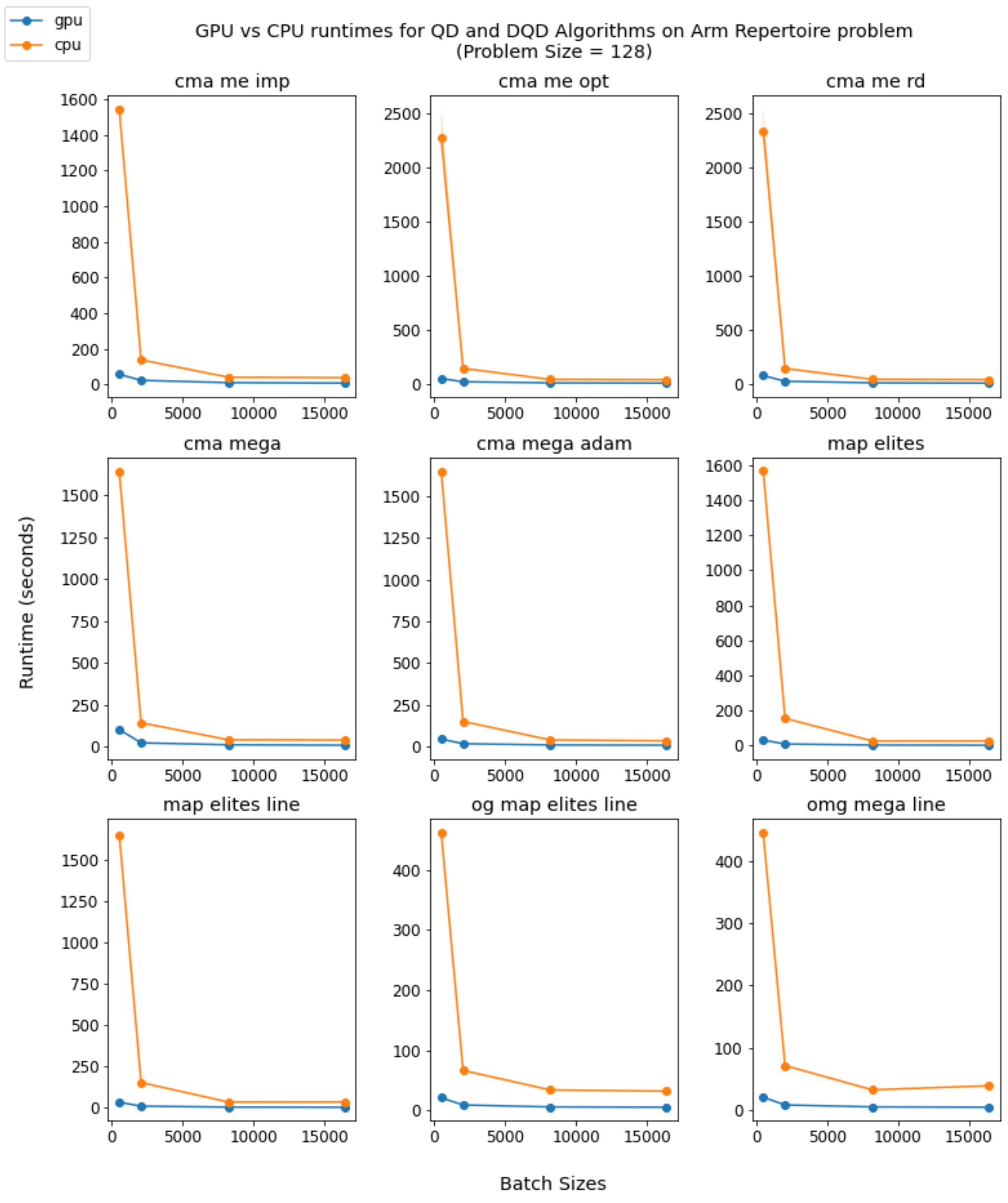


Fig C.49: Runtime of QD and DQD Algorithms on all the three problems for different batch sizes and constant problem size equal to 128

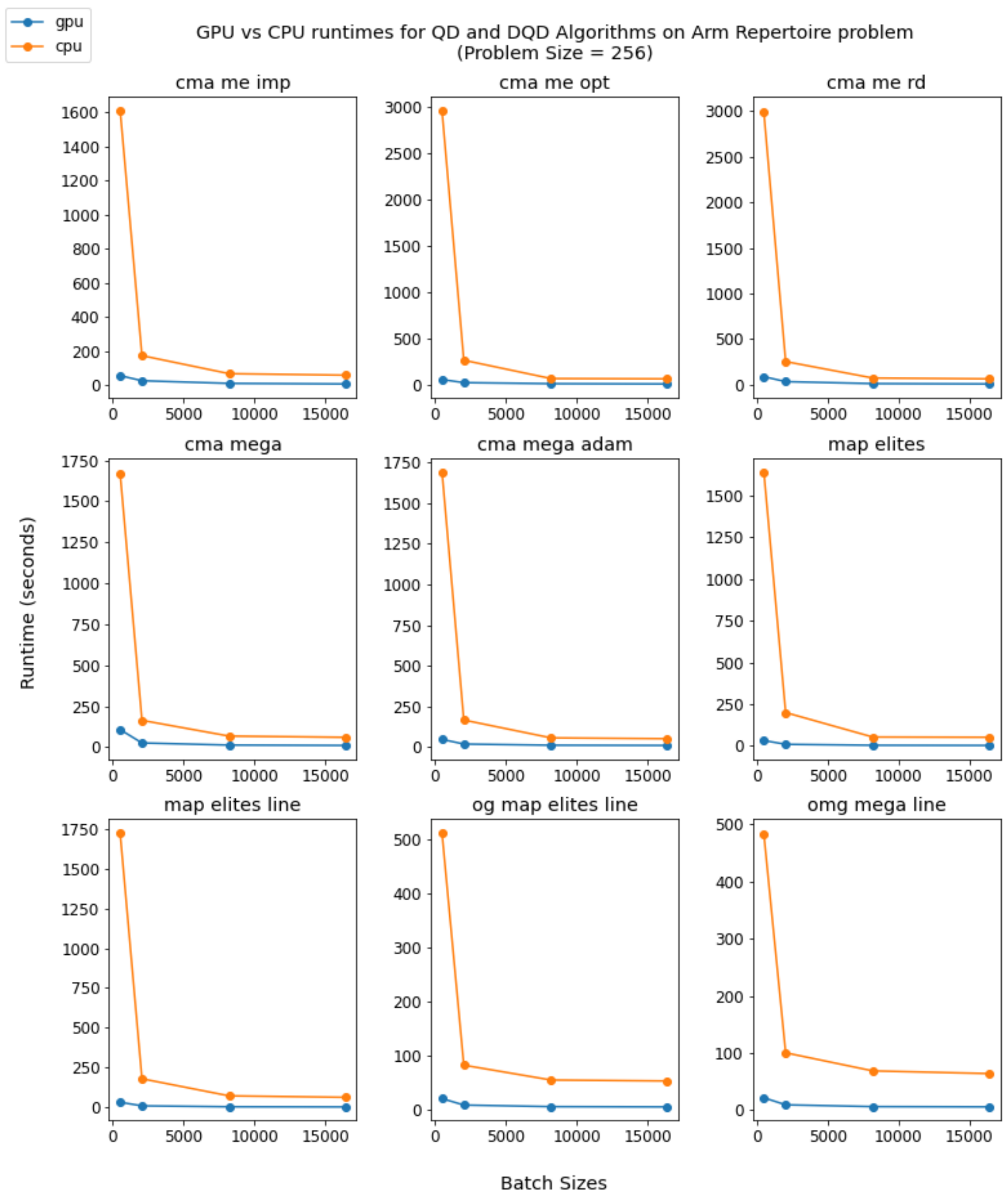


Fig C.50: Runtime of QD and DQD Algorithms on all the three problems for different batch sizes and constant problem size equal to 256

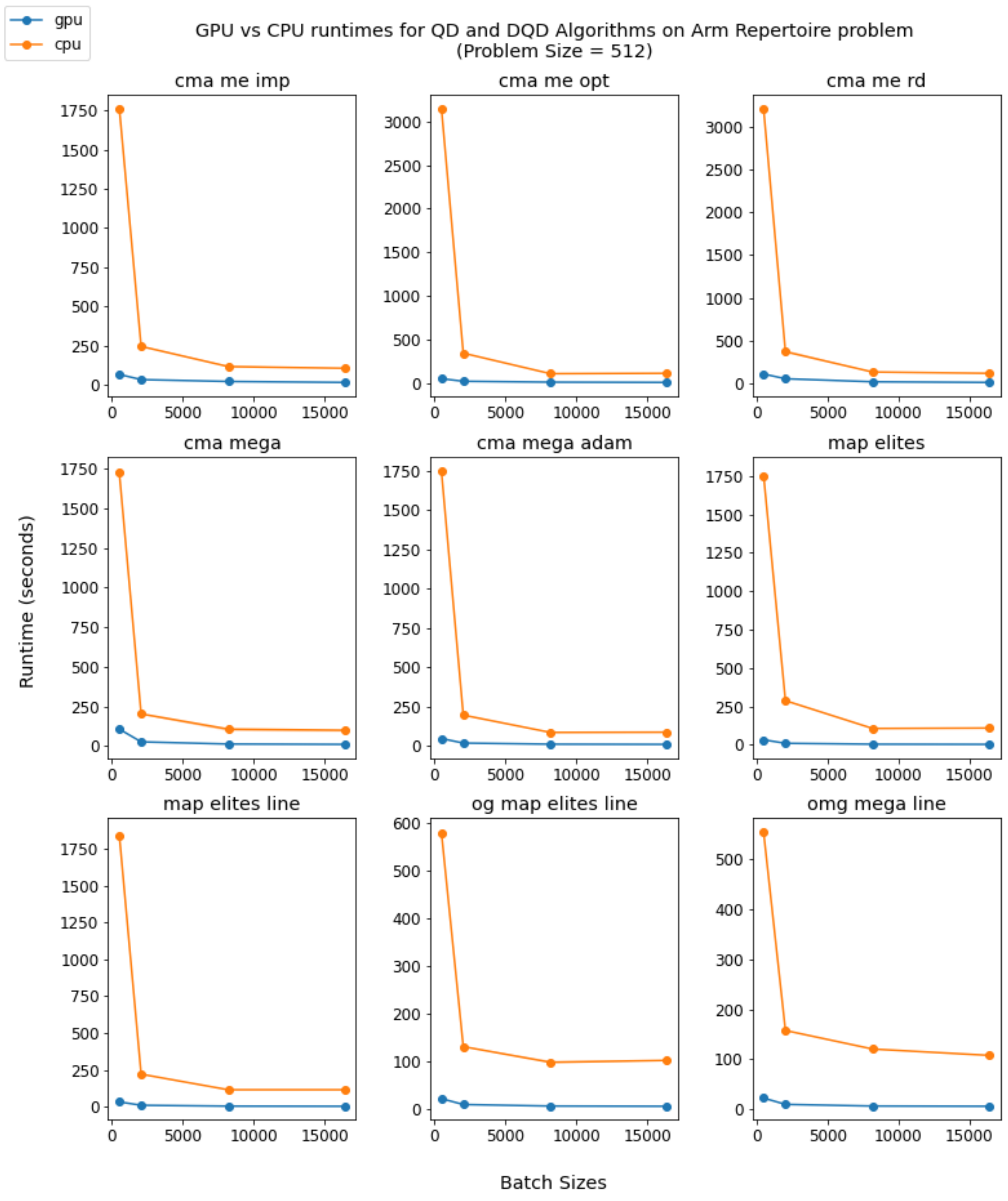


Fig C.51: Runtime of QD and DQD Algorithms on all the three problems for different batch sizes and constant problem size equal to 512

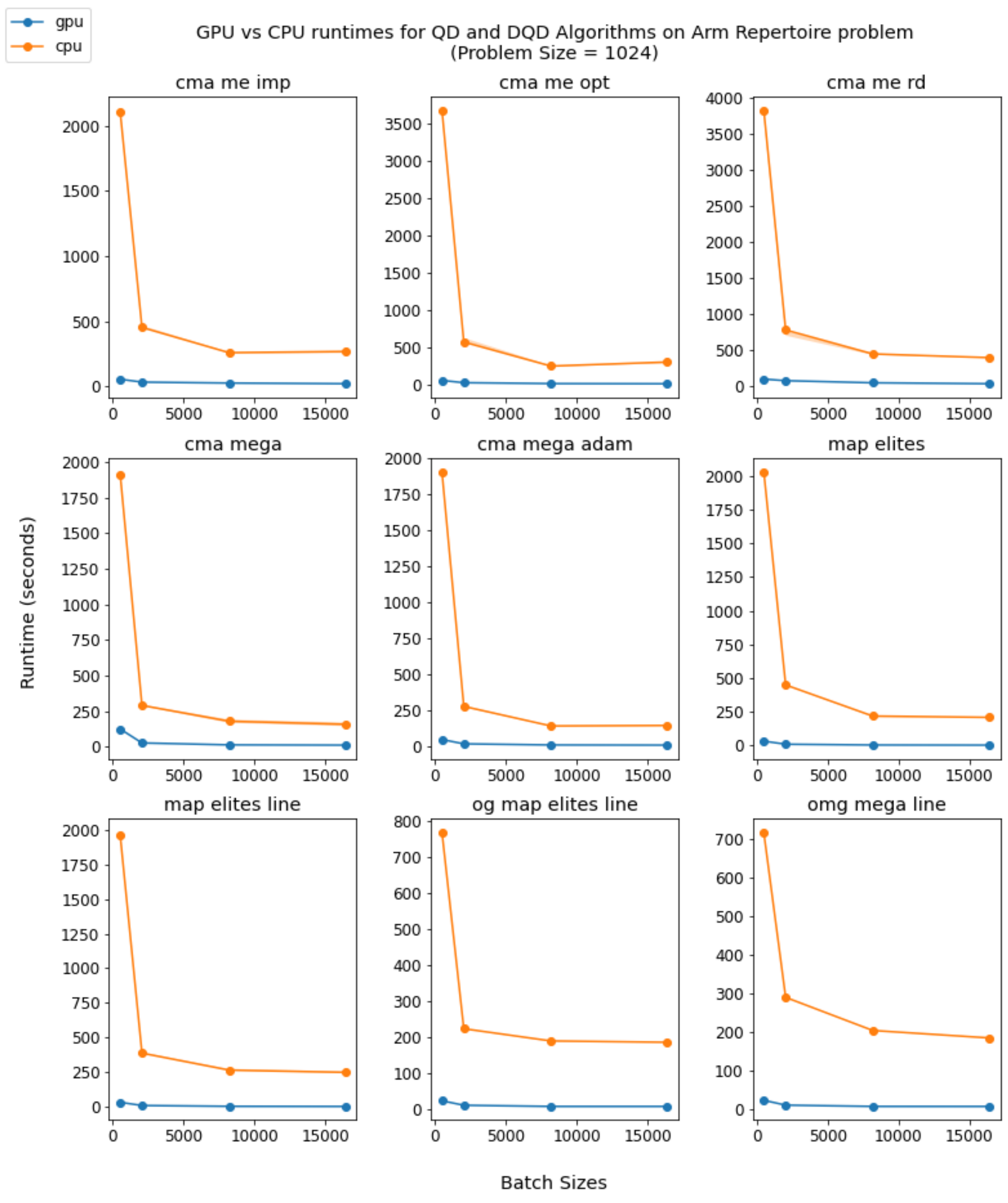


Fig C.52: Runtime of QD and DQD Algorithms on all the three problems for different batch sizes and constant problem size equal to 1024

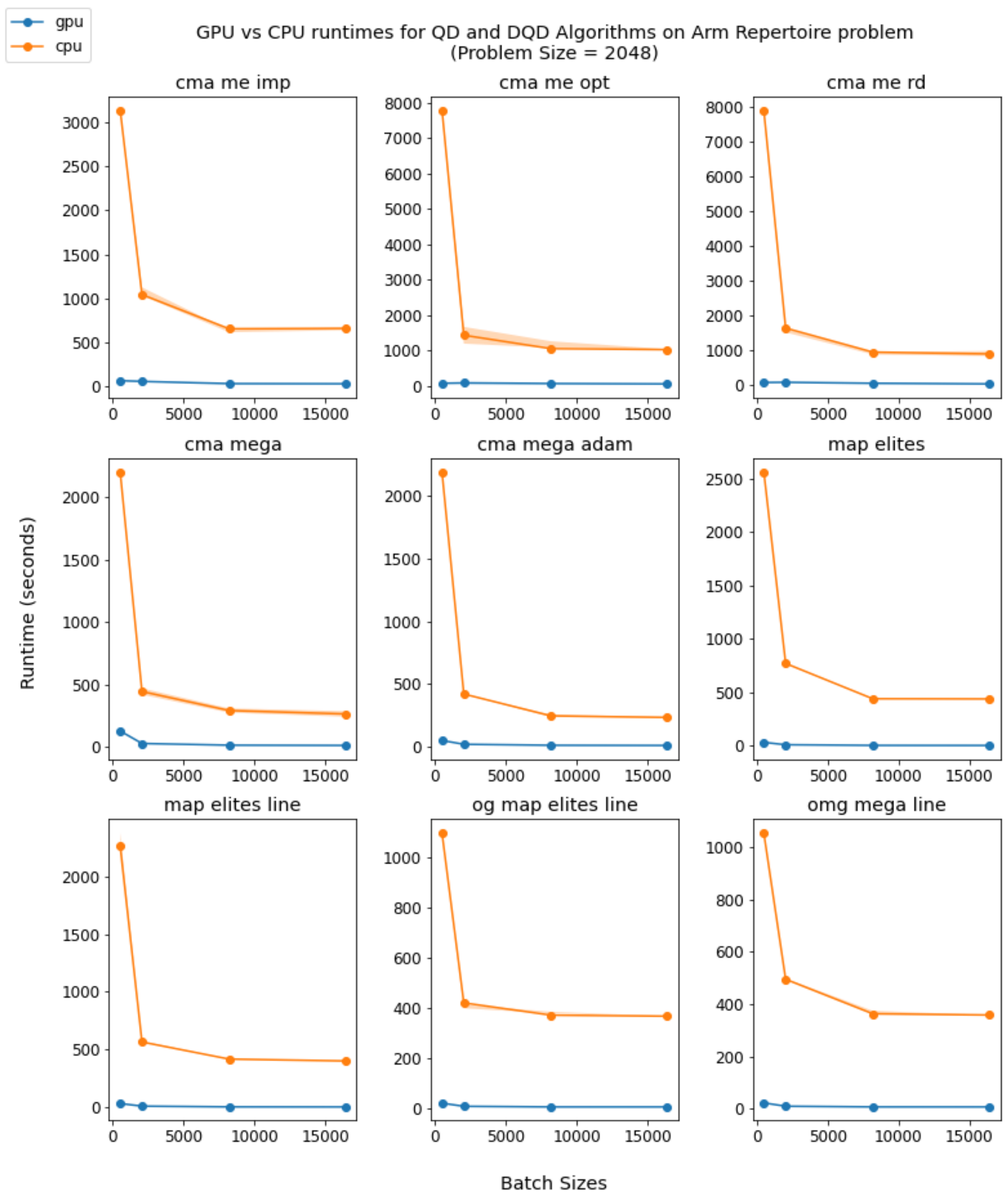


Fig C.53: Runtime of QD and DQD Algorithms on all the three problems for different batch sizes and constant problem size equal to 2048

C.6.1 Rastrigin with Distorted Behavior Space

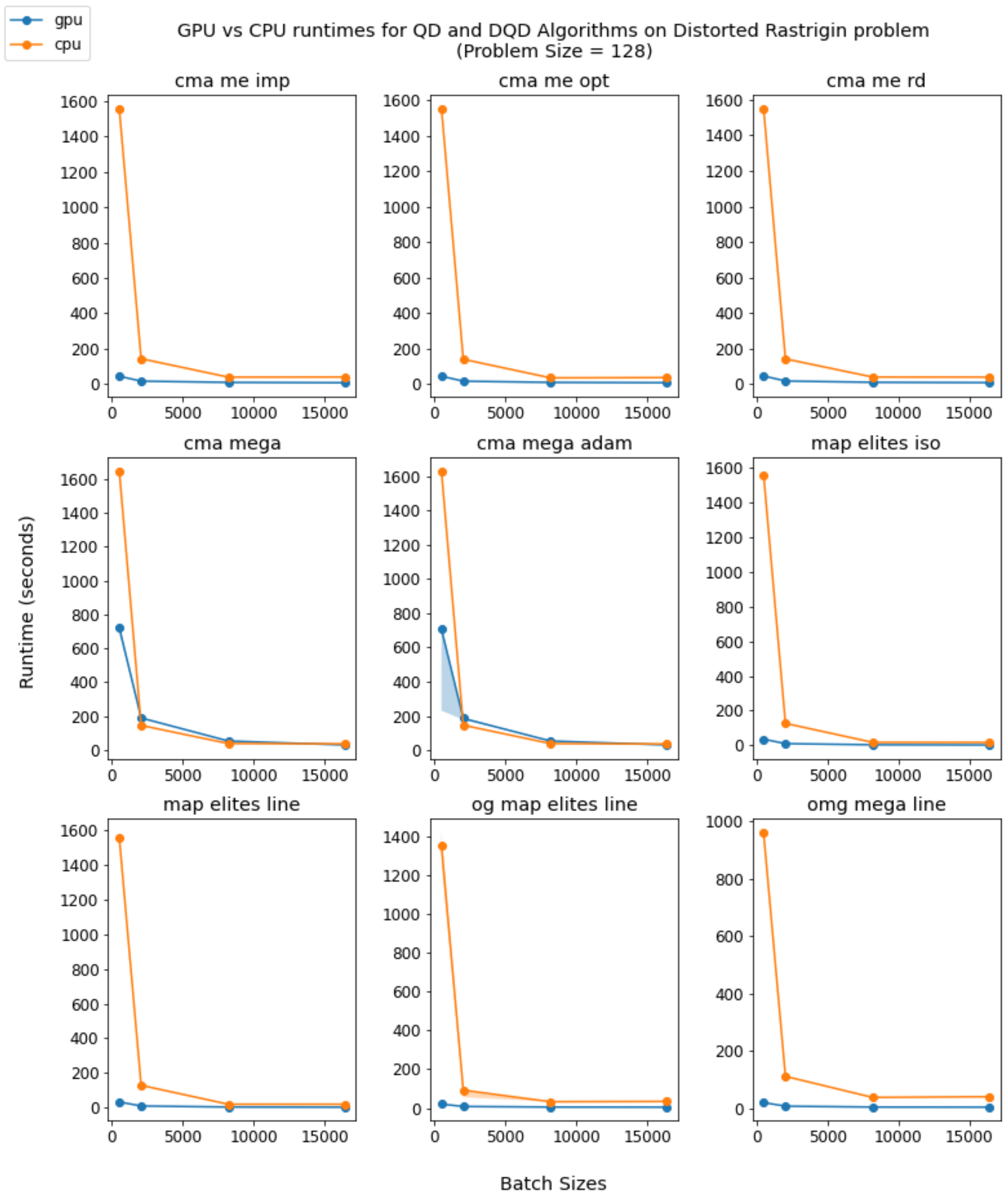


Fig C.54: Runtime of QD and DQD Algorithms on all the three problems for different batch sizes and constant problem size equal to 128

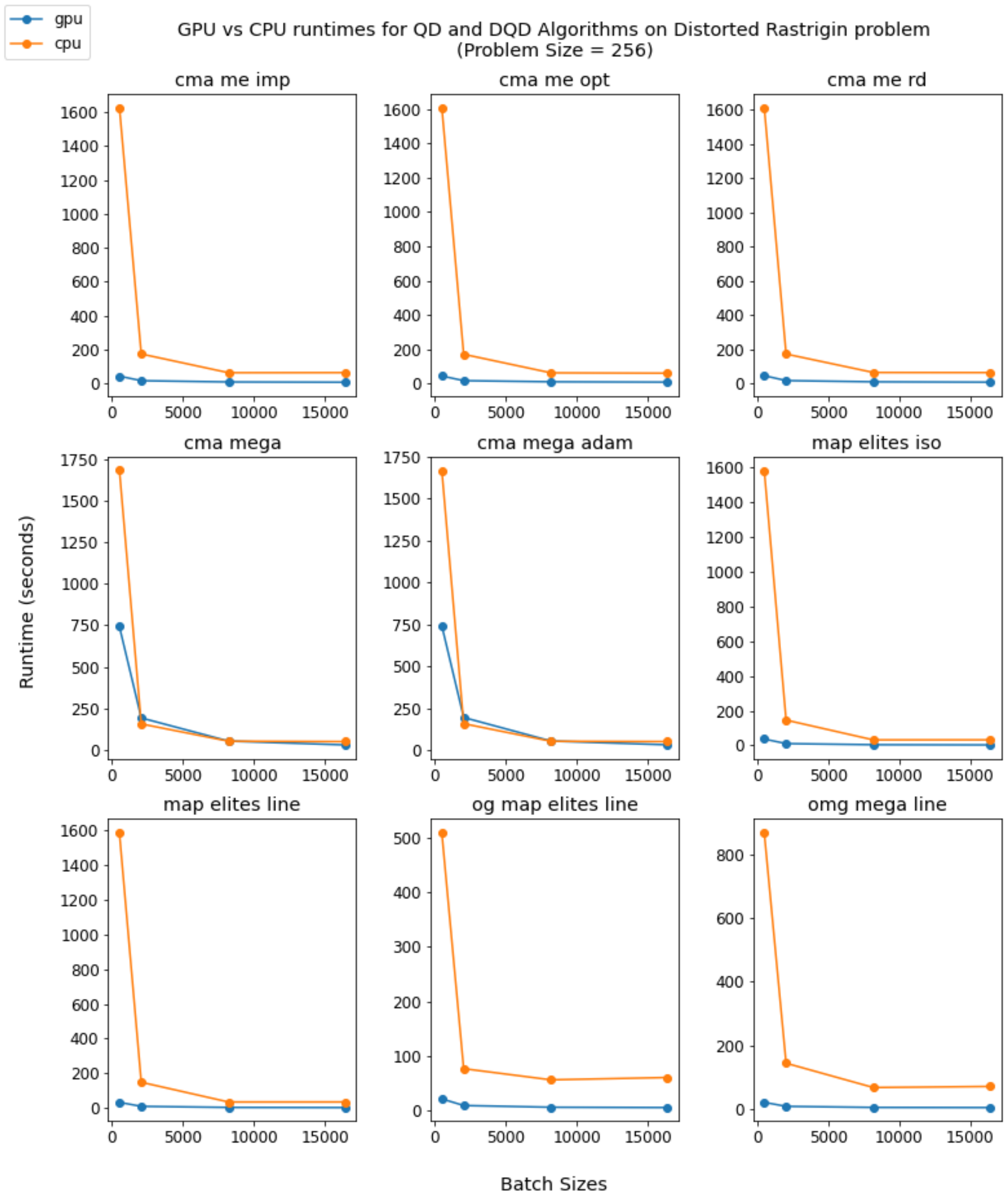


Fig C.55: Runtime of QD and DQD Algorithms on all the three problems for different batch sizes and constant problem size equal to 256

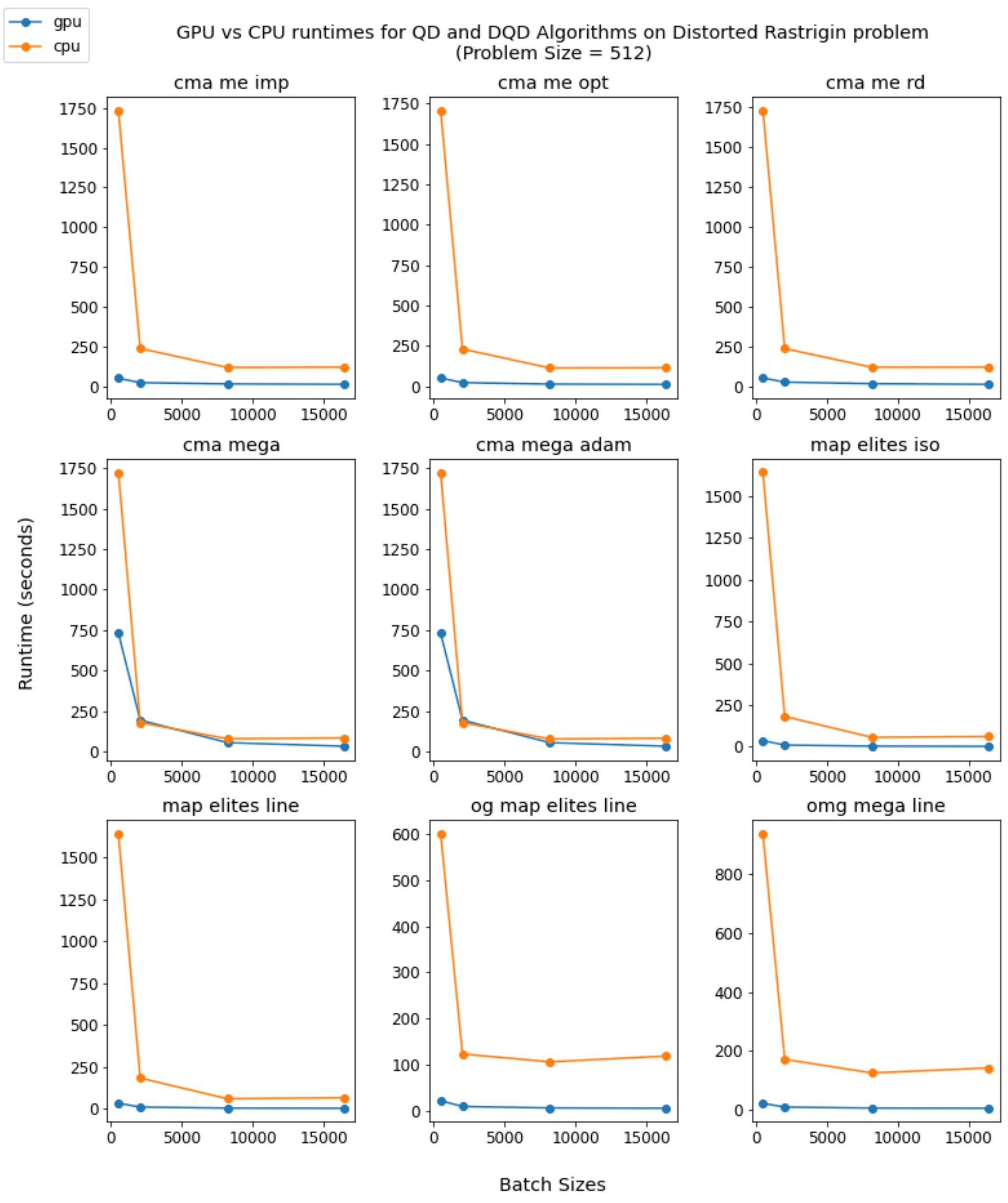


Fig C.56: Runtime of QD and DQD Algorithms on all the three problems for different batch sizes and constant problem size equal to 512

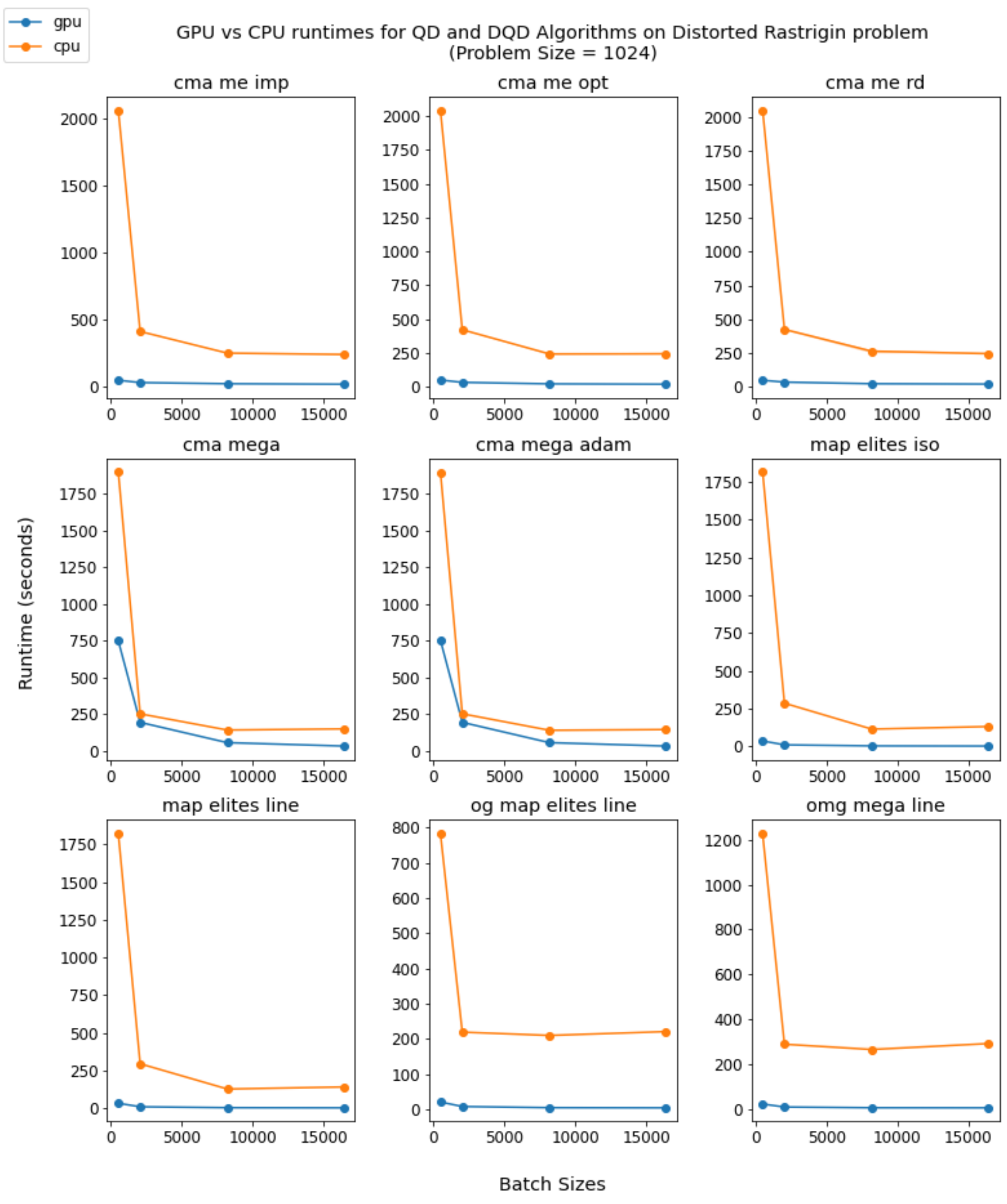


Fig C.57: Runtime of QD and DQD Algorithms on all the three problems for different batch sizes and constant problem size equal to 1024

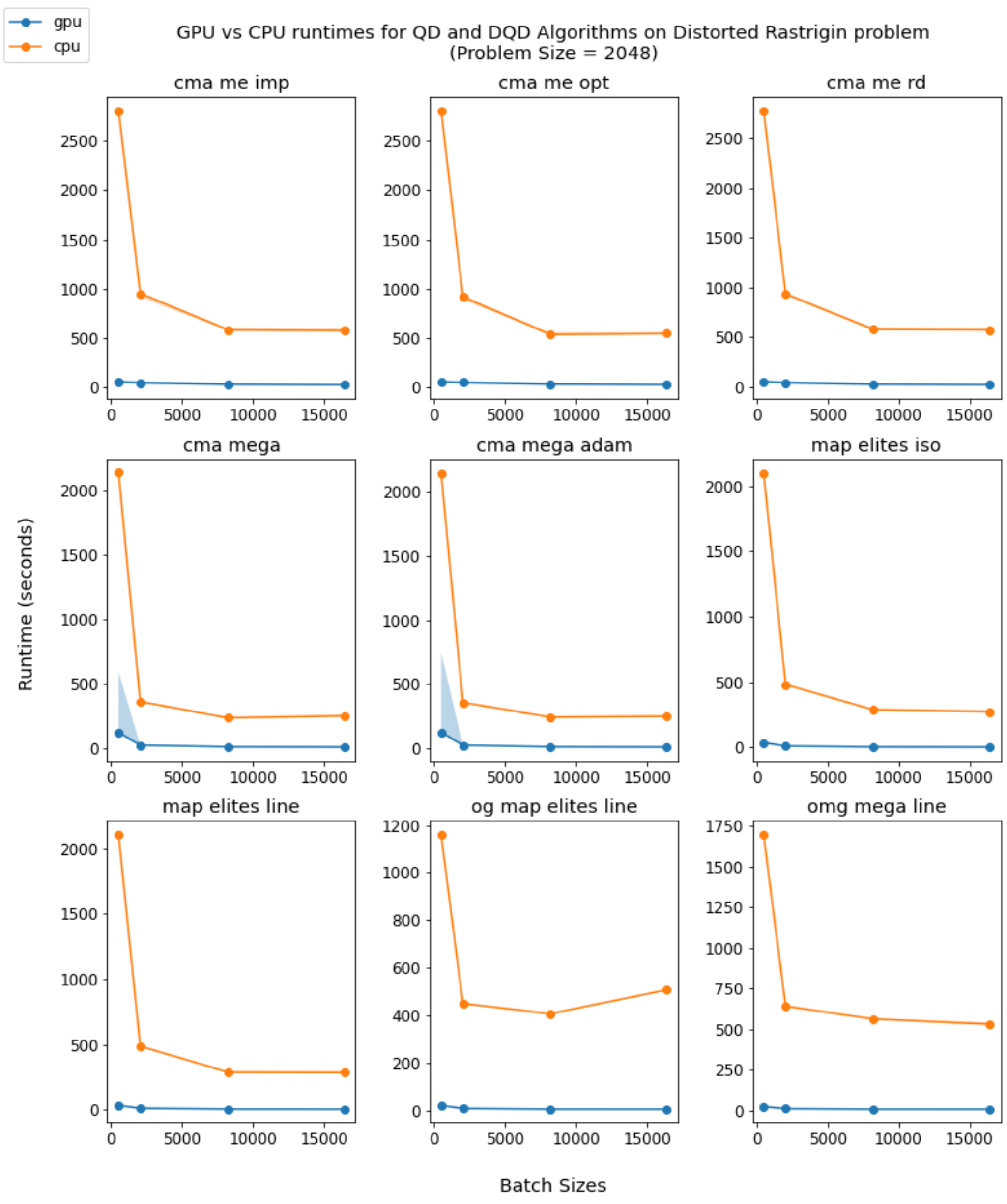


Fig C.58: Runtime of QD and DQD Algorithms on all the three problems for different batch sizes and constant problem size equal to 2048

C.6.2 Rastrigin with simple encoding

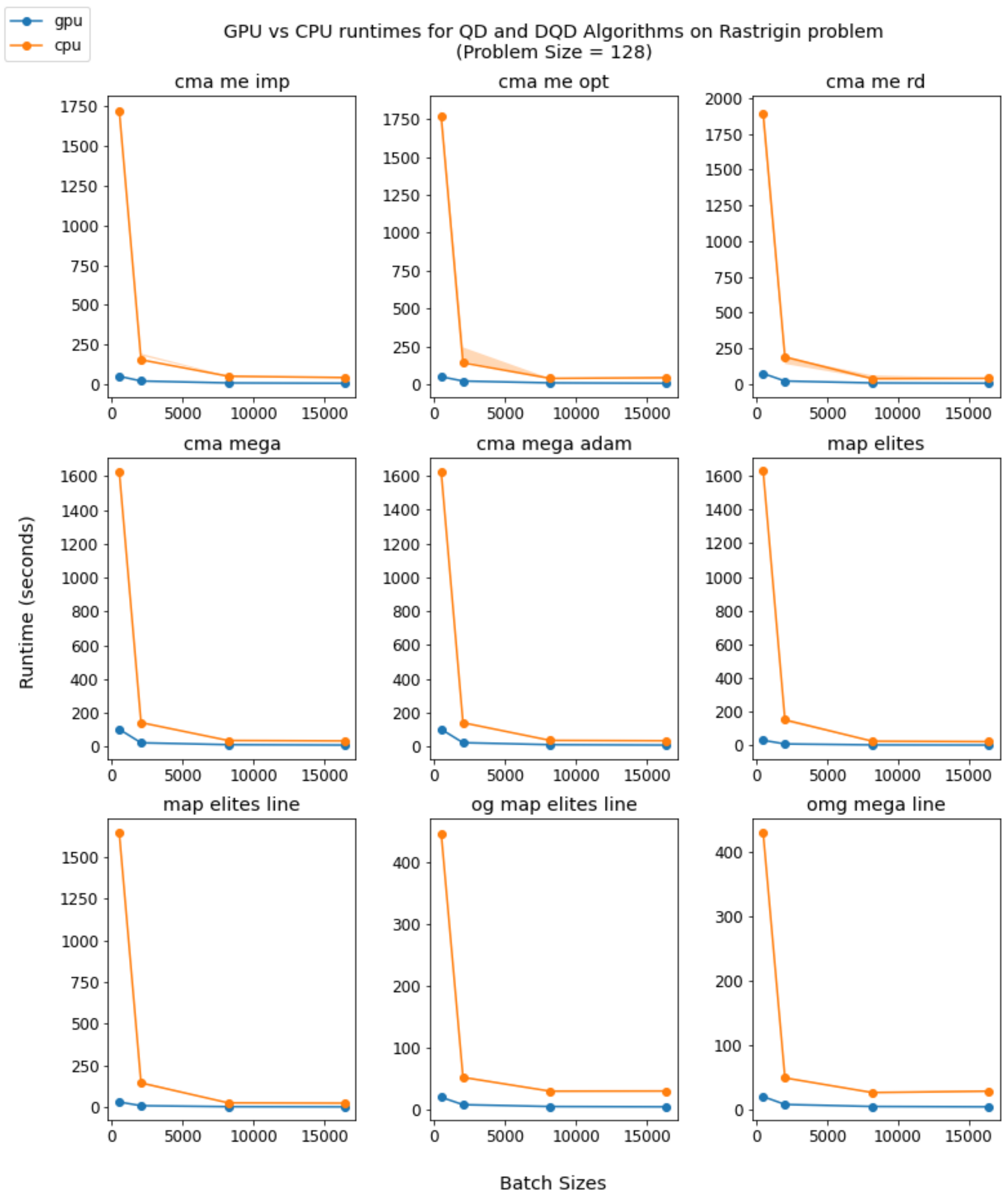


Fig C.59: Runtime of QD and DQD Algorithms on all the three problems for different batch sizes and constant problem size equal to 128

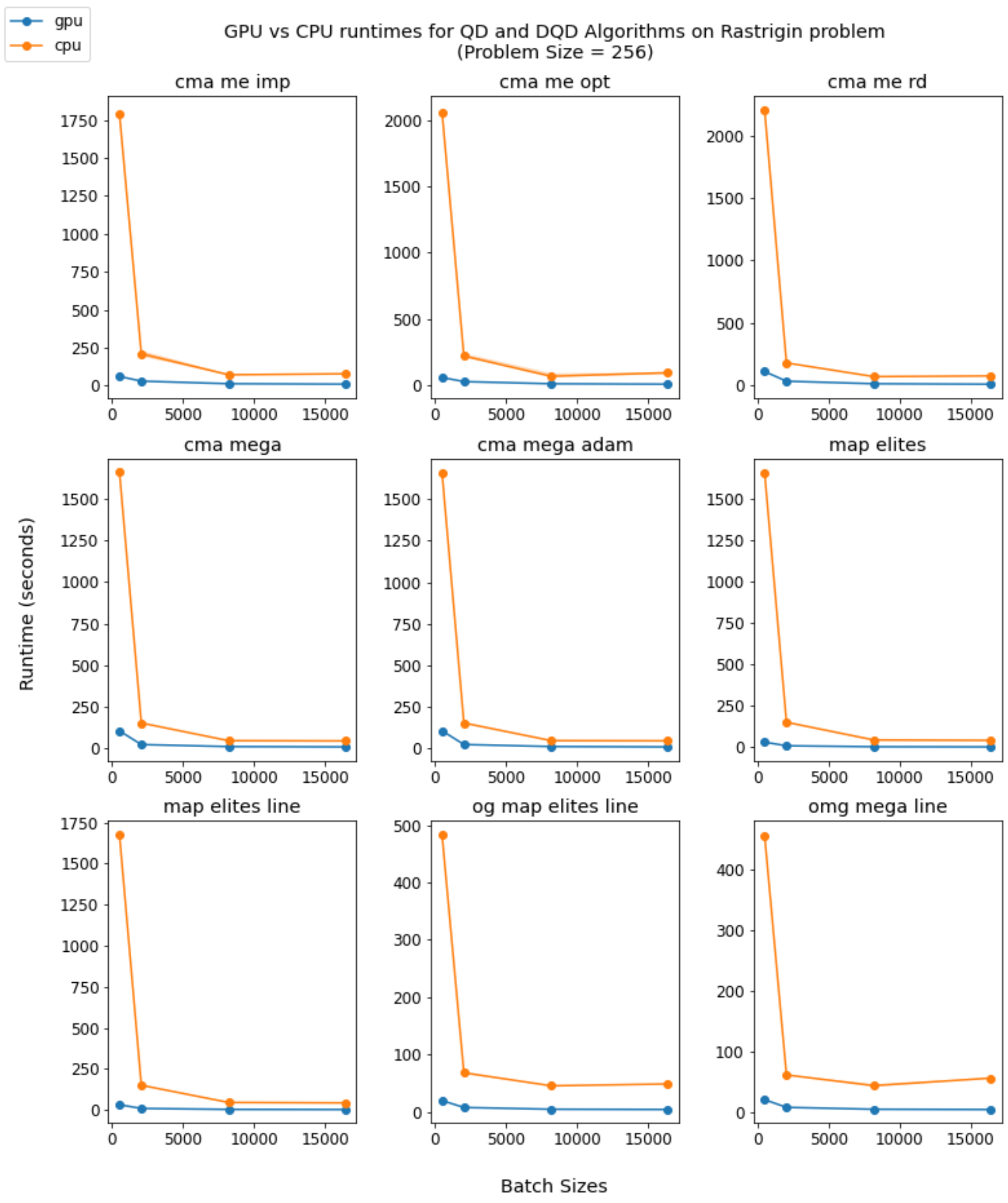


Fig C.60: Runtime of QD and DQD Algorithms on all the three problems for different batch sizes and constant problem size equal to 256

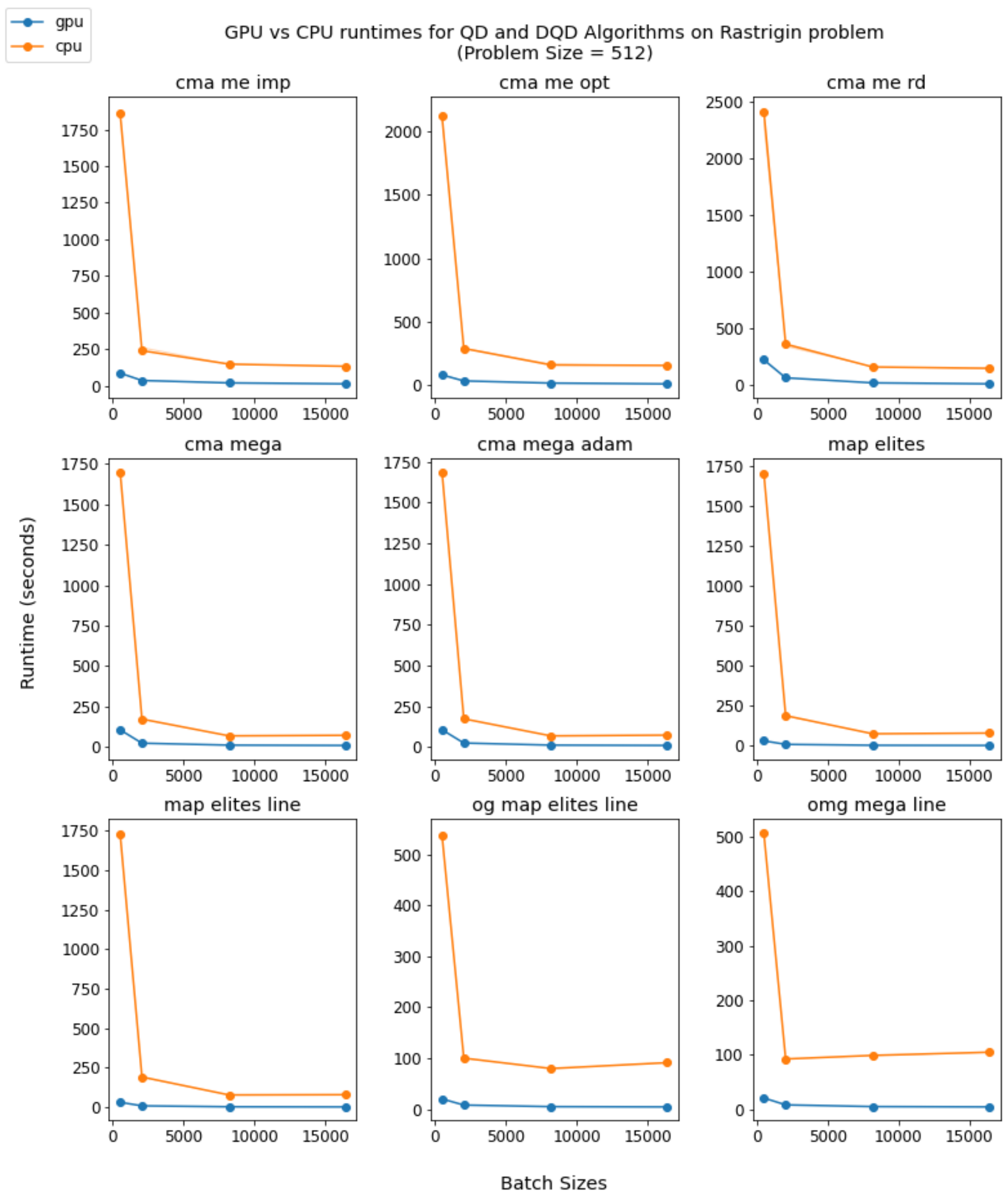


Fig C.61: Runtime of QD and DQD Algorithms on all the three problems for different batch sizes and constant problem size equal to 512

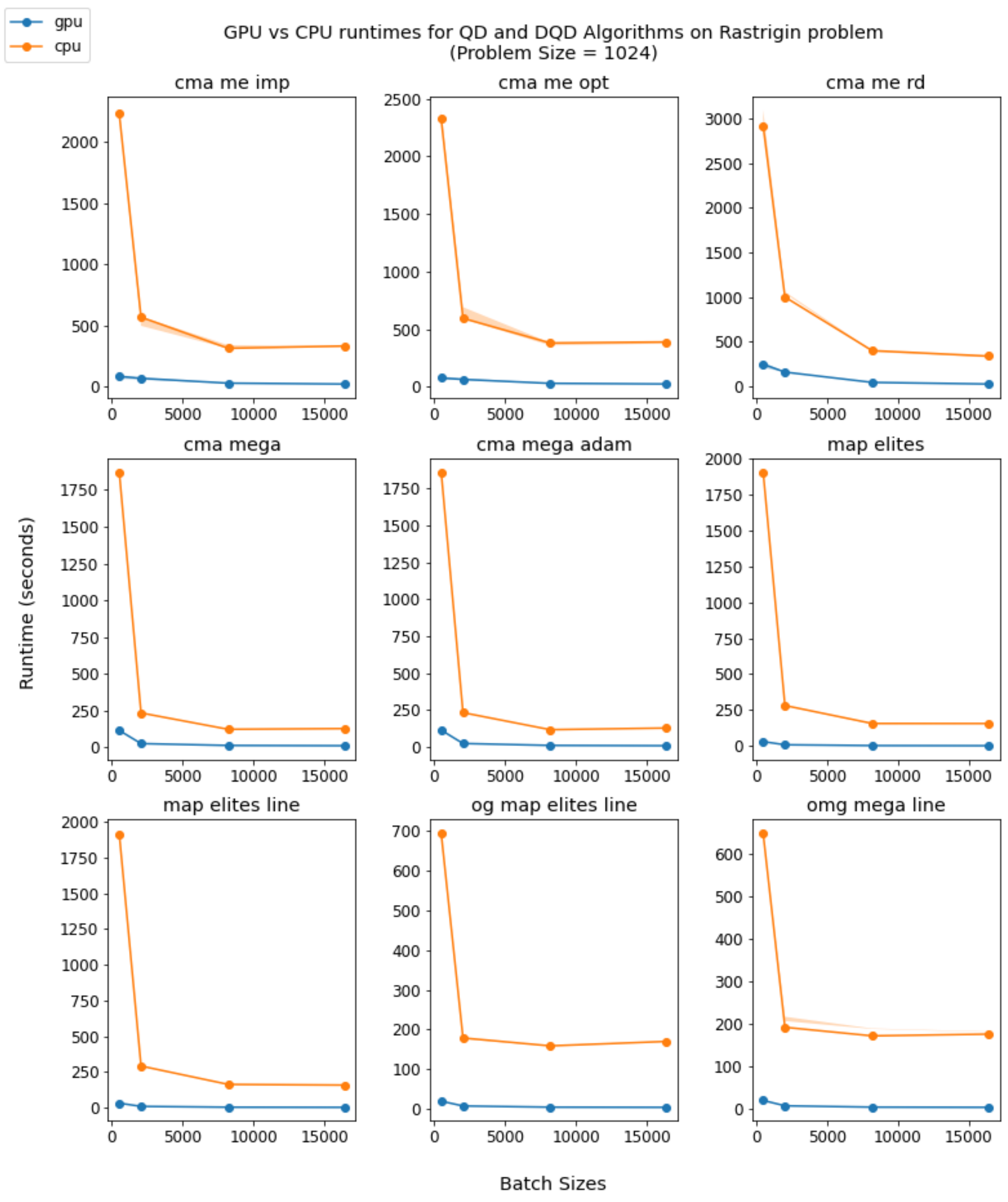


Fig C.62: Runtime of QD and DQD Algorithms on all the three problems for different batch sizes and constant problem size equal to 1024

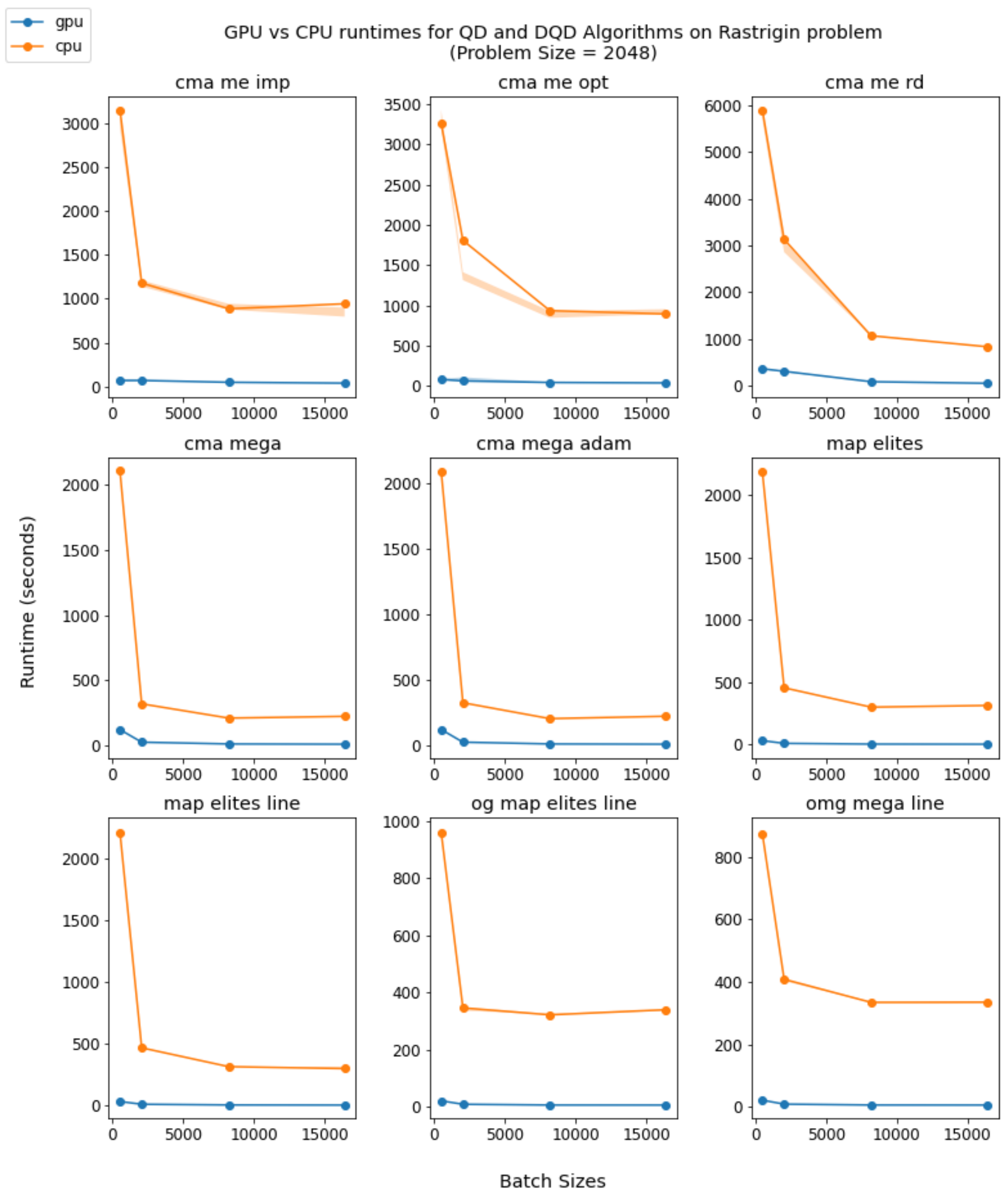


Fig C.63: Runtime of QD and DQD Algorithms on all the three problems for different batch sizes and constant problem size equal to 2048

The End.