Thesis Dissertation

Accelerating Quality Diversity Algorithms using K-Means Clustering

Andreas Pattichis

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

May 2023

UNIVERSITY OF CYPRUS
COMPUTER SCIENCE DEPARTMENT

Accelerating Quality Diversity Algorithms using K-Means Clustering

Andreas Pattichis

Supervisors
Dr. 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 University of Cyprus

May 2023

Acknowledgements

I would like to thank my supervisors, Dr. Chris Christodoulou and Dr. Vassilis Vassiliades, for their guidance and assistance throughout my Bachelor Thesis project. Their help was invaluable, and I am very appreciative of the time and effort they invested in assisting me to complete it.

Abstract

Quality Diversity (QD) algorithms, with their aim of finding diverse, high-quality solutions across solution spaces, present a unique approach to optimization, distinctly different from other traditional techniques. Despite their advantages, these algorithms struggle to navigate non-convex elite hypervolumes characterized by complex and irregular geometries. This research proposes an adaptive strategy to overcome this challenge, focusing on the context of MAP-Elites, a well-known QD algorithm. Our solution centers on integrating K-Means clustering into the emitter selection process, crucial for generating diverse, high-performing solutions. We evaluated our strategies through a series of experimental scenarios, focusing on both exploration and exploitation capabilities. Comparative analysis with the established Iso+LineDD emitter demonstrated varying performance across different scenarios and performance metrics. While our techniques outperformed the Iso+LineDD emitter in some cases, they did not always produce improved performance, indicating the need for further enhancement and exploration of alternative clustering strategies.

Contents

CHAPTER 1 INTRODUCTION **1**

1.1. Evolution of Quality Diversity Algorithms 1

1.2. Motivation 2

1.3. Previous Quality Diversity Research related to Our Work 2

1.4. Objectives 3

1.5. Potential Impact 3

CHAPTER 2 BACKGROUND **4**

2.1. Mathematical Foundations 4

2.1.1. Function 4
2.1.2. Discovering the Field of Mathematical Optimization 4
2.1.3. Rastrigin Function 4
2.1.4. Distance Metrics 6
2.1.5. Mathematical field of Probability theory 6
2.1.6. Stochastic Variables 7
2.1.7. Distributions 7

2.2. Evolutionary Algorithms 9

2.2.1. Overview of Evolutionary Algorithms 9
2.2.2. Application of Genetic Algorithms 9

2.3. Clustering 11

2.3.1. Introduction to Cluster Analysis 11
2.3.2. K-Means Clustering algorithm 11

2.4. Quality-Diversity Background 15

2.4.1. Introduction to Quality Diversity algorithms 15
2.4.2. QD Problem Formulation 15
2.4.3. Illumination 16
2.4.4. Initial QD Algorithms 17
2.4.5. Introduction to QD Optimization framework 18
2.4.6. Key Components of the QD Optimization Framework 19
2.4.7. Evaluation Metrics for Assessing QD Algorithms' Success 21
2.4.8. Elite Hypervolume 21
2.4.9. Multi-dimensional Archive of Phenotypic Elites (MAP-Elites) 23
2.4.10. Emitters 25

CHAPTER 3 METHODOLOGY **28**

3.1. Problem Formulation 28

3.2. Proposed Clustering-Based Emitter 29

3.3. JAX and QDJAX Library 31

3.4. K-Means for JAX Implementation 31

3.5. Implementation 34

3.5.1. Implementation of Iso+LineDD Emitter on QDJAX 34
3.5.2. Adapting Iso+LineDD Emitter through Reduced Batch Sampling 35
3.5.3. Enhancing Emitter Selection Process through K-Means Clustering Integration 37

3.6. Experimental Setup 44

3.6.1. Problem domains 44
3.6.2. Experimental Scenarios 45
3.6.3. Key Metrics for Evaluating Performance 47

CHAPTER 4 EXPERIMENTAL RESULTS AND DISCUSSION 49

4.1. INTRODUCTION 49

4.2. RESULTS FOR 2-DIMENSIONAL PROBLEM DOMAINS 49

4.2.1. Scenario 1 with 100x100 Grid Size for 2-D Robotic Arm Problem 50
4.2.2. Scenario 2 with 100x100 Grid Size for 2-D Rastrigin Function 57
4.2.3. Scenario 3 with 400x400 Grid Size for 2-D Robotic Arm Problem 64
4.2.4. Scenario 4 with 400x400 Grid Size for 2-D Rastrigin Function 71

4.3. RESULTS FOR HIGHER-DIMENSIONAL PROBLEM DOMAINS 78

4.3.1. Scenarios 5 - 6 with 100x100 Grid Size for 10-D Problem Domains 78
4.3.2. Scenario 7 - 8 with 100x100 Grid Size for 100-D Problem Domains 92

4.4. MAIN FINDINGS 105

CHAPTER 5 CONCLUSIONS AND FUTURE WORK 107

5.1. CONCLUSIONS 107

5.2. FUTURE WORK 108

REFERENCES 109

APPPENDIX A IMPLEMENTATIONS A-1

A.1 IMPLEMENTATION OF K-MEANS CLUSTERING ON JAX A-1

A.2 IMPLEMENTATION OF ISO+LINEDD EMITTER ON QDJAX A-5

A.3 IMPLEMENTATION OF ISO+LINEDD EMITTER WITH REDUCED BATCH SAMPLING A-8

A.4 IMPLEMENTATION OF STRATEGY 1 ..... A-12

A.5 IMPLEMENTATION OF STRATEGY 2 ..... A-18

Chapter 1
Introduction

1.1. Evolution of Quality Diversity Algorithms 1
1.2. Motivation 2
1.3. Previous Quality Diversity Research related to Our Work 2
1.4. Objectives 3
1.5. Potential Impact 3

1.1. Evolution of Quality Diversity Algorithms

Unlike standard optimization algorithms that aim to find one best solution, Quality Diversity (QD) algorithms strive for a diverse group of high-performing alternatives [1]. As time has passed, these QD algorithms have undergone several improvements, with new strategies and techniques being introduced to enhance their overall efficiency and effectiveness.

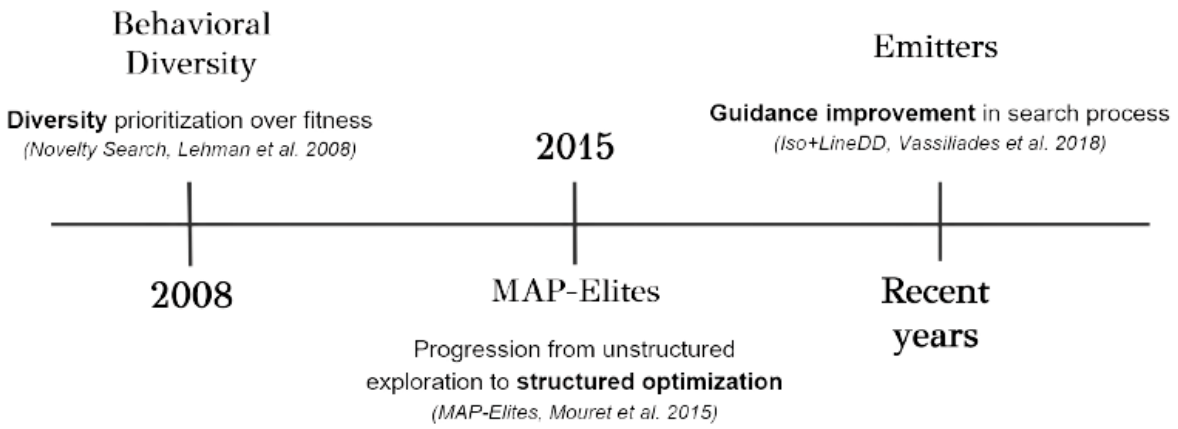
Lehman and Stanley introduced the Novelty Search algorithm in 2008 [1], which is one of the earliest examples of QD algorithms. Instead of focusing on the fitness of solutions, the Novelty Search algorithm promoted the exploration of the search space by prioritizing the novelty of the solutions for the first time. Other QD algorithms have been developed since then, such as Multi-dimensional Archive of Phenotypic Elites (MAP-Elites) proposed by Mouret and Clune in 2015 [2].

In order for QD algorithms to fulfill their objective of exploring the search space, they maintain and optimize a population of solutions with diverse behaviors and traits [2]. By doing so, QD algorithms are able to discover a set of solutions that cover a wide range of behaviors, instead of solely focusing on a single optimal one [3].

Particularly, QD algorithms investigate regions referred to as non-convex elite hypervolumes, which are regions of the solution space that contain high-performing solutions but have complex and irregular geometries [4]. Their non-convex nature presents challenges, as these areas may include multiple peaks and valleys that make searching for solutions more difficult [5].

QD algorithms have proven to be an essential addition to optimization algorithms due to their ability to identify solutions that other regular optimization algorithms may overlook. In multi-objective optimization situations for example, when multiple conflicting objectives must be optimized concurrently, QD algorithms can identify solutions that cover a large range of trade-offs between the objectives [2].

Recent years have seen the introduction of emitters, such as the Iso+LineDD emitter. Emitters are components that guide the search process more efficiently, helping to explore new regions of the solution space in many QD algorithms [4].



1.2. Motivation

Quality Diversity algorithms, especially in complex environments, often encounter difficulty when attempting to navigate non-convex elite hypervolumes [5]. These complex regions, although rich in high-performing solutions, present complex geometries that make navigation difficult [5]. Despite the success of QD algorithms like MAP-Elites in discovering diverse solutions in high-dimensional search spaces, their reliance on user-defined boundaries for the behavior space limits their effectiveness in adapting to the shape of these non-convex regions [6]. This limitation restricts the discovery and optimization of diverse, high-performing solutions in these crucial areas of the search space.

Tackling this challenge has significant implications. If QD algorithms can adapt more efficiently to the shape of the behavior space, they would be better equipped to identify superior solutions in complex and high-dimension spaces [6]. This adaptability would make them even more powerful tools for a wide range of optimization tasks. As a result, the motivation of our work is to develop an adaptive strategy capable of addressing these difficulties associated with exploring and optimizing non-convex elite hypervolumes.

1.3. Previous Quality Diversity Research related to Our Work

One approach of addressing this issue stated in section 1.2. was introduced with the Cluster-Elites algorithm [7]. This algorithm utilizes a clustering technique based on density to maximally spread a set of centroids on a potentially non-convex manifold in the behavior space. The behavior space is partitioned into clusters of similar solutions using this density-based clustering

method, and the centroids of these clusters serve as the representative solutions in the MAP-Elites algorithm. The centroids are updated at each generation. In a maze navigation problem, the Cluster-Elites algorithm was found to perform comparably or better than other algorithms such as MAP-Elites. This successful use of clustering methods implies that they have the potential to improve the performance of numerous QD algorithms. That said, it is necessary to conduct further research to explore the possibility of integrating alterative clustering techniques into existing QD algorithms for an overall enhancement in performance.

1.4. Objectives

The primary goal of this research is to improve the exploration of high-performing behaviors in the behavior space, to better comprehend potential system behaviors, and to optimize non-convex elite hypervolumes using Quality Diversity algorithms.

To tackle these challenges, we selected a variety of distinct objectives in my thesis proposal in order to investigate the new strategies in exploring and optimizing these non-convex regions. First and foremost, the research aims to investigate two new methods for enhancing the emitter selection process in MAP-Elites through the integration of clustering techniques such as K-Means. The novel strategies' applicability and performance will be assessed in various experimental scenarios focusing on both their exploration and exploitation capabilities. The performance of the proposed strategies will also be assessed by comparing them to the Iso+LineDD emitter, on which the new strategies are based on. This comparison will help determine any improvements in key metrics achieved by the proposed strategies. Finally, the research aims to offer insights into the scalability of the proposed strategies, as well as their capacity to manage complex, higher-dimensional problem domains.

1.5. Potential Impact

By addressing these objectives, these strategies could lead to a more effective exploration of varied behaviors in the behavior space as well as provide significant insights into the potential behaviors of a system. By integrating clustering techniques into the emitter's selection process, the proposed methods can potentially also lead to the discovery of high-performing solutions that may not have been found using other methods. The ultimate objective is to develop a strategy capable of identifying superior solutions in complex and high-dimensional spaces, with the potential to surpass current techniques in revealing novel, high-performance solutions.

Chapter 2
Background

2.1. MATHEMATICAL FOUNDATIONS 4
2.2. EVOLUTIONARY ALGORITHMS 9
2.3. CLUSTERING 11
2.4. QUALITY-DIVERSITY BACKGROUND 15

2.1.Mathematical Foundations

2.1.1. Function

A function f is a mathematical correlation that assigns a distinct element from a set X (commonly referred to as the function's domain) to a unique element in a set Y (known as the function's co-domain) [8]. The aforementioned association may be represented as $f: X \rightarrow Y$ [8].

2.1.2. Discovering the Field of Mathematical Optimization

The domain of function, let's assume f, is frequently used to find the optimal solution. This has led to the emergence of the field of mathematical optimization. The objective of an optimization problem is to maximize or minimize the value of a given function, depending on the specific goal of the problem [9].

The procedure involves determining the optimal solution $x^* \in \mathbb{R}$ within the domain of the function [9]. The constraints differ depending on whether the optimization problem has the objective of maximizing value of its function, meaning that it is a *maximization problem*, or whether it tries to minimize the function value, meaning that it is a *minimization problem* [9]. Formally, the aforementioned categories of optimization problems are denoted as [9]:

1. For Maximization problems: $f(x^*) \geq f(x)$ for all $x \in \mathbb{R}$
2. For Minimization problems: $f(x^*) \leq f(x)$ for all $x \in \mathbb{R}$

2.1.3. Rastrigin Function

The Rastrigin Function is a mathematical function commonly used as a benchmark for optimization algorithms [10]. It is a non-convex function with multiple local minima and a global minimum at the origin [10]. The Rastrigin Function is often used to test the performance of optimization algorithms due to its complex and multi-modal nature [10].

Being one of the most popular benchmark functions in the optimization research, the Rastrigin Function can be characterized by its non-linear and multimodal nature, which can put to the test many for optimization algorithms [11]. The function dates back to 1974, where L.A. Rastrigin introduced a test problem, aiming to assess the efficacy of optimization problems [10].

The Rastrigin function is mathematically defined as follows:

$$f(x) = A \cdot n + \sum_{i=1}^{n} [x_i^2 - A\,cos(2\pi x_i)]$$

where:

- "n" represents the number of dimensions,
- "A" is a constant (usually set to 10)
- "x_i" is the i^{th} component of the input vector x

As explained, the Rastrigin Function is known to possess a multitude of local minima, thereby it poses a significant challenge for optimization algorithms to locate the global minimum, particularly in scenarios where the number of dimensions escalates [10].

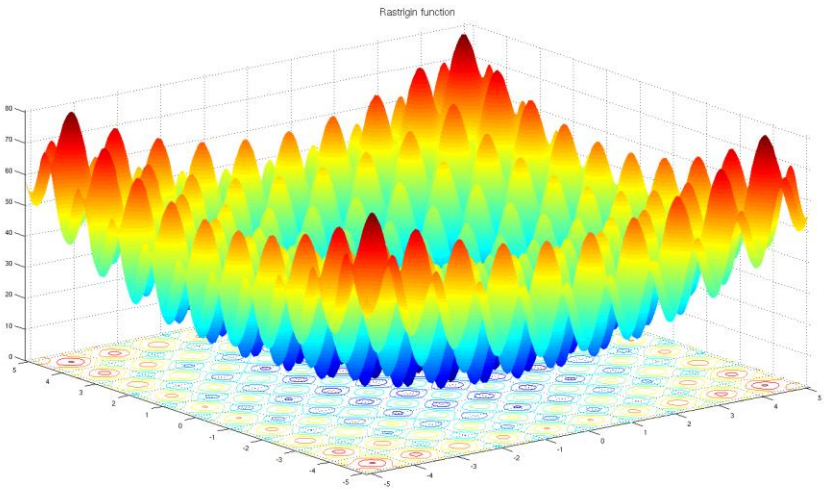


Figure 2.1 Rastrigin Function of two variables (n=2), retrieved from [11]

2.1.4. Distance Metrics

2.1.4.1. Pythagorean Theorem

The Pythagorean theorem, an essential principle in mathematics, relates to right-angled triangles [12]. Mathematically, this theorem can be denoted as an equation also called Pythagorean Equation [12]:

$$c^2 = a^2 + b^2$$

where:

- "a", "b" represents the lengths of the two legs of the right-angled triangle
- "c" represents the length of the hypotenuse

2.1.4.2. Euclidean Distance

The Euclidean distance is one of the most widely used distance metrics. It calculates the distance between two data points in the space [13]. This is done by measuring the straight-line distance between the two [13].

In order to calculate the Euclidean distance ||p - q|| between two data points p and q in a 2-dimensional space, with coordinates (x_1, y_1) and (x_2, y_2) respectively, we use the Pythagorean theorem to calculate the value of the formula given by the square root of the sum of the squares of the differences between the x-coordinates and y-coordinates of the two points [13].

Generalized for a higher dimensional space n, the Euclidean distance ||p - q|| between two data points p and q can be calculated as [13]:

$$\|p - q\| = \sum_{i=1}^{n} \sqrt{(p_i - q_i)^2}$$

In the following sections, we see its importance in determining the similarity or dissimilarity between objects or data points.

2.1.5. Mathematical Field of Probability theory

The study of random phenomena and the quantification of uncertainty are topics covered in the mathematical field of probability theory [14]. It essentially offers a framework for modeling the probability of events and their results.

In this subject, a random process, also known as an experiment, is a method that generates a single outcome from a set of potential outcomes [14]. The collection of all potential results from such an experiment is referred to as the sample, which is subdivided into events [14]. Each event corresponds to one or multiple outcomes [14].

Events are given as probabilities, typically denoted as p, which can take real values ranging from 0 to 1 inclusive [14]. A probability of 0 denotes an impossibility, whereas a probability of 1 denotes certainty in the event. The sample space's total set of probabilities for all potential outcomes equals one [14].

2.1.6. Stochastic Variables

A random variable, also referred to as a stochastic variable, is a quantifiable function associated with a probability space [15]. Random variables can be characterized as real single valued functions that are defined within a probability space. The probability distribution of one is determined by its distribution function [15].

When a random variable assumes a finite number of values, each with the corresponding probabilities, its probability distribution is considered discrete and can be calculated by summing the probabilities of the individual values it assumes [15]. The discrete probability distribution formula is given as:

$$P_x(A) = \sum_{x_n \in A} P_n$$

where:

- "A" is an event in the sample space
- "P_n" is the probability of the value "x_n"

On the other hand, if a function called the probability density (PDF) exists such that the random variable's probability distribution is determined by integrating this function over a given interval, the distribution is referred to as continuous [15]. The formula for the continuous probability distribution is:

$$P_x(B) = \int_B p_x(x)\, dx$$

where:

- "B" is an is an interval in the sample space
- "$p_x(x)$" is the probability density function

2.1.7. Distributions

2.1.7.1. The Uniform Distribution

The Rectangular Distribution, commonly referred to as the Uniform Distribution, is a type of probability distribution in which all possibilities fall within a specified range and have an equal

chance of happening [16].

Mathematically, the PDF of a continuous uniform distribution can be expressed as:

$$p(x) = \begin{cases} \frac{1}{b-a}, x \in [a,b] \\ 0, \quad x \notin [a,b] \end{cases}$$

Where “a” and “b” are the lower and upper bounds of the range, respectively.

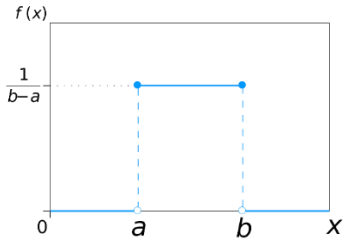


Figure 2.2 PDF of the uniform probability distribution, retrieved from [17]

The Uniform Distribution is frequently used in optimization algorithms and simulations where random sampling from a known range with equal probabilities is required.

2.1.7.2. The Normal distribution

The Gaussian Distribution, also referred to as the Normal Distribution, is a well-known continuous probability distribution recognized for its symmetric bell-shaped curve [18].

The Gaussian Distribution's PDF is formulated as follows [18]:

$$\varphi(x) = \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-(x-\mu)^2/2\sigma^2}$$

In this equation:

- “μ” denotes the mean
- “σ” indicates the standard deviation

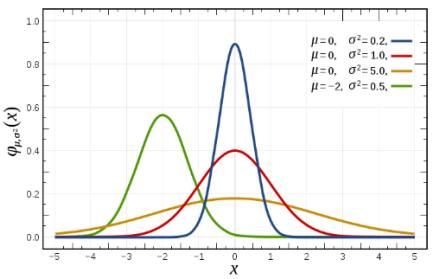


Figure 2.3 displays a variety of Normal Distribution Probability Density Functions (PDFs) with varying means μ, and variances, σ², retrieved from [18]

2.2.Evolutionary Algorithms

2.2.1. Overview of Evolutionary Algorithms

Inspired by nature's evolutionary processes, Evolutionary Algorithms (EAs) are a group of optimization class designed to find the most effective solutions for complex optimization challenges [19]. These challenges often involve determining the ideal value for a specific objective function while following a pre-established set of constraints [19]. While the techniques involved may vary, the core concept behind EAs remains constant. Their ability to discover the best feasible solution makes them highly beneficial for search and optimization tasks [20].

EAs' optimization procedure involves exploring the solution space to locate the most favorable outcome by assessing the fitness value of each potential solution through an evaluation process. To calculate the fitness value of each candidate solution, the evaluation utilizes the fitness function [21]. This value represents the extent to which a solution achieves the goals of the problem being addressed [21]. Solutions with higher fitness values have increased opportunities to reproduce and transmit their genetic material to the following generation, while those with lower fitness values are less likely to do so [21].

2.2.2. Application of Genetic Algorithms

Genetic Algorithms (GAs) are a widely used form of EA that draw inspiration from Charles Darwin's theory about natural selection [21]. They simulate the natural selection process to produce solutions to problems. The standard approach for EAs, including GAs, involves the incremental evolution of a set of candidate solutions over time by applying genetic operators such as selection, crossover, and mutation [21].

In order to employ a genetic algorithm, it is necessary to encode the problem and represent potential solutions as chromosomes, which are commonly binary number strings [22]. The objective is to optimize the fitness function, which may involve either maximizing or minimizing its value [23]. A final solution is considered acceptable upon fulfillment of a specified terminating condition, such as a maximum number of iterations or a fitness threshold [23].

The initial stage of the process involves the creation of a set of potential solutions. Following the creation of the population, each member undergoes an evaluation utilizing a fitness function that is customized to the specific problem being addressed. This fitness function then produces a numeric output that represents the potential of an individual to serve as a viable solution. The generation after that gets determined through the process of selection, through which the individuals that demonstrate the highest level of fitness are selected [21].

To create the population of the next generation, the selected parents are utilized by combining their traits via genetic operators such as crossover and mutation, in order to create the new offspring solutions. Crossover involves combining the genetic material of two parents to create new offspring, introducing diversity into the population through novel genetic material combinations [21]. Mutation, the other genetic operator, introduces random changes to an individual's genetic material, further enhancing population diversity, and it is a common approach to avoid being trapped in local optima [21, 23]. The algorithm must terminate eventually, either after reaching a maximum runtime or a performance threshold, at which point a final solution selected and returned [21].

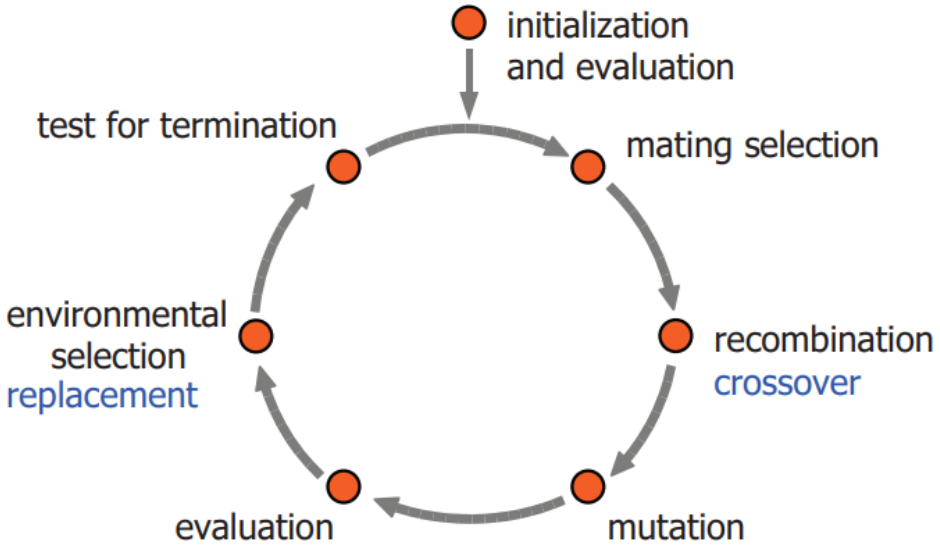


Figure 2.4 The evolutionary cycle, retrieved from [21]

In conclusion, understanding the principles and mechanisms of EAs is necessary and can be used as a strong foundation for exploring more advanced optimization techniques. As we progress to the following sections on Clustering and Quality-Diversity optimization, it is demonstrated how these methodologies utilize the fundamental principles of EAs to generate novel solutions for a diverse set of problems.

2.3.Clustering

2.3.1. Introduction to Cluster Analysis

Cluster analysis is a powerful technique for categorizing objects into groups based on the specific attributes they share [24]. The goal of clustering is to assemble similar objects together while separating dissimilar ones into different groups, thus achieving high internal homogeneity and external heterogeneity for clusters [24].

Figure 2.5 A clustering algorithm aims to identify clusters of similar data by analyzing raw groupings based on shared characteristics, figure retrieved from [25]

The similarity of two objects can be assessed using numerous metrics, which determine their distance in a given space. Euclidean distance is a common metric often used for this purpose [24], which was also touched upon in the Mathematical section earlier.

There are several clustering strategies, each with its own set of advantages and disadvantages. For instance, hierarchical clustering gradually breaks down data until the desired cluster formation emerges [24]. This method can be implemented in either an agglomerative (bottom-up) or divisive (top-down) approach [24]. Another popular method is partitioning clustering, which includes algorithms such as K-Means and seeks to determine a cluster structure that minimizes certain error criteria [24].

In the next section, we will explore in greater detail one of these two highly popular algorithms: The K-Means Clustering.

2.3.2. K-Means Clustering algorithm

The K-Means clustering algorithm is a widely employed partitioning clustering approach that aims to divide a dataset into a predetermined number (k) of distinct, non-overlapping clusters [26]. The center of each cluster signifies the mean of its data points [26]. Placing these centers wisely is crucial, as different locations can yield varying outcomes [26]. This algorithm is especially suitable for scenarios where the number of clusters is known beforehand, and the clusters are anticipated to be spherical and of similar sizes [26]. This section offers a comprehensive understanding of the K-Means clustering algorithm, exploring its underlying principles, the involved steps, and a few of its practical applications.

Before diving into the details of the K-Means clustering algorithm, it is essential to understand

its primary structure and the objective it aims to achieve. The algorithm follows a series of iterative steps to optimize the placement of clusters, ultimately with the objective of minimizing the within-cluster sum of squares [26]:

$$J = \sum_{i=1}^{m} \sum_{k=1}^{K} w_{ik} \left\| x^i - \mu k \right\|^2$$

where:

- '**m**' is the total number of data points in the dataset
- '**K**' is the total number of clusters
- '$\mathbf{x^i}$' is the i^{th} data point in the dataset
- '**μk**' is the centroid of the k^{th} cluster
- '$\mathbf{w_{ik}}$' is a binary indicator variable that equals 1 if the data point x^i belongs to the k^{th} cluster, and 0 otherwise
- '$\left\| \boldsymbol{x^i} - \boldsymbol{\mu k} \right\|^2$' is the squared Euclidean distance between the data point x_i and the centroid μk of the k^{th} cluster

Given a set of data points $X = [x_1, x_2, x_3, ..., x_n]$ and a set of cluster centers $V = [v_1, v_2, ..., v_c]$, the K-means algorithm proceeds as described below [26]:

1. First, 'c' cluster centers are randomly selected from the dataset. Next, the distance between each data point and the cluster centers is computed. Each data point is then assigned to the closest cluster center among all available centers.
2. The cluster centers are subsequently updated using the formula:

$$v_i = (1 / c_i) \sum_{j=1}^{C_i} x_i$$

   In this formula, '$\mathbf{c_i}$' refers to the number of data points in the i^{th} cluster, and '$\mathbf{C_i}$' represents the set of data points in the i^{th} cluster.
3. Following this, the distance between each data point and the revised cluster centers is recalculated. The process continues by iterating between assigning data points to the nearest cluster center and updating the cluster centers until no data points are reassigned.

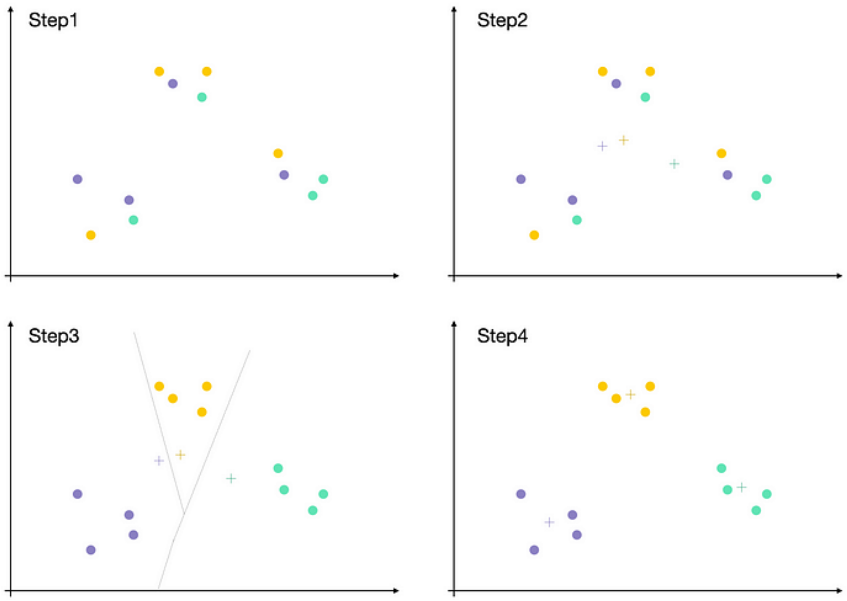


Figure 2.6 Snippets from the development of raw data to clusters, from [27]

A key challenge in implementing the K-Means algorithm is determining the optimal number of clusters (k). Several methods have been proposed to identify the best value for k, such as the elbow method and the average silhouette method [24].

The elbow method, shown in Figure 2.7, is a widely used technique for determining the appropriate number of clusters. It involves examining a plot displaying the within-cluster sum of squares against the number of clusters [24]. The ideal number of clusters is identified at the point where the curve forms a distinct "elbow"-like bend [24]. The rationale behind choosing this point is that adding more clusters beyond the elbow does not yield significant improvements in the clustering structure, as there is no considerable decrease in the within-cluster sum of squares [24].

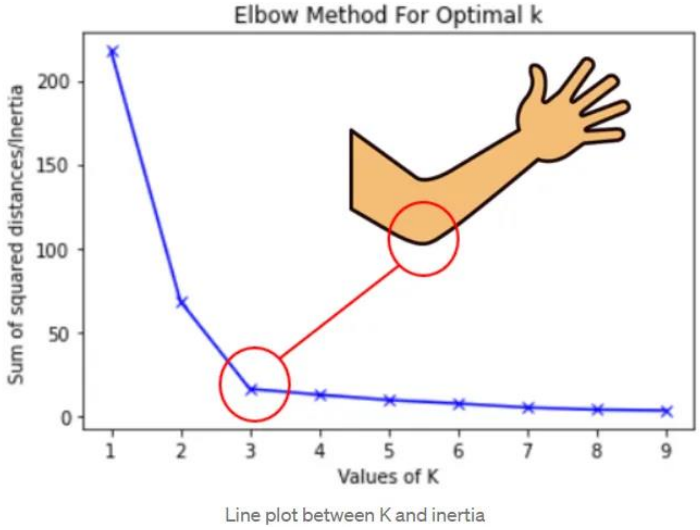


Figure 2.7 Elbow method for figuring the optimal number of clusters, obtained from [24]

Another method for identifying the optimal number of clusters is the average silhouette method. This approach, as shown in Figure 2.8, evaluates the quality of clustering by calculating the average silhouette value for each cluster. A higher average silhouette value is indicative of a superior clustering structure [24]. When using this method, the optimal number of clusters is determined by the point at which the average silhouette value reaches its maximum [24]. By selecting the value of k that maximizes the average silhouette value, a more cohesive and well-separated clustering structure can be achieved [24].

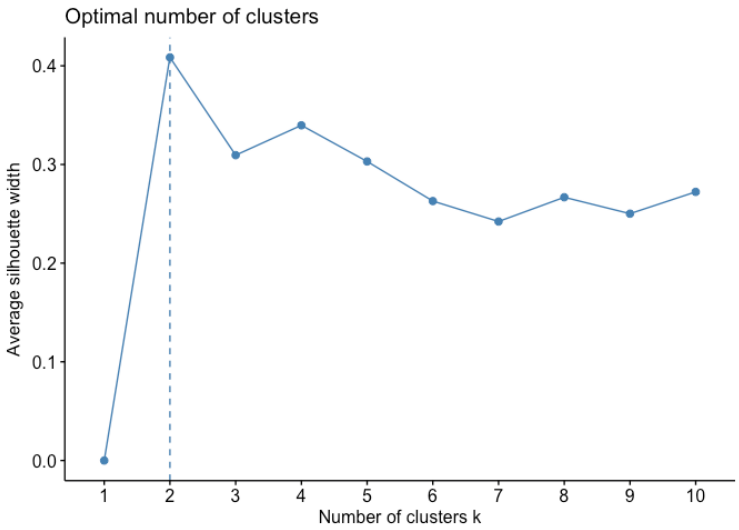


Figure 2.8 The silhouette method, also for figuring the optimal number of clusters, retrieved from [24]

K-means is a straightforward yet effective clustering algorithm with a variety of applications. For instance, it has been employed in medical diagnosis to support doctors' decision-making processes and enhance diagnostic accuracy [27]. In other fields such as those of the delivery systems, K-means can be used to determine the number and location of delivery stores within a region [27]. Additionally, it is commonly utilized for customer segmentation, dividing customers into groups based on similar interests [27]. There are numerous other interesting applications of the K-means algorithm that showcase its versatility.

Despite its usefulness, K-means is not without limitations [27]. From not being well-suited for high-dimensional data, as the data may become sparse in higher dimensions, to being prone to local optima, meaning the clustering result may be affected by the initial cluster assignment, these are just a few of the limitations that make the algorithm far from perfect [27].

2.4.Quality-Diversity Background

2.4.1. Introduction to Quality Diversity algorithms

Quality-Diversity (QD) algorithms represent a novel class of optimization techniques that differ from traditional optimization approaches [1]. Their primary goal is to discover an array of diverse, high-quality solutions throughout the entire solution space, placing equal importance on both the quality and the diversity of the solutions obtained [3]. QD algorithms assume that addressing these two objectives simultaneously is likely to be more efficient than performing independent constrained optimizations.

The motivation behind developing QD algorithms comes from the desire to explore a wide variety of solutions in complex domains, such as robotics. These algorithms have proven to be highly effective in such areas [27]. The evaluation of a QD algorithm's overall effectiveness involves three primary factors. Firstly, the extent of coverage within the feature space, which illustrates the diversity of the solutions [3]. Secondly, the evenness of the coverage in the feature space [3]. And thirdly, the performance of individual solutions found within the feature space [3]. By maintaining diversity among solutions, QD algorithms can facilitate innovation, improve adaptability, and enhance the overall robustness of the system that is being optimized.

QD algorithms are characterized by several key components, including the concept of niches, selection mechanisms, and the balance between exploration and exploitation. Niches allow QD algorithms to focus on distinct regions of the solution space, while selection mechanisms enable the identification of high-quality solutions within those regions [3]. The balance between exploration and exploitation ensures that the algorithm can effectively search the solution space without getting trapped in local optima [3].

QD optimization presents several unique challenges, such as maintaining diversity, avoiding local optima, and efficiently exploring large search spaces [28]. Overcoming these challenges often requires the development of specialized algorithms tailored to the specific requirements of the problem domain. This section serves as an introduction to the broader discussion of QD algorithms and their applications.

2.4.2. QD Problem Formulation

As explained in section 2.4.1, QD optimization aims to discover a diverse set of high-quality solutions in the solution space by simultaneously considering both the quality and diversity of the solutions. The objective function in QD:

$$f_\theta, \hat{b}_\theta \leftarrow f(\hat{\theta})$$

returns a fitness value, and a behavioral descriptor (or feature vector) [29]. Assuming that the fitness function is maximized, the goal in QD optimization is to find for each point $\hat{b} \in B$ the parameters $\hat{\theta}$ that maximize the fitness value:

$$\forall \hat{b} \in B, \qquad \widehat{\theta^*} = \arg\max_{\theta} f_\theta \; s.t. \hat{b} = \widehat{b_\theta}$$

This formulation states that, for each point $\hat{b}$ in the feature space B, we aim to find the parameters $\hat{\theta}$ that provide the maximum fitness value $f(\hat{\theta})$ while satisfying the constraint $\hat{b} = b(\hat{\theta})$, where $b(\hat{\theta})$, is the behavioral descriptor for the given parameters $\hat{\theta}$ [29].

When the BD is two-dimensional, the result is often displayed as a colored image or heatmap [29]. In the context of Evolutionary Algorithms, a QD solution is referred to as an organism, phenotype, or individual, and it serves as the representation used in the algorithm to generate other solutions [30]. The actions performed by the organism, or solution, are the organism's behavior, which creates the previously defined behavioral descriptor. The performance of a QD algorithm is called fitness, and the expression, or function that provides the fitness value is known as the fitness function.

By using this mathematical representation, we can clearly define the problem of QD optimization as the search for parameters $\hat{\theta}$ that maximize the fitness function $f(\hat{\theta})$ while preserving the desired behavior described by the behavioral descriptor $b(\hat{\theta})$ [29].

2.4.3. Illumination

The concept of "illumination" in QD optimization sets it apart from other optimization approaches like Global Optimization, which focuses on finding a single global optimum, and Multimodal Optimization, which aims to discover multiple optima in the parameter space. Illumination algorithms, such as the Multi-dimensional Archive of Phenotypic Elites (MAP-Elites), concentrate on uncovering a diverse set of high-performing solutions throughout the feature space instead of just seeking optimal solutions [29].

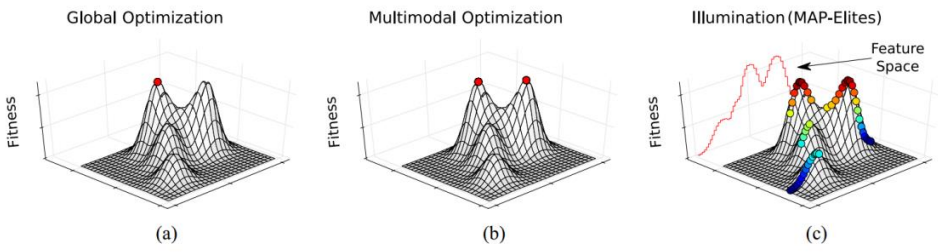


Figure 2.9 Difference between the three Optimization approaches, retrieved from [29]

MAP-Elites was the pioneering algorithm in QD to be classified as an "illumination algorithm," underlining the progression of QD algorithms over time [29]. By illuminating the feature space with the highest performing solutions, illumination algorithms illustrate the trade-offs between performance and the features of the solutions of interest [29].

2.4.4. Initial QD Algorithms

2.4.4.1. Stochastic Sampling

One of the fundamental methods employed across various different problems is stochastic sampling, which involves randomly selecting solutions from the solution space for evaluation using objective and behavioral functions [2]. This technique is primarily used as a benchmark for comparing performance against novel algorithms in diverse areas [2].

2.4.4.2. Novelty Search + Local Competition (NS+LC)

Novelty Search (NS) is an initial Quality-Diversity algorithm introduced by Lehman and Stanley [1], that prioritizes the discovery of innovative solutions over optimizing a specific objective. By calculating the novelty of a solution based on its behavior, NS encourages the exploration of diverse regions within the solution space [31]. This approach has yielded groundbreaking solutions, particularly in complex search spaces where conventional optimization algorithms risk becoming trapped in local optima.

Despite its potential, NS has a key limitation: it overlooks the quality of solutions. To tackle this, Lehman and Stanley introduced the Novelty Search + Local Competition (NS+LC) algorithm [32], which combines novelty search with a local competition mechanism, evaluating both novelty and quality. The NS+LC fitness function considers not only the novelty but also the solution's performance relative to its neighbors, allowing for exploration while maintaining a focus on high-quality solutions [32]. NS+LC promotes diversity in the feature space by ensuring solutions compete on performance only with nearby neighbor solutions, recognizing that each of the solutions might have unique features and constraints [32].

Despite its advantages, NS+LC faces several limitations. Firstly, it requires a computationally demanding neighbor search with a complexity of $n \cdot log(n)$ [33]. Moreover, the algorithm could "cycle" by returning to the same feature space region repeatedly [30]. Last but not least, because NS+LC's search strategy unevenly focused on areas with unknown or useful solutions, it may miss important regions [30]. These shortcomings emphasized the need for more advanced Quality Diversity algorithms capable of overcoming these challenges.

2.4.4.3. Multi-Objective Landscape Exploration (MOLE)

Multi-Objective Landscape Exploration (MOLE) is another initial QD algorithm that seeks to address the QD problem. Proposed by Clune, Mouret, and Lipson [34], MOLE focuses on exploring the search space using multi-objective optimization techniques. By optimizing multiple objectives simultaneously, MOLE aims to find diverse and high-quality solutions [34]. This approach allows the algorithm to efficiently navigate complex search spaces, overcoming the limitations of single-objective optimization methods, and promoting the discovery of innovative solutions [34].

However, MOLE also has some disadvantages. As a multi-objective approach, it may require more computational resources compared to single-objective algorithms [30]. Moreover, determining the appropriate balance between different objectives can be challenging, which might lead to difficulties in finding the optimal trade-off between quality and diversity [30].

2.4.5. Introduction to QD Optimization Framework

The QD optimization framework, developed by Cully and Demiris [28], serves as a unifying modular approach, aiming to simplify the design, implementation, and analysis of the QD algorithms, in order to encourage innovation and progress within the field [28]. The framework makes it easier to develop and customize QD algorithms by providing a clear understanding of the concepts, elements, and procedures that support them, thus promoting the sharing and reusability of algorithmic components.

Algorithm 2 QD-Optimization algorithm (I iterations)

$\mathcal{A} \leftarrow \emptyset$ ▷ *Creation of an empty container.*
for iter = 1 → 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**
    $\{f_\theta, \boldsymbol{b}_\theta\} \leftarrow f(\theta)$ ▷ *Evaluation of the individual and recording of its descriptor and performance.*
    if ADD_TO_CONTAINER(θ, $\mathcal{A}$) **then** ▷ *"ADD_TO_CONTAINER" returns true if the individual has been added to the container.*
      UPDATE_SCORES(parent(θ), Reward, $\mathcal{A}$) ▷ *The parent might get a reward.*
    else
      UPDATE_SCORES(parent(θ), -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 $\mathcal{A}$

Figure 2.10 Pseudocode Snippet of QD Optimization Algorithm, retrieved from [28]

The high-level algorithm consists of the following iterative steps:

1. Generate a new set of candidate solutions, known as offspring, by applying selection operators on a population of existing solutions.
2. Evaluate the performance and behavior of each offspring using an objective function and a behavioral function, respectively.
3. Update the container, a data structure used for storing discovered solutions, by potentially adding the new offspring based on their performance and behavioral scores.
4. Update the population scores, which guide the selection and exploration of the solution space.

By following these steps and making informed choices for the container, selection operators, and population scores, researchers and practitioners can tailor QD algorithms to suit their specific needs and application domains. The QD optimization framework, therefore, provides a flexible and systematic approach to the study and development of QD algorithms.

2.4.6. Key Components of the QD Optimization Framework

The QD optimization framework is composed of several critical components that shape the behavior and performance of QD algorithms. These components include among others the *containers*, the *selection operators*, and the *population scores*.

2.4.6.1. Containers

Containers are data structures used to store discovered solutions in QD algorithms. Two primary types of containers used are N-Dimensional Grid and Centroidal Voronoi Tessellation containers. The N-Dimensional Grid containers partition the behavioral space into cells of an N-dimensional grid and store the highest performing solutions within each cell [6]. Each cell represents a unique region of the behavior space, and the grid-like structure enables for a more efficient organization and retrieval of solutions. Gaier et al. [35] applied this container type in the surrogate-assisted illumination of aerodynamic design exploration, while Justesen et al. [36] utilized it in MAP-Elites for noisy domains by adaptive sampling.

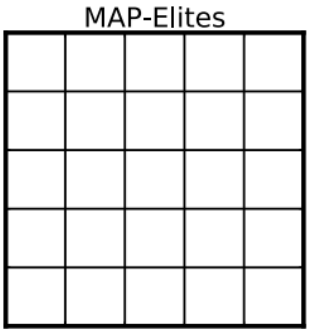


Figure 2.11 MAP-Elites container, adapted from [6]

Vassiliades et al. [6] also introduced the Centroidal Voronoi Tessellation container, which employs Centroidal Voronoi Tessellations to scale up the MAP-Elites algorithm. CVT-based containers dynamically partition the behavior space into irregularly shaped cells, adapting to the underlying structure of the solution space [6]. This approach results in a more balanced coverage of the space, leading to better performance and diversity in the discovered solutions. By using this container, the MAP-Elites algorithm can effectively cover larger solution spaces, thus improving the overall performance of the algorithm [6].

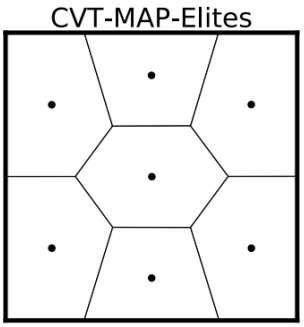


Figure 2.12 CVT-MAP-Elites container, adapted from [6]

2.4.6.2. Selection Operators

Selection operators also play a crucial role in QD algorithms, as they influence how candidate solutions are selected from the population to generate the offspring solutions. Two important selection operators are the *Uniform Random* and *Score Proportionate* selections, where each offers distinct advantages and exploration-exploitation trade-offs.

The Uniform Random Selection operator assigns equal probabilities to all candidate solutions when selecting them for the offspring generation [30]. This approach ensures an unbiased exploration of the solution space, enabling the algorithm to consider a wide range of potential solutions. On the other hand, the Score Proportionate Selection operator prioritizes high-scoring solutions by selecting candidates with probabilities proportional to their scores [30]. By favoring promising solution areas, this method enhances the exploitation of high-performing regions within the solution space.

2.4.6.3. Population Scores

In QD algorithms, population scores are critical guiding indicators for the selection and exploration stages. Among the various scores employed in QD algorithms, *fitness*, *novelty*, and *curiosity* scores are particularly common and informative.

The fitness score evaluates a solution's performance in relation to the objective function of the problem [30]. The novelty score evaluates a solution's uniqueness in comparison to other solutions in the population [30]. Finally, the curiosity score assesses a solution's ability to lead to the discovery of novel, high-performing solutions [30].

2.4.7. Evaluation Metrics for Assessing QD Algorithms' Success

Evaluating the success of QD algorithms involves various metrics, with two fundamental and commonly used criteria providing a solid foundation for assessing their performance.

The first criterion, *performance*, measures the quality of solutions discovered by the algorithm, typically based on objective function values [2]. This metric is essential for measuring the algorithm's ability to converge towards optimal or high-performing solutions and attaining the desired optimization goals.

The second criterion, *coverage*, examines the algorithm's capacity to explore diverse areas of the solution space and generate a broad range of high-performing solutions [2]. As the pursuit of diversity is highly important to QD optimization, this metric offers valuable insights into the algorithm's effectiveness in achieving its objectives and promoting innovation across the solution space.

2.4.8. Elite Hypervolume

The Elite Hypervolume is a concept introduced by Vassiliades and Mouret [4] in the context of QD optimization. It refers to a specific subset of the genotypic space that includes the highest performing solutions, or elites, within the search space [4]. This section will dive deeper into the Elite Hypervolume, discussing its definition, rationale, and its impact on the development and enhancement of QD algorithms.

Vassiliades and Mouret [4] observed that the best performing solutions identified by MAP-Elites tend to be concentrated in a particular region of the genotypic space. This phenomenon occurs because many genotypes can map to the same cell in the feature space, given the non-linear relationship between a genotype and its corresponding behavior [4]. Therefore, the top performing solutions are often located in close proximity to each other. This insight led to the introduction of the Elite Hypervolume concept, which characterizes the regions of the genotypic

space that contain these high-performing solutions.

Mathematically, the Elite Hypervolume is defined as [4]:

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

It includes a set of m individuals, each representing the highest performing solution (elite) in its corresponding area or niche within the feature space [4]. In this equation, Ci is a subset of X that represents the portion of the genotype space corresponding to the i^{th} region in the feature space. The index i ranges from 1 to q, with q being less than or equal to k (k being the niche capacity). The goal of a QD algorithm incorporating the Elite Hypervolume is to identify the set E, as computing H is computationally demanding [4].

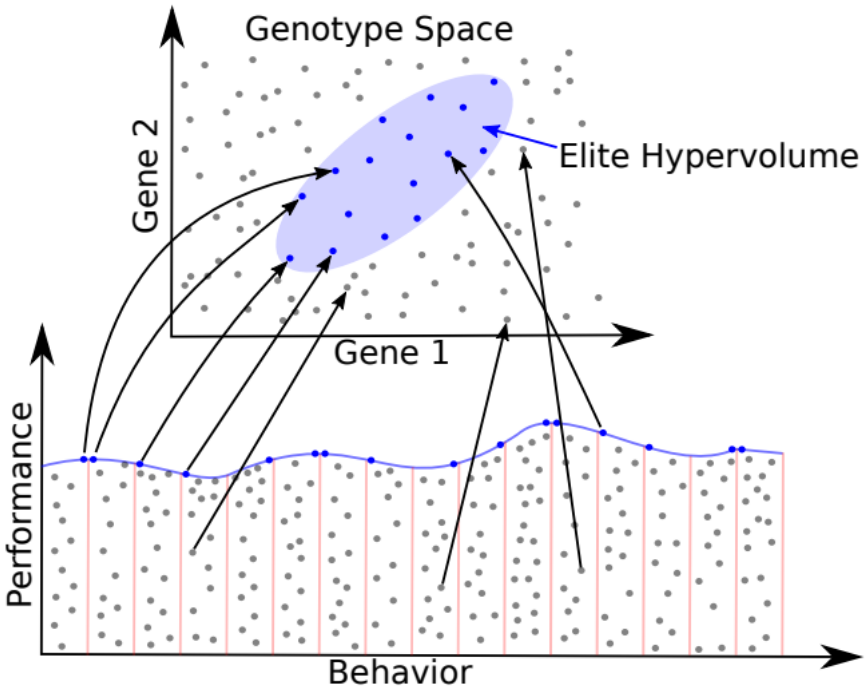


Figure 2.13 Illustration of the Elite Hypervolume, retrieved from [4]

In Figure 2.13, the Elite Hypervolume is illustrated within the genotypic space, showing the distribution of high-performing solutions (elites) surrounded by the Elite Hypervolume. The figure demonstrates how these elites are concentrated in a specific area of the genotypic space, validating the observations made by Vassiliades and Mouret [4]. By visually representing the Elite Hypervolume, this image provides a clearer understanding of the concept and its implications for developing and enhancing QD algorithms.

Understanding and leveraging the properties of the Elite Hypervolume has significant influence

in the design and improvement of QD algorithms, particularly concerning MAP-Elites and its directional variation operators [4]. By effectively utilizing the characteristics of the Elite Hypervolume, it is possible to develop more efficient and powerful strategies for discovering high-performing solutions in the search space.

In the subsequent sections, we will examine the MAP-Elites algorithm in more detail, focusing on how the Elite Hypervolume concept is integrated with directional variation operators to enhance the algorithm's performance and effectiveness in solving optimization problems.

2.4.9. Multi-dimensional Archive of Phenotypic Elites (MAP-Elites)

The MAP-Elites algorithm is a well-known Quality-Diversity algorithm introduced by Mouret and Clune [2]. It aims to discover a diverse set of high-performing solutions in the search space while considering multiple objectives or features [2]. This section will provide an in-depth overview of MAP-Elites, its algorithmic structure, important applications, and the development of variations that have been developed since its introduction.

MAP-Elites is designed to maintain an archive of high-performing solutions, where each solution is assigned to a cell in an N-dimensional grid based on its feature values [2]. The algorithm's primary objective is to find the best performing solution for each cell, resulting in a diverse set of high-performing solutions across the search space [2]. The archive is continuously updated as the algorithm iterates, replacing solutions within cells if a better-performing candidate is found.

The MAP-Elites algorithm consists of the following steps [2]:

1. Initialize an empty archive with a predefined grid structure based on the problem's feature dimensions.
2. Generate an initial population of candidate solutions.
3. Evaluate the performance and features of each candidate solution.
4. Insert each candidate into the corresponding cell in the archive if the cell is empty or if the candidate outperforms the current occupant.
5. Select parents from the current archive using a selection operator (e.g., random selection, tournament selection).
6. Apply variation operators (mutation, crossover) to the selected parents to generate offspring.
7. Evaluate the offspring's performance and features, and then insert them into the archive following the same rules as in step 4.

8. Iterate steps 5-7 for a predefined number of generations or until a stopping criterion is met.

For a clearer understanding of the MAP-Elites algorithm, refer to Figure 2.12, which presents a pseudocode description of its simple, default version. This pseudocode outlines the key steps involved in MAP-Elites, including initialization, evaluation, archive updating, and the use of variation operators to generate offspring solutions. By following this algorithm, MAP-Elites can effectively explore the search space and maintain a diverse set of high-performing solutions [2].

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

Figure 2.14 MAP-Elites algorithm pseudocode, adapted from [2]

MAP-Elites has been applied to various problem domains, showcasing its versatility and effectiveness. The algorithm has proven capable of uncovering high-performing solutions across diverse areas of the search space, fulfilling its QD objectives.

As shown in Figure 2.15, MAP-Elites searches inside a high-dimensional space for the best performing solutions in a low-dimensional feature space defined by user-selected dimensions. As further detailed in section 2.4.3., the term "illumination algorithm" refers to its ability to illuminate the fitness potential of different areas within the feature space, taking into account trade-offs between performance and features of interest.

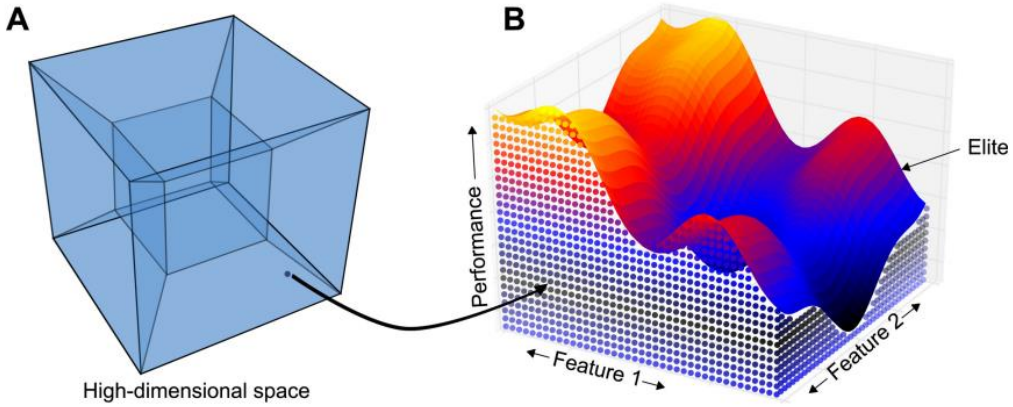


Figure 2.15 MAP-Elites algorithm visualized, revealing optimal solutions in user-defined dimensions while considering trade-offs between performance and features, retrieved from [2]

Since the introduction of MAP-Elites, researchers have developed several variations to address specific challenges or improve its performance. One area of focus has been the development of novel emitters, particularly directional variation operators, to guide the exploration and exploitation of the search space more effectively. These emitters and directional variation operators will be discussed in greater detail in the following sections.

2.4.10. Emitters

Emitters are an innovative concept introduced by Vassiliades and Mouret [4] in the context of the MAP-Elites algorithm. They facilitate efficient exploration of the search space by leveraging the elite hypervolume and interspecies correlations [4]. These emitters generate new candidate solutions by sampling the search space and directing the search towards regions with higher potential for improvement. By doing so, they enhance the overall performance of the algorithm [4].

2.4.10.1. Introduction to the concept of Emitters

A key component of emitters is directional variation operators, which allow them to produce new candidate solutions by perturbing existing ones in the search space [4]. These operators enable the search process to be guided in specific directions, leading to a more targeted exploration and exploitation of the search space. In their work, Vassiliades and Mouret [4] proposed three types of directional variation operators, namely *Iso*, *LineDD*, and *Iso+LineDD*.

2.4.10.2. Directional Variation Operators

The **Iso emitter** is the most straightforward of the three proposed operators. It works by applying an isotropic Gaussian distribution to collect new candidate solutions around a single genotype, ignoring the second elite. MAP-Elites algorithm [4] employs this operator to ensure that the search process keeps its attention on a particular direction for efficient exploration of the search space and facilitating the discovery of high-performing solutions.

The **LineDD (Distance-Dependent Line Variance) emitter** builds on the Iso emitter by leveraging correlation among two elites to sample a new candidate solution near the first genotype, limited to the line connecting both elites [4]. As a result, by considering multiple directions, the LineDD emitters can more efficiently identify regions of the search space that contain elite solutions and provide a more comprehensive exploration.

The third emitter called **Iso+LineDD** is a combination of the Iso and LineDD operators, combining their individual advantages. It produces a new candidate solution from two-parent elites by sampling near the first elite using an isotropic Gaussian distribution, while biasing the search towards the direction of correlation of the second elite [4]. This strategy takes into account the correlation between the two elites, allowing for a more targeted exploration of the search space and the discovery of solutions with higher performance.

To implement the Iso+LineDD emitter, first, the emitter selects two elite genotypes x_i and x_j uniformly from the container. Then, it generates a new genotype x_i' by calculating the result of the variation formula as follows [4]:

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

This operator's direction is considered positive ($x_j - x_i$), but a negative direction ($x_i - x_j$) could also be used to investigate various regions of the search space [4].

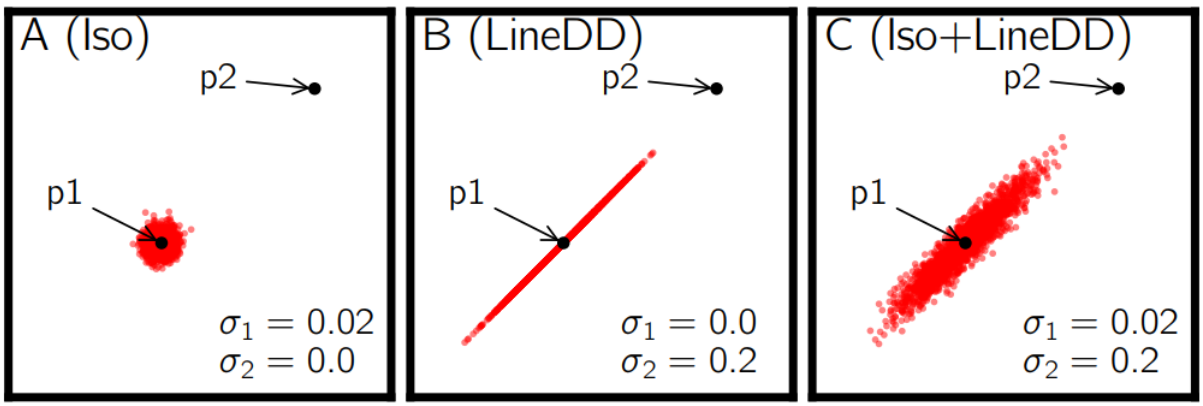


Figure 2.16 Iso, LineDD & Iso+LineDD emitters' approaches, retrieved from [4]

Figure 2.16 illustrates the three emitters' approaches to biasing the mutation of an elite p_1 in the direction of correlation with another elite p_2 [4]:

A. Approach A, as shown in the figure, involves the Iso emitter sampling from an isotropic Gaussian distribution that is centered at p_1. This approach disregards any correlations between the two points.

B. Approach B on the other hand, demonstrates the LineDD emitter sampling along the correlation direction between p_1 and p_2 with a variance that depends on the distance. This approach concentrates on exploring the search space in a more focused manner, but it does not investigate all directions surrounding p_1.

C. Regarding the approach denoted as C in the figure, it demonstrates the utilization of Iso+LineDD emitter, that samples from a multivariate Gaussian distribution with p1 as the center and p2 as the direction vector. This approach now enables the algorithm to explore the search space surrounding p1 while also exploiting the correlations between p1 and p2, thereby leading to a more successful search process.

Chapter 3
Methodology

3.1. Problem Formulation 28
3.2. Proposed Clustering-Based Emitter 29
3.3. JAX and QDJAX Library 31
3.4. K-Means for JAX Implementation 31
3.5. Implementation 34
3.6. Experimental Setup 44

3.1. Problem Formulation

As mentioned in chapter 2, QD algorithms have gained significant attention in recent years as a powerful approach for exploring and optimizing complex and high-dimensional search spaces. Instead of focusing on finding a single optimal solution, QD algorithms aim to discover a diverse set of high-performing solutions, providing valuable insights into the potential behaviors of a system across a wide range of characteristics [3]. A key component of QD algorithms is the emitter, which is responsible for generating new solutions by guiding both the exploration and the optimization, and ultimately shaping the algorithm's ability to effectively navigate the search space [4].

A major challenge faced by QD optimization algorithms, such as MAP-Elites, is exploring and optimizing regions within the search space that exhibit complex geometries, non-linear relationships between variables, and complex structures [5]. Traditional QD algorithms, such as MAP-Elites, rely on user-defined bounds and grids in the behavior space to guide exploration and optimization [5]. While this approach can be effective in certain scenarios, it may fail to capture the underlying structure of more complex search spaces, resulting in inefficient exploration and exploitation [5].

To address this issue, there is a growing interest in developing more adaptive algorithms that can better understand and leverage the structure of the search space, ultimately leading to improved exploration and optimization efficiency. In this study, we propose a novel method for enhancing the exploration and optimization capabilities of QD algorithms by modifying the Iso+LineDD emitter. We have chosen to modify the Iso+LineDD emitter in particular, as it is designed to handle challenging search spaces, making it well-suited for the task of improving exploration and optimization in complex domains [4]. Our approach involved incorporating clustering techniques, such as K-Means, for a more targeted search process.

The primary goal of our proposed clustering-based emitter is to enable the QD optimization algorithm to better adapt to the complex geometries of complex search spaces. By employing clustering techniques to identify more promising areas of the archive, where clusters of similar solutions are found, our proposed emitter aims to discover a diverse set of high-performing solutions potentially overlooked by other approaches.

In this research, we use the QD optimization framework and the Iso+LineDD emitter to investigate the challenges of complex search spaces further. By using the K-Means clustering algorithm, our focus is to incorporate clustering techniques into the emitter's selection process. By doing so, our aim is to improve the exploration and exploitation capabilities of the QD optimization algorithm, facilitating the discovery of high-performing solutions and providing a more comprehensive understanding of a system's potential behaviors.

As we proceed with our discussion in this chapter, we will provide a detailed overview of our proposed clustering-based emitter and describe its algorithm implementation within the QD optimization framework. We will also discuss the experimental setup, performance evaluation, and comparison of our clustering-based emitter to other existing emitters, including the Iso+LineDD emitter. Through this analysis, we aim to demonstrate the advantages of incorporating clustering techniques into the emitter in order to effectively address the challenges posed by complex search spaces.

3.2. Proposed Clustering-Based Emitter

In this section, we introduce our proposed clustering-based emitter, which aims to enhance the exploration and exploitation capabilities of the QD optimization framework. This new emitter builds upon the principles of the Iso+LineDD emitter and employs clustering techniques for a more efficient and targeted search in the solution space.

The primary objective of our clustering-based emitter is to enable the QD optimization algorithm to better adapt to the tricky search spaces characterized by complex geometries. By leveraging clustering techniques, our proposed emitter seeks to explore and exploit the search space more efficiently, leading to the discovery of a diverse set of high-performing solutions that may not have been identified previously by other similar approaches.

In this thesis, our primary focus is on the implementation and testing of the clustering-based emitter utilizing the K-Means clustering algorithm. It is worth noting that our original plan was to start with K-Means clustering and then, depending on the outcomes, optimize the clustering algorithm or potentially employ alternative clustering techniques such as Hierarchical clustering.

Unfortunately, we encountered unexpected challenges during the setup of our experiments, which resulted in a substantial loss of time. To maintain adherence to our project deadlines, we strategically decided to concentrate solely on K-Means clustering. With these experiences and findings in hand, we are able to see many potential opportunities for future research to investigate the applicability of other clustering algorithms as well.

The clustering-based emitter operates by integrating clustering techniques into the generation process of offspring solutions while retaining the same offspring generation formula as the Iso+LineDD emitter. The key distinction lies in the method of selecting elites used to produce offspring solutions. Instead of randomly selecting two elites from the archive, as the Iso+LineDD emitter does, our approach aims to incorporate clustering within the non-empty solutions of the archive at each generation before the selection.

Specifically, in our proposed approach, we introduce a two-step process for selecting elites to generate offspring solutions. The first step involves uniformly sampling a small number of elites from the archive, which will serve as the foundation for generating new offspring. These elites are chosen randomly to maintain the algorithm's diversity and will later be perturbed towards the direction of correlation of other elites identified through the K-Means clustering process in each iteration.

The second step involves applying K-Means clustering to the archive, generating clusters of elites that illustrate similar features. This enhancement to the Iso+LineDD emitter’s implementation is hoped to allow for more effective guidance in generating new diverse solutions. Instead of creating offspring solutions that go in random directions, we focus on perturbing the selected elites towards the direction of correlation of other elites located closer to the cluster centers, where high-performing elites of the elite hypervolume are more likely to be concentrated [4]. It is hoped that these cluster centers can better represent the features characterizing the elites within the selected clusters. Thus, by employing this modification, we aim to use these elites to generate offspring solutions with better diversity and higher fitness levels.

We evaluate two distinct strategies for perturbing towards the direction of correlation of the baseline elites selected in Step 1. The first strategy focuses on utilizing intra-cluster members for the variation process, specifically the elites closest to the cluster centers, as these solutions are the most representative of each cluster's characteristics and performance. The second strategy involves using the cluster centroids as the target for perturbation. These centroids represent the average position of the elites within each cluster, capturing the central tendencies of the grouped solutions and their characteristics [26].

By incorporating either of these strategies, we anticipate enhancing the algorithm's performance metrics, leading to improved exploration and exploitation capabilities in the search space. Our approach targeted to provide a more effective method for navigating complex problem domains and discovering diverse, high-performing solutions.

3.3. JAX and QDJAX Library

In today's world, due to the rapid growth of areas such as Artificial Intelligence and Machine Learning, there is an unprecedented need for hardware acceleration [30]. This need arises from the necessity to manage and process the vast amounts of data used in tasks within this field [30]. During the investigation of techniques to improve the performance of the used hardware, it was discovered that GPUs offered a significant improvement in performance, especially when compared to CPU. This discovery has led to the development of machine learning frameworks like JAX, which take advantage of GPUs and other hardware to improve their performance, further accelerating progress in the fields of machine and deep learning [30].

The QDJAX library is a specialized implementation of QD algorithms within the JAX framework. The aim of the library is to facilitate the exploration of the potential benefits of specialized hardware in the field of Quality Diversity by researchers. It is an experimental tool for investigating novel ideas, offering the flexibility to be easily modified based on the goals of the experiment at hand. It is currently being supervised by the CYENS Research Center of Excellence - Learning Agents and Robots (LEAR) MRG.

The proposed new clustering-based emitter leverages the capabilities of the QDJAX library, which is built on the JAX framework. This library supports a variety of QD emitters, making it a versatile tool for conducting experiments. The library is also used to evaluate the performance of the proposed emitters with common test functions in optimization and machine learning.

3.4. K-Means for JAX Implementation

This section provides a more comprehensive explanation of the JAX-based K-Means implementation utilized in the emitter, as discussed in Section 3.2. With the goal of fulfilling the specific requirements of this study, this K-Means implementation for JAX has been adapted from an existing GitHub repository (https://github.com/creinders/ClusteringAlgorithmsFromScratch). The algorithm’s implementation is divided into three main classes: **BaseAlgorithm**, **KMeansBase**¸ **KMeansJax**. For reference and more details, refer to Appendix A.1 which includes the complete implementation.

The **BaseAlgorithm** class lays the groundwork for further algorithm implementations. It includes two main attributes: *verbose*, which is a Boolean flag that controls the display of progress information during algorithm execution; *callback*, which is an optional callback function invoked at certain points throughout the optimization process.

The **KMeansBase** class, derived from the BaseAlgorithm class, sets the foundation for K-Means clustering algorithms. It introduces new attributes specific to K-Means: *n_clusters* representing the number of clusters to create, *max_iter* indicating the maximum number of iterations to perform, *early_stop_threshold* serving as a threshold for early stopping when the change in centroids falls below this value, and *rng* as a random number generator utilized for initializing centroids.

Additionally, this class implements methods like **init_clusters**, which initializes centroids by randomly selecting data points; **prepare**, which reads input data and initializes centroids; and **fit**, which applies the K-Means algorithm to input data, returning centroids and cluster assignments. The **_main_loop** method, responsible for defining the main loop of the K-Means algorithm, is left to be implemented by subclasses.

The **KMeansJax** class inherits from KMeansBase and offers a JAX-based K-Means clustering algorithm. The method **init_clusters** has been adapted to override the original, with the aim to allow centroid initialization either randomly or by accepting them as parameters. This change enables faster convergence and increased control over initial centroids in our implementation. Furthermore, the **fit** method is overridden to accept the initial centers as optional parameters.

In essence, two main modifications were made to the original K-Means implementation. Firstly, a method named **closest_points_to_center** was added to the KMeansJax class, that calculates the closest points from a specific cluster center after applying the K-Means algorithm to the data. This method uses specific terms, which are explained as follows:

1. Mask: A mask is essentially a binary array that functions to filter or select particular elements from another given array (in this scenario data points), and it does so based on certain conditions, where if an element meets the condition, it receives a value of 1. Otherwise, it receives a value of 0.
2. Large value mask: To prevent specific data points from having a significant impact on later calculations within the input array of data points, it was necessary to create a large value mask and assign those particular elements with higher numerical values to eliminate

any influence from external points beyond the given cluster *i*.

3. Masked data points: Masked data points are the result of applying a mask to an array of data points. In this case, masked data points are obtained by adding the values of the input data array *X* and the large value mask *L* for each corresponding cell, effectively removing the influence of points outside cluster *i* on the distance calculations.

The algorithm for determining the closest points to a cluster center is as follows:

Algorithm 1 Closest Points to Cluster Center

Procedure GetClosestPointsToCenter($\mathbf{X}, \mathbf{A}, \mathbf{C}, i, k$)
Input: data matrix $\mathbf{X}$, assignments vector $\mathbf{A}$, cluster centroids matrix $\mathbf{C}$, cluster index i, number of closest points k

1: Create a mask $\mathbf{M}$ to identify the data points that belong to the specified cluster i:

$$\mathbf{M}_j \leftarrow (\mathbf{A}_j == i) \quad \forall j \in 1, 2, \ldots, |\mathbf{A}|$$

2: Create a large value mask $\mathbf{L}$, assigning a large value to the points outside cluster i and the value 0 to the points inside the cluster

$$\mathbf{L}_j = \begin{cases} 0, & \text{if } \mathbf{M}_j \text{ is True (point belongs to cluster } i) \\ 10^9, & \text{if } \mathbf{M}_j \text{ is False (point does not belong to cluster } i) \end{cases} \qquad \forall j \in 1, 2, \ldots, |\mathbf{A}|$$

3: Compute masked data points:

$$\mathbf{Y} = \mathbf{X} + \mathbf{L}$$

4: Calculate the squared Euclidean distance $\mathbf{D}$ between the data points $\mathbf{Y}$ and the centroid $\mathbf{C}_i$ of the specified cluster i:

$$\mathbf{D} = \sum_{j=1}^{n} (\mathbf{Y}_j - \mathbf{C}_{ij})^2$$

5: Sort $\mathbf{D}$ in ascending order and get the indices $\mathbf{I}$ of the sorted array
6: **return** the top k closest data points in $\mathbf{X}$, indexed by the first k elements of $\mathbf{I}$

Figure 3.1 Pseudocode of the method that returns the k closest points of the specified cluster center

The procedure starts by creating a mask, *M*, which identifies data points belonging to the specified cluster *i*. Subsequently, a large value mask, *L*, is created, assigning a large value to points outside cluster *i* to eliminate their influence when calculating the closest points. The masked data points, *Y*, are then computed by adding the values of both the input data array *X* and the large value mask *L* for each corresponding cell. Next, the squared Euclidean distance, *D*, between the masked data points *Y* and the centroid C_i of the specified cluster *i* is calculated. The distance is computed as the sum of the squared differences between each data point y_j and the corresponding centroid coordinate c_{ij} for all *j* from 1 to *n*. Once *D* is obtained, it is sorted in ascending order, and the indices *I* of the sorted array are extracted. The procedure then returns the top *k* closest data points the specified cluster *i* from the input array *X*.

The second modification in the K-Means implementation, as mentioned before, involves

adapting the algorithm to support the initialization of cluster centers either randomly or by passing them as parameters. In the context of the emitter, the K-Means algorithm is called multiple times. During the first iteration, the centers are initialized randomly. However, for subsequent iterations, the initial centers are set to the returned centers of the previous iterations. In section 3.5, an in-depth analysis of the introduction of clustering techniques to the selection process of the emitter is discussed. For now, it should be noted that our aim behind this approach is to accelerate the process by utilizing the knowledge gained from previous iterations and incrementally finding the optimal clusters in the input data with fewer maximum iterations. Consequently, the algorithm converges faster, reducing the computational overhead and enhancing the execution time performance of the emitter.

3.5. Implementation

3.5.1. Implementation of Iso+LineDD Emitter on QDJAX

This section dives deeper into the implementation of the Iso+LineDD emitter within the QDJAX library, highlighting its integration within the framework and the various components involved. The complete implementation of the emitter can be found in Appendix A.2.

The algorithm of the Iso+LineDD emitter consists of the following steps:

Algorithm 2 Iso+LineDD Emitter

Require: archive $\mathcal{A}$, iterations T, offspring count N_{off}, isotropic variance σ_1, directional variance σ_2
Ensure: Updated archive $\mathcal{A}$

Initialize archive $\mathcal{A}$ with random solutions
for $t = 1$ to T **do**
  for $k = 1$ to $batch\_size$ **do**
    Sample two elites $\mathbf{x}_i$ and $\mathbf{x}_j$ uniformly from $\mathcal{A}$
    Generate new offspring $\mathbf{x}_k$ by perturbing x_i according to the formula:

$$\mathbf{x}_k = \mathbf{x}_i + \sigma_1 \cdot \mathcal{N}(0, I) + \sigma_2 \cdot (\mathbf{x}_j - \mathbf{x}_i) \cdot \mathcal{N}(0, 1)$$

    Evaluate fitness and behavioral descriptor of $\mathbf{x}_k$: $\mathbf{f}_k$, $\mathbf{b}_k$
    Update $\mathcal{A}$ with $\mathbf{x}_k$ using the container's update function

Figure 3.2 Pseudocode of the Iso+LineDD Emitter

The QDJAX implementation of the emitter consists of three distinct classes within the module: **EmitterParams**, **EmitterState**, and **IsoLineEmitter**. These classes serve various purposes, such as defining the parameters and states used by the emitter and implementing the core functionality of the emitter:

1. The **EmitterParams** class defines the parameters used by the algorithm and allows for manual modification outside the emitter. This class includes the initial solution vector *x0*, isotropic variance *iso_sigma*, directional variance *line_sigma*, and the lower and upper bounds of the search space.
2. The **EmitterState** class defines the state of the emitter that should exclusively be used and modified by the emitter itself. In the case of the Iso+LineDD emitter, this class is left empty as there is no need for specific state information.
3. The **IsoLineEmitter** class serves as the primary class that implements the Iso+LineDD emitter. It inherits from the EmitterBase class and overrides necessary methods to deliver the specialized functionality of the emitter. The **ask** and **tell** methods are the key functions driving the emitter's behavior.

The **ask method** is in charge of generating new solutions according to the outlined algorithm. It begins with the selection of two elite solutions from the archive, *xi* and *xj*. Subsequently, it generates new offspring by perturbing *xi* as per the provided formula. The new solutions are clipped to ensure they stay within the specified bounds of the search space.

The **tell method** is used to update the archive with newly generated solutions, their fitness values, and behavioral descriptors. The function updates the archive with the help of the **add_to_archive** function.

To initialize the Iso+LineDD emitter, the **init method** is invoked with the appropriate random number generator key. This method sets the default EmitterParams and EmitterState values for the emitter instance.

The following sections detail the modifications made to the emitter's selection process and the different stages that are followed to accomplish the intended functionality.

3.5.2. Adapting Iso+LineDD Emitter through Reduced Batch Sampling

In this section, we introduce the first modification to the Iso+LineDD emitter as an intermediate step for incorporating clustering techniques in future work. This modification is focused on changing the selection process of elites during offspring generation to optimize the integration with clustering, which is better explained in the following section.

The original Iso+LineDD emitter uniformly samples two elites from the solution space for each offspring generation. In the QDJAX library, two sets of elites, *B1* and *B2*, are randomly selected from the solution space. Both sets contain an equal number of elites. For each offspring, an elite from *B1* and another from *B2* are chosen to undergo the variation process.

In the modified implementation, the sampling method for the *B1* set of elites was adjusted. Instead of selecting an equal number of elites as in *B2*, the emitter now targets a smaller number of elites to include in *B1*. Consequently, the elites in *B1* will be used for generating multiple offspring solutions, unlike in the original implementation where each elite is used only once. The motivation behind this modification is to maintain diversity in elite selection while targeting more promising regions in the solution space in the next step. When K-Means clustering was introduced later, this approach was expected to generate diverse, high-fitness solutions, as the offspring solutions would be better characterized by the traits of the elites sampled through the K-Means clustering. The decision behind the selection of different batch sizes of elites for *B1*, was implemented to allow more flexibility in experimentation with different *B1* batch sizes, which can help to determine the optimal size for guiding the emitter in exploring more effectively the promising regions.

Algorithm 3 Modified Iso+LineDD Emitter (Without Clustering)

Require: archive $\mathcal{A}$, iterations T, number of elites for variation generation N_e, offspring count N_{off}, isotropic variance σ_1, directional variance σ_2
Ensure: Updated archive $\mathcal{A}$

Initialize the archive $\mathcal{A}$ with random solutions
for $t = 1$ to T **do**
  Sample a batch of N_e elites uniformly from $\mathcal{A}$ to form $\mathcal{B}_1$
  Sample a batch of N_{off} elites uniformly from $\mathcal{A}$ to form $\mathcal{B}_2$
  for $k = 1$ to N_{off} **do**
    Select elites $\mathbf{x}_i \in \mathcal{B}_1$ and $\mathbf{x}_j \in \mathcal{B}_2$
    if $N_e < N_{off}$ **then**
      Use elites from $\mathcal{B}_1$ multiple times
    Generate new offspring $\mathbf{x}_k$ by perturbing $\mathbf{x}_i$ according to the formula:

$$\mathbf{x}_k = \mathbf{x}_i + \sigma_1 \cdot \mathcal{N}(0, I) + \sigma_2 \cdot (\mathbf{x}_j - \mathbf{x}_i) \cdot \mathcal{N}(0, 1)$$

    Evaluate fitness and behavioral descriptor of $\mathbf{x}_k$: $\mathbf{f}_k$, $\mathbf{b}_k$
    Update $\mathcal{A}$ with $\mathbf{x}_k$ using the container's update function

Figure 3.3 Pseudocode of the adaptation on the original Iso+LineDD Emitter implementation through reduced batch sampling for future clustering integration in the selection process

By reducing the batch size of *B1*, we establish a foundation for integrating clustering techniques with better results in future work. In particular, with this first modification we expect that the integration of K-Means clustering will have a greater effect on the generated solutions in each iteration. Because the batch size of B1 is smaller, the same elites from B1 will be allowed to inherit traits from different clusters represented by the elites in B2. In essence, this approach aims to maintain diversity in the offspring generation, while focusing more on the promising regions in the solution space, as represented by the clusters. By combining these traits from the same B1 elites with different B2 elites, it is hoped that the offspring solutions will better represent the

characteristics of the clusters created by the clustering algorithm, which can potentially lead to a more efficient exploration of the search space and improved performance of the Iso+LineDD emitter. However, this approach is anticipated to yield worse results than the original IsoLineDD emitter in this stage as the clustering technique has not yet been introduced to the selection process.

In our implementation, we introduce the **get_rand_elites_for_reduced_batch** method to generate a batch of elite solutions from the archive by randomly selecting a specified number of elite solutions (*num_of_rand_elites*) and repeating each solution to fill the *batch_size*. The **ask** method incorporates changes in the selection process, while the **tell** method is updated to insert entries into the archive. This follows the same structure as the original implementation.

The full implementation of this modified emitter and the new method can be found in Appendix A.3. This modification serves as a preparatory step towards exploring more advanced techniques, such as clustering, to further enhance the performance of the Iso+LineDD emitter.

3.5.3. Enhancing Emitter Selection Process through K-Means Clustering Integration

In this section, we explore the integration of K-Means clustering into the selection process of the elites forming the second batch, B2. Building upon the modifications introduced in Section 3.5.2, which reduced the number of selected elites in batch B1, our primary goal is to refine the emitter's selection process for more effective exploration of promising regions in the solution space, resulting in diverse offspring solutions with high-fitness values.

An overview of the clustering-based adjustment to the selection procedure, utilizing the K-Means implementation on JAX (as discussed in Section 3.4), is provided in this section. Then, in subsections 3.5.3.1 and 3.5.3.2, respectively, detailed explanations of the two different strategies of this implementation are provided. It is important to emphasize that the only difference between these two versions lies in the selection process of the elites that form batch B2; all other aspects of the implementation still follow the same principles of the Iso+LineDD emitter.

Both versions employ unique strategies for perturbing the direction of the baseline elites selected in batch B1, while following the principles of the original Iso+LineDD emitter, including the variation process and formula. Contrary to the random selection of non-empty solutions from the archive, our approach groups the elites of the archive into clusters based on their similarity before the emitter's selection process starts. The number of clusters is determined by a predefined parameter. Once clustering is complete, the emitter selects a specific number of intra-cluster members from each cluster to form the batch B2.

The two versions of the implementation differ in their methods of choosing these cluster members: one selects elites closest to the cluster centers, while the other samples directly the centroids of the clusters. The rationale for both implementations is to guide the emitter's selection process towards regions in the solution space more likely to yield candidate solutions exhibiting higher fitness and diversity. By focusing on elites near the cluster centers, we expect these solutions to better represent the elites within each cluster, owing to their distinct features and representation of various regions of the solution space.

Subsection 3.5.3.1 provides an in-depth analysis of the first strategy that focuses on the elites closest to the cluster centers and perturbs them based on the Iso+LineDD variation process. Subsection 3.5.3.2 examines the implementation that uses centroids as the target for perturbation, also following the Iso+LineDD variation formula.

As discussed in Section 3.2, the clustering-based emitter seeks to enhance the exploration and exploitation capabilities of the QD optimization framework through the incorporation of clustering techniques in the selection process of the emitter. By implementing either strategy outlined in subsections 3.5.3.1 and 3.5.3.2, we anticipate improved algorithm performance metrics, leading to a more efficient approach for navigating complex problem domains and uncovering diverse, high-performing solutions.

3.5.3.1. Strategy 1: Perturbation Towards Elites Closest to Cluster Centers

This strategy aims to enhance the Iso+LineDD emitter selection process by incorporating K-Means clustering, which guides the perturbation of baseline elites (from batch B1) towards the elites nearest to the cluster centers. This updated emitter is based on the modified Iso+LineDD emitter presented in Section 3.5.2, which maintains reduced batch sampling for B1 to preserve diversity while focusing on more promising regions of the solution space. The full implementation of Strategy 1 can be found in Appendix A.4.

The high-level pseudocode of the modified emitter offers a clear illustration of the differences mentioned in this section, between this version and the previous one:

Algorithm 4 Modified Iso+LineDD Emitter (K-Means Clustering + Closest elites to each center)

Require: archive $\mathcal{A}$, iterations T, number of elites for variation generation N_e, offspring count N_{off}, number of clusters N_c, isotropic variance σ_1, directional variance σ_2
Ensure: Updated archive $\mathcal{A}$

Initialize the archive $\mathcal{A}$ with random solutions
for $t = 1$ to T **do**
    Sample a batch of N_e elites uniformly from $\mathcal{A}$ to form $\mathcal{B}_1$
    Call GetClosestElitesKmeans() to form batch $\mathcal{B}_2$ with N_{off} closest elites to the selected cluster centers
    for $k = 1$ to N_{off} **do**
        Select elites $\mathbf{x}_i \in \mathcal{B}_1$ and $mathbfx_j \in \mathcal{B}_2$
        if $N_e < N_{off}$ **then**
            Use elites from $\mathcal{B}_1$ multiple times
        Generate new offspring $\mathbf{x}_k$ by perturbing $\mathbf{x}_i$ according to the formula:

$$\mathbf{x}_k = \mathbf{x}_i + \sigma_1 \cdot \mathcal{N}(0, I) + \sigma_2 \cdot (\mathbf{x}_j - \mathbf{x}_i) \cdot \mathcal{N}(0, 1)$$

        Evaluate fitness and behavioral descriptor of $\mathbf{x}_k$: $\mathbf{f}_k$, $\mathbf{b}_k$
        Update $\mathcal{A}$ with $\mathbf{x}_k$ using the container's update function

Figure 3.4 Pseudocode of the adaptation on the modified Iso+LineDD Emitter implementation mentioned in section 3.5.2 with K-Means clustering integration in the selection process and the usage of the elites closer to the cluster centers

The main objective of integrating K-Means clustering into the selection process is to identify promising regions in the solution space, thus generating diverse offspring solutions with high-fitness values. To achieve this, we apply K-Means clustering to the non-empty solutions in the archive before choosing elites for batch *B2*. This step involves clustering the elites based on their similarities, with the number of clusters determined by a predefined parameter. Once the clustering process is complete, we select a specific number of intra-cluster members from each cluster to form batch *B2*, by targeting the elites that are nearest to the cluster centers.

By choosing elites close to the cluster centers, we expect these solutions to better represent the elites within each cluster due to their distinct features and coverage of various regions in the solution space. Consequently, this strategy aims to guide the emitter's selection process towards regions with a higher likelihood of producing candidate solutions exhibiting increased fitness and diversity.

To implement the first strategy, we modified the Iso+LineDD emitter and developed a new method to select the elites closest to each cluster center, forming batch *B2*. One of the main changes is the usage of **EmitterState** class to store and initialize cluster centers for the K-Means clustering algorithm. The **EmitterState** is a data class that holds information about the emitter's state, which should not be modified by the user. In this case, it stores the cluster centers from the

previous iterations. The **EmitterState** is initialized with an array of zeros for the cluster centers, which will be updated during the ask method.

The initialization of the required parameters, such as *archive, x0, iso_sigma, line_sigma, batch_size, num_of_clusters, and num_elites_to_repeat,* takes place in the **IsoLineKmeansVariationEmitter** class. In addition to this, the **EmitterState** class is also initialized through the **init** method, which assigns to the variable of the class an initial array consisting of only zeroes to serve as the cluster centers. The verification of the presence of exclusively zero values in the centers array of the EmitterState class is performed within the **ask** method of the emitter. This check is crucial to determine whether the K-Means clustering instance's centers in the present iteration need to be randomly initialized or if they should be assigned to the values of the final centers from the previous iteration.

The modified Iso+LineDD emitter now includes an additional step in the **ask** function that tries to enhance the selection process by performing K-Means clustering and selecting the nearest elites to the cluster centers to form batch B2. This crucial step is executed by invoking the **get_closest_elites_kmeans** method in the ask method. This method returns the selected elites for batch *B2* which are subsequently employed to guide the perturbation of the baseline elites from batch B1. After obtaining the selected elites and the new cluster centers from the **_archive.get_closest_elites_kmeans** method, the method updates the **EmitterState** with the new centers. The updated **EmitterState** is then used in the next iteration of the ask method to guide the K-Means clustering process faster.

Algorithm 5 Get the Closest Elites to each Cluster Center using K-means

Procedure GetClosestElitesKmeans($\mathcal{A}, n_E, n_C$)
Input: archive $\mathcal{A}$, batch size N_e, number of clusters N_c

Find indices of non-empty cells in $\mathcal{A}$
Get non-empty elites $\mathbf{E}$ from $\mathcal{A}$
Perform K-means clustering on $\mathbf{E}$, obtaining cluster centers $\mathbf{C}$ and assignments $\mathbf{P}$
Initialize an empty batch $\mathcal{B}$ that will contain all the elites that will be returned
if $N_e < N_c$ **then**
    Sample N_e clusters uniformly from $\mathbf{C}$
    for $c = 1$ to N_e **do**
        Call GetClosestPointsToCenter() to get the closest elite to the center of the selected cluster c and append it to $\mathcal{B}$
else
    Calculate the number of elites to be obtained per cluster n_{epc}:

$$n_{epc} = N_e \div N_c$$

    Calculate the remaining number of elites to be obtained per cluster n_r:

$$n_R = N_e \bmod N_c$$

    for $c = 1$ to N_c **do**
        Call GetClosestPointsToCenter() to get the closest n_{epc} elites to the center of cluster c
        Append the closest n_{EPC} elites to $\mathcal{B}$
    for $c = 1$ to n_r **do**
        Call GetClosestPointsToCenter() to get the next closest elite $n_{epc} + 1$ to the center of cluster c
        Append the $n_{epc} + 1$ closest elite to $\mathcal{B}$
return the batch $\mathcal{B}$ containing the selected elites

Figure 3.5 Pseudocode of the method that is utilized to return the closest elites to each cluster center from the archive to the ask function of the emitter

Regarding specifically the **get_closest_elites_kmeans** method that is implemented within the archive class of the library, it works by calling the static **_get_closest_elites_kmeans** method, which begins by reshaping the objective values in the archive state into a 2D fitness grid and identifying the indices of non-empty cells. K-Means clustering is subsequently applied to non-empty solutions of the archive, initializing the cluster centers randomly if *init_centers* is None or using the provided initial centers. Once the clustering process is complete, the method calculates the number of elites to select from each cluster based on the *batch size* and the *number of clusters*. It then chooses the nearest elites to each cluster center using the **closest_points_to_center** function from the K-Means implementation. In the final step, the method concatenates the selected elites to form batch *B2* and returns it, along with the cluster centers.

3.5.3.2. Strategy 2: Perturbation Towards Cluster Centroids

Strategy 2 focuses on perturbing solutions towards the direction of correlation of the centroids of clusters instead of selecting the closest elites from each cluster center, as in Strategy 1. The centroids are not necessarily actual solutions in the archive; however, they represent the average behavior within each cluster and can potentially guide the search in the behavior space more effectively. This section presents a detailed explanation of Strategy 2, its method, implementation, and how it differs from Strategy 1.

Algorithm 6 Obtain Cluster Centroids as Elites using K-means

Procedure GetCentroidElitesKmeans($\mathcal{A}, N_e, N_c$)
Input: archive $\mathcal{A}$, batch size N_e, number of clusters N_c

Find indices of non-empty cells in $\mathcal{A}$
Get non-empty elites $\mathbf{E}$ from $\mathcal{A}$
Perform K-means clustering on $\mathbf{E}$, obtaining cluster centers $\mathbf{C}$ and assignments $\mathbf{P}$
Initialize an empty batch $\mathcal{B}$ that will contain all the elites that will be returned
if $N_e < N_c$ **then**
    Sample N_e clusters uniformly from $\mathbf{C}$
    for $c = 1$ to N_e **do**
        Append the centroid of the selected cluster c to $\mathcal{B}$
else
    Calculate the number of times n_c to repeat each centroid obtained:

$$n_c = N_e \div N_c$$

    Calculate the remaining number of times n_R to repeat each centroid obtained:

$$n_r = N_e \bmod N_c$$

    for $c = 1$ to N_c **do**
        Append centroid of selected cluster c to $\mathcal{B}$, repeating it n_c times
    for $c = 1$ to n_r **do**
        Append centroid of selected cluster c to $\mathcal{B}$
return the batch $\mathcal{B}$ containing the selected elites

Figure 3.6 Pseudocode of the method that is utilized to return the centroid to each cluster center from the archive to the ask function of the emitter

The method used for Strategy 2 involves K-means clustering to obtain centroids as elites. The **GetCentroidElitesKmeans** procedure is responsible for generating a batch of elites *B2* consisting of centroids from selected clusters. In the implementation, the corresponding method **get_centroid_elites_kmeans** calls the static **_get_centroid_elites_kmeans**, which starts by finding the indices of non-empty cells in the archive. Similar to Strategy 1, it begins by reshaping the objective values in the archive state into a 2D fitness grid and identifying the indices of non-empty cells, extracting the corresponding elites. K-means clustering is then performed on these elites, providing cluster centers *C* and assignments *P*. An empty batch *B* is initialized to contain the selected elites.

Depending on the number of elites *Ne* and clusters *Nc*, the method determines how many times each centroid should be repeated in the final batch *B*, in order to fill the batch size. If the number of elites to be returned *Ne* is less than the number of clusters *Nc*, then the method uniformly samples *Ne* clusters from *C* and appends the centroid of each selected cluster to *B*. Otherwise, each centroid is repeated *nc* times, where *nc* is the result of dividing *Ne* by *Nc*. Then the remaining *nr* centroids are appended to batch *B*. Finally, the batch containing the selected elites is returned, along with the cluster centers.

The steps of the modified Iso+LineDD Emitter using Strategy 2 are listed in Figure 3.7. The main difference between Strategy 2 and Strategy 1 lies in the selection of elites for the second parent batch B2. Instead of selecting the closest elites to the cluster centers, we directly use the centroids themselves.

Algorithm 7 Modified Iso+LineDD Emitter (K-Means Clustering + Centroids as elites)

Require: archive $\mathcal{A}$, iterations T, number of elites for variation generation N_e, offspring count N_{off}, number of clusters N_c, isotropic variance σ_1, directional variance σ_2
Ensure: Updated archive $\mathcal{A}$

1: Initialize the archive $\mathcal{A}$ with random solutions
2: **for** $t = 1$ to T **do**
3: Sample a batch of N_e elites uniformly from $\mathcal{A}$ to form $\mathcal{B}_1$
4: Call GetCentroidElitesKmeans() to form batch $\mathcal{B}_2$ with N_{off} centroids of selected clusters as elites
5: **for** $k = 1$ to N_{off} **do**
6: Select elites $\mathbf{x}_i \in \mathcal{B}_1$ and $\mathbf{x}_j \in \mathcal{B}_2$
7: **if** $N_e < N_{off}$ **then**
8: Use elites from $\mathcal{B}_1$ multiple times
9: Generate new offspring $\mathbf{x}_k$ by perturbing $\mathbf{x}_i$ according to the formula:

$$\mathbf{x}_k = \mathbf{x}_i + \sigma_1 \cdot \mathcal{N}(0, I) + \sigma_2 \cdot (\mathbf{x}_j - \mathbf{x}_i) \cdot \mathcal{N}(0, 1)$$

10: Evaluate fitness and behavioral descriptor of $\mathbf{x}_k$: $\mathbf{f}_k$, $\mathbf{b}_k$
11: Update $\mathcal{A}$ with $\mathbf{x}_k$ using the container's update function

Figure 3.7 Pseudocode of the adaptation on the modified Iso+LineDD Emitter implementation mentioned in section 3.5.2 with K-Means clustering integration in the selection process and the usage of the centroids the clusters

In the modified Iso+LineDD Emitter, we first sample a batch of *Ne* elites uniformly from the archive to form the first parent batch *B1*. Next, we call *GetCentroidElitesKmeans* to obtain the centroids of the selected clusters. These centroids are used as the second parent batch *B2*. The rest of the procedure remains the same as in Strategy 1, where we generate offspring by perturbing the first parent according to the formula in line 9.

The full implementation of Strategy 2 can be found in Appendix A.5. Note that the implementation details may differ from the pseudocode for readability purposes.

In conclusion, Strategy 2 uses cluster centroids as the reference points for generating offspring solutions through perturbation, providing an alternative approach to Strategy 1, which selects the closest elites to the cluster centers.

3.6. Experimental Setup

This section outlines the experimental setup employed to assess the proposed strategies for improving emitter selection through the integration of K-Means clustering. We provide details on the problem domains chosen for the experiments, namely the Arm Repertoire and the Rastrigin Function. These problem domains are selected due to their complexity and diverse solution spaces, which allow for a comprehensive evaluation of the exploration and exploitation capabilities of the modified Iso+LineDD Emitter with the proposed clustering-based strategies. All experiments are conducted using MAP-Elites' Grid Archive.

3.6.1. Problem domains

Two distinct problem domains are selected for the experiments: the *Robotic Arm Repertoire* and the *Rastrigin Function*. Both domains were implemented in the QDJAX library, together with their respective objective and behavioral description functions and gradients. It is important to note that these objective functions determine the goals of optimization problems by quantifying the quality of solutions. Behavioral descriptor functions, on the other hand, describe the features of solutions, enabling comparisons and categorization based on their characteristics.

The implementations can be found in the **problem_definitions.py** and **qd_functions.py** files. For the experiments conducted in this thesis, their normalized version is used.

3.6.1.1. N-Dimensional Robotic Arm Repertoire

The first problem we address is the control of a simulated N-Degree-of-Freedom (N-DoF) robotic arm. The objective function of the problem aims to minimize the joint angles' variance, while the behavioral descriptor function represents the end-effector positions (x, y) of the robotic arm. The overall goal of this task is to discover how to access all reachable points by the arm while minimizing the variance between the different angles applied in each of its DoF. The optimal joint angles should be as close to each other as possible, resulting in a smooth curve for the arm's final position. We set N to be 2, 10, and 100 for the experiments.

The implementation of the problem can be found in the **get_norm_arm_info** function in **problem_definitions.py** and the corresponding functions **norm_arm_fit** and **norm_arm_beh** in **qd_functions.py**.

3.6.1.2. N-Dimensional Rastrigin Function

The second problem domain is the N-Dimensional Normalized Rastrigin Function, known for its numerous local minima. By using the Rastrigin Function, the experiment aims to tackle a general optimization problem with a multimodal landscape, posing a different set of challenges for the proposed algorithms. For the purposes of the experiments conducted, N is set to be 2, 10, and 100.

The implementation of the problem can be found in the **get_norm_rastrigin_info** function in **problem_definitions.py** and the corresponding functions **norm_rastrigin_fit** and **norm_rastrigin_beh** in **qd_functions.py**.

3.6.2. Experimental Scenarios

To assess the performance of the proposed strategies for enhancing emitter selection through the integration of K-Means clustering, we designed several experimental scenarios. These scenarios encompass various configurations and combinations of the two problem domains, the N-Dimensional Robotic Arm Repertoire and the N-Dimensional Rastrigin Function, and the two strategies, Perturbation Towards Elites Closest to Cluster Centers (Strategy 1) and Perturbation Towards Cluster Centroids (Strategy 2), as well as the modified version without clustering that was used as an intermediate step towards our final goal.

The following hyperparameters were used for all scenarios:

- Isotropic variance $\sigma 1$: 0.02
- Directional variance $\sigma 2$: 0.2
- Batch size for batch B2: 2^7
- Number of clusters: 2^2, 2^4, 2^6,2^8
- Repetitions: 10
- Evaluations: 10^6

The scenarios 1 and 2 are organized into two primary categories, distinguished by grid size: Scenarios featuring a 100x100 grid size, and those utilizing a 400x400 grid size. These scenarios address 2D problem domains, with a focus on assessing the impact of changing the number of clusters and elites for each set of elites in batch B1. Within each category, the performance of the emitters are evaluated based on exploration and exploitation capabilities, allowing for comparisons across diverse configurations of problem domains, strategies, and hyperparameters.

3.6.2.1. Scenarios 1 - 2 with 100x100 Grid Size for 2-D problem domains

For each of the following scenarios, the respective problem is addressed using the different emitters and a grid size of 100x100, while varying the number of elites in batch B1 and examining the impact of the number of clusters on emitter performance.

Scenario 1A: 2-D Robotic Arm with 2^2 Elites

Scenario 1B: 2-D Robotic Arm with 2^4 Elites

Scenario 1C: 2-D Robotic Arm with 2^6 Elites

Scenario 2A: 2-D Rastrigin Function with 2^2 Elites

Scenario 2B: 2-D Rastrigin Function with 2^4 Elites

Scenario 2C: 2-D Rastrigin Function with 2^6 Elites

3.6.2.2. Scenarios 3 - 4 with 400x400 Grid Size for 2-D problem domains

For each of the scenarios that follow, the problem is handled utilizing different emitters and a grid size of 400x400, while modifying the number of elites in batch B1 and investigating the impact of cluster size on emitter performance.

Scenario 3A: 2-D Robotic Arm with 2^2 Elites

Scenario 3B: 2-D Robotic Arm with 2^4 Elites

Scenario 3C: 2-D Robotic Arm with 2^6 Elites

Scenario 4A: 2-D Rastrigin Function with 2^2 Elites

Scenario 4B: 2-D Rastrigin Function with 2^4 Elites

Scenario 4C: 2-D Rastrigin Function with 2^6 Elites

3.6.2.3. Additional Scenarios: Higher-Dimensional Problems

To further evaluate the performance of the proposed strategies, we apply the different emitters to higher-dimensional problems. The N-Dimensional Robotic Arm and N-Dimensional Rastrigin Function problems are addressed with N set to 10 and 100. These scenarios provide insights into the scalability of the strategies and their ability to handle more complex problem domains.

In each of these higher-dimensional scenarios, all four emitters (Iso+LineDD, Modified Iso+LineDD without K-Means Clustering, Strategy 1, and Strategy 2) are compared, with a focus on the impact of changing the number of clusters and elites in higher-dimensional problems. The performance of the emitters are assessed in terms of exploration and exploitation capabilities, enabling us to compare the results across different configurations and hyperparameter settings in higher-dimensional problem domains. Note that the experiments in this section are performed on MAP-Elites Grid Archive with dimensions 100x100. Initially, our objective was to also test the emitters with a larger grid size 400x400 as in 2-D problem domains. However, these experiments

require significant time to execute, and running them for multiple repetitions will exceed the deadline of this paper.

By investigating a wide range of experimental scenarios with different problem domains, strategies, and hyperparameter settings, we aim to provide a comprehensive evaluation of the proposed strategies for enhancing emitter selection through the integration of K-Means clustering with the MAP-Elites Grid Archive.

For each of the following scenarios, the respective problem is addressed using the different emitters and varying the number of elites in batch B1, examining the impact of the number of clusters on emitter performance in higher-dimensional problem domains.

Scenarios 5 – 6 with 10-Dimensional Problems:

Scenario 5A: 10-D Robotic Arm with 2^2 Elites

Scenario 5B: 10-D Robotic Arm with 2^4 Elites

Scenario 5C: 10-D Robotic Arm with 2^6 Elites

Scenario 6A: 10-D Rastrigin Function with 2^2 Elites

Scenario 6B: 10-D Rastrigin Function with 2^4 Elites

Scenario 6C: 10-D Rastrigin Function with 2^6 Elites

Scenarios 7 – 8 with 100-Dimensional Problems:

Scenario 7A: 100-D Robotic Arm with 2^2 Elites

Scenario 7B: 100-D Robotic Arm with 2^4 Elites

Scenario 7C: 100-D Robotic Arm with 2^6 Elites

Scenario 8A: 100-D Rastrigin Function with 2^2 Elites

Scenario 8B: 100-D Rastrigin Function with 2^4 Elites

Scenario 8C: 100-D Rastrigin Function with 2^6 Elites

3.6.3. Key Metrics for Evaluating Performance

To evaluate and compare the performance of the proposed strategies, we use several performance metrics, including:

1. Coverage: The percentage of cells in the behavior space that have been explored, giving a measure of the exploration capability of the algorithm [37].
2. Max fitness: The highest fitness value found in any cell of the archive, indicating the best solution found by the algorithm [37].
3. Mean fitness: The average fitness value of the solutions in the archive, providing an indication of the algorithm's exploitation capability.

4. QD score: A quality diversity measure that sums the finesses of all solutions in the archive, taking into consideration both the quality and the diversity of the algorithm [37]. Two algorithms may produce archives with equal QD Scores, but one may have different Coverage or Max Fitness score from the other [37].

In the plots, the color of the emitter lines is determined by the performance of the QD score. The emitter with the highest score is assigned the color blue, the second highest gets orange, the third highest is given green, and the fourth highest is represented by red. To visualize the metrics, each plot provides an overview of the emitter's performance across all repetitions. Instead of showing separate lines for each repetition, the plot processes the data across all repetitions and displays the aggregated results for each emitter. In particular, the **median**, **q25** (25th percentile), and **q75** (75th percentile) values are calculated for each metric and each emitter by considering all the repetitions. With these values, each plot is able to show the central tendency as well as the spread of the performance data for each emitter, while also taking into account all the repetitions.

By employing these performance metrics, we can obtain a comprehensive evaluation of the proposed strategies' effectiveness in enhancing emitter selection through the integration of K-Means clustering with the MAP-Elites Grid Archive.

Chapter 4
Experimental Results and Discussion

4.1. INTRODUCTION 49
4.2. RESULTS FOR 2-DIMENSIONAL PROBLEM DOMAINS 49
4.3. RESULTS FOR HIGHER-DIMENSIONAL PROBLEM DOMAINS 78
4.4. MAIN FINDINGS 105

4.1. Introduction

This chapter is structured to first present the results for 2-dimensional problem domains in Section 4.2, specifically addressing Scenarios 1 - 4 with grid sizes of 100x100 and 400x400. Following that, Section 4.3 discusses the results obtained for higher-dimensional problem domains, covering Scenarios 5 - 8, which involve 10-dimensional and 100-dimensional problems. Finally, Section 4.4 concludes the chapter by listing the main findings of our research, summarizing everything that we have learned.

4.2. Results for 2-Dimensional Problem Domains

In this section, we present the experimental results for 2-dimensional problem domains, specifically addressing Scenarios 1 - 4 with grid sizes of 100x100 and 400x400. The performance of the proposed strategies, Perturbation Towards Elites Closest to Cluster Centers (Strategy 1) and Perturbation Towards Cluster Centroids (Strategy 2), along with the Modified Iso+LineDD without K-Means Clustering, is analyzed and compared against the baseline Iso+LineDD emitter. The key performance metrics used for the evaluation include Coverage, Max Fitness, Mean Fitness, and QD Score. As mentioned in the Methodology chapter, the emitter with the highest QD score in each experiment is assigned the color blue, the second highest gets orange, the third highest is given green, and the fourth highest is represented by red.

4.2.1. Scenario 1 with 100x100 Grid Size for 2-D Robotic Arm Problem

The first scenario, Scenario 1, refers to a two-dimensional Arm problem. The analysis focuses on a problem domain that is characterized by a grid size of 100x100.

Beginning with Scenario 1A, it was shown that **when utilizing a small number of clusters with the K-Means clustering algorithm, all metrics amongst each emitter recorded relatively similar performance**. One possible explanation for this is that the division of the solution space into sub regions is limited when the number of clusters is smaller. Consequently, **the performance of the four emitters matches up in terms of coverage, fitness, and QD score**. It is possible that the benefits of K-Means clustering, and the proposed strategies are not yet pronounced enough to have any substantial impact.

As the number of clusters increases, the strategies divide the solution space into finer sub regions. **In this case, the graphs show that the coverage score is lower for the proposed strategies when compared to the baseline Iso+LineDD emitter.** This suggests that the proposed strategies could potentially get "stuck" in local optima, limiting their ability to explore other regions of the solution space. In contrast, the baseline Iso+LineDD emitter may maintain a more balanced exploration-exploitation trade-off as the coverage is significantly higher. This allows the emitter to accomplish substantially greater coverage across the solution space.

Regarding the mean fitness metrics observed in this scenario, **a progressive increase in the score has been observed across all the emitters**, indicating that they are increasing the quality of their solutions over time. However, **for larger numbers of clusters, a significant delay was observed for the two proposed strategies**, which can again be attributed to the emitters focusing on local optima or spending additional time optimizing their search within the clusters.

When it comes to the maximum fitness metric however, regardless of the number of clusters, all emitters achieve the same maximum fitness. This demonstrates that the proposed strategies are still capable of identifying optimal solutions in the search space, despite the differences shown in other metrics such as coverage and mean fitness.

Last but not least, **as the number of clusters increased, the QD score of the proposed strategies decreased**, indicating a decline in the algorithm's overall quality and diversity and verifying the results of the other metrics.

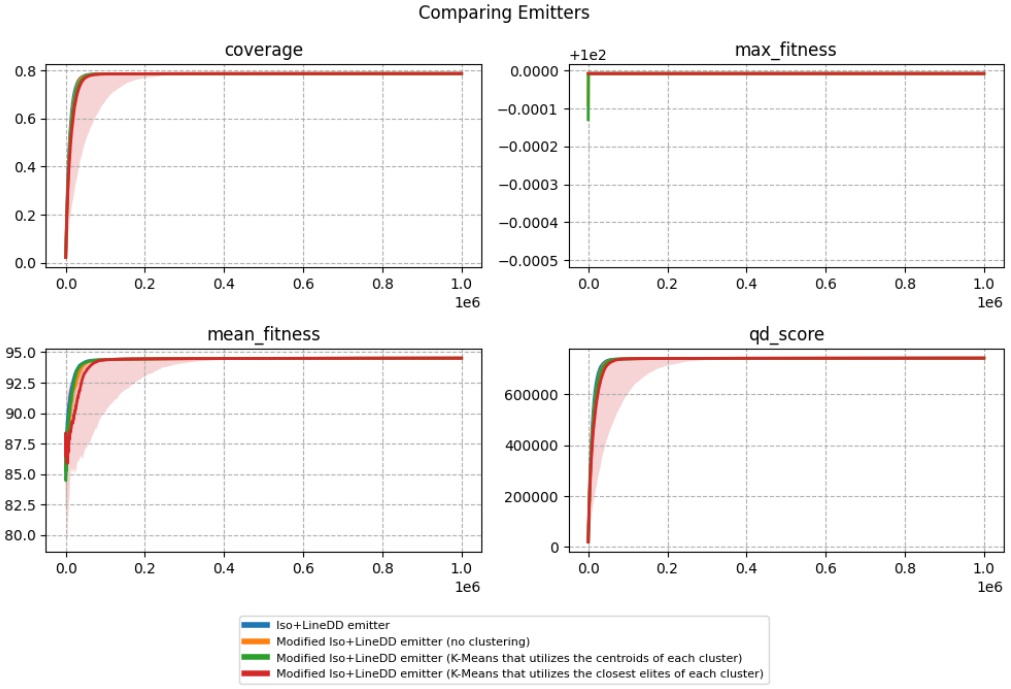
Moving on, the comparison between Scenario 1A results with those from Scenarios 1B and 1C indicates that the **emitters' performance remains consistent regardless of how many elites are sampled in batch size of B1**. This suggests that for the case of this problem domain, the

search space may be relatively simple, allowing the emitters to achieve comparable performance regardless of the batch size.

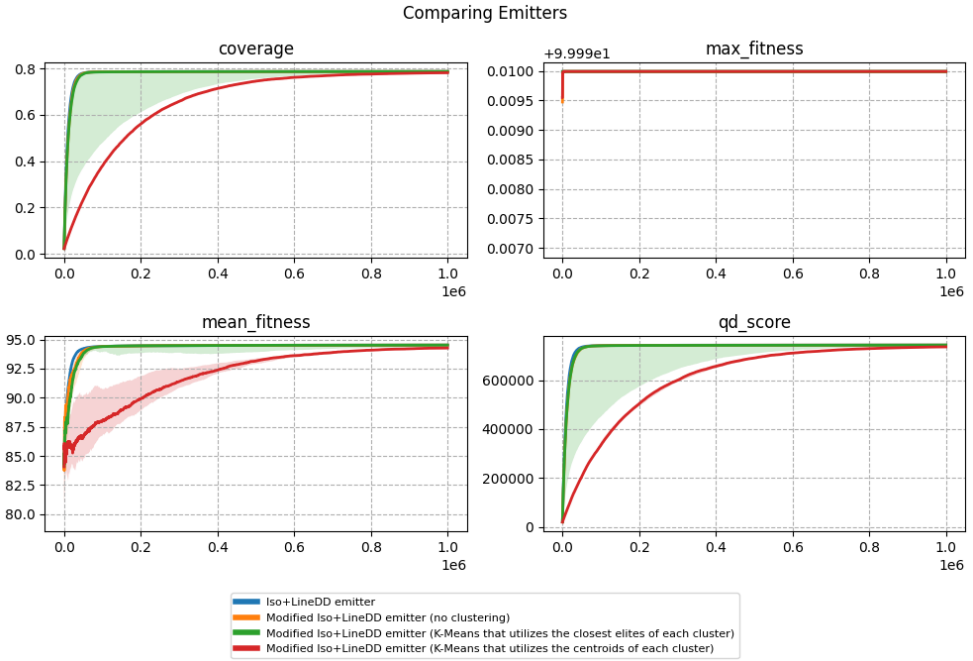
As for the overall comparison between the two proposed strategies, they both performed in a very similar way. The occasional faster convergence to the best score by one of the two emitters could be attributed to a lucky initialization or a more effective exploration of the search space in that specific set of runs.

Scenario 1A: 2-D Robotic Arm with 2^2 Elites

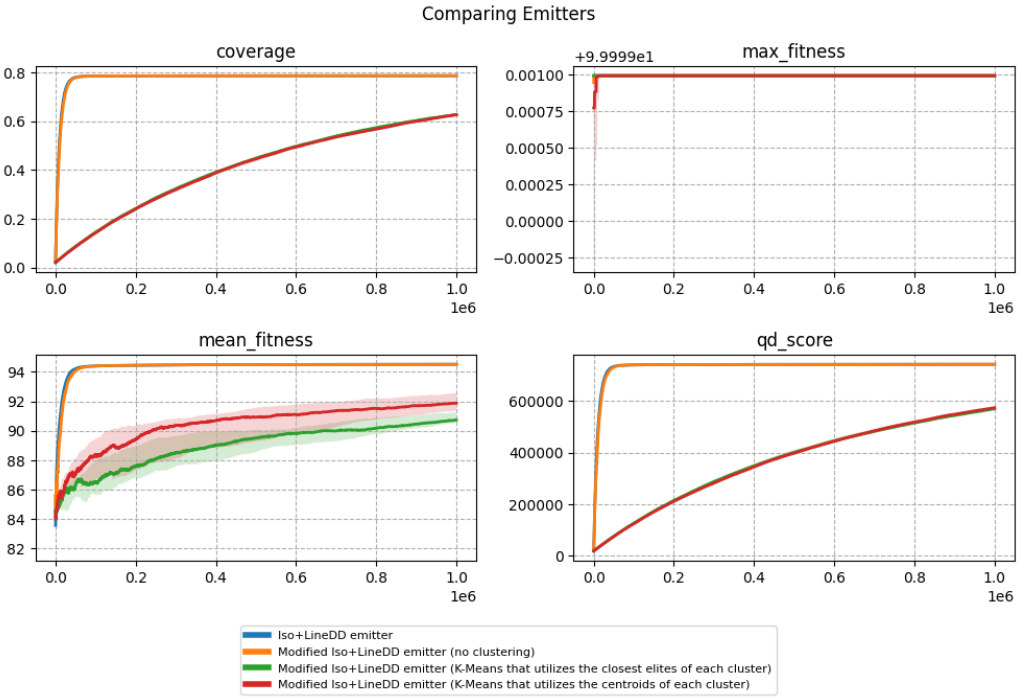
Number of clusters: 2^2



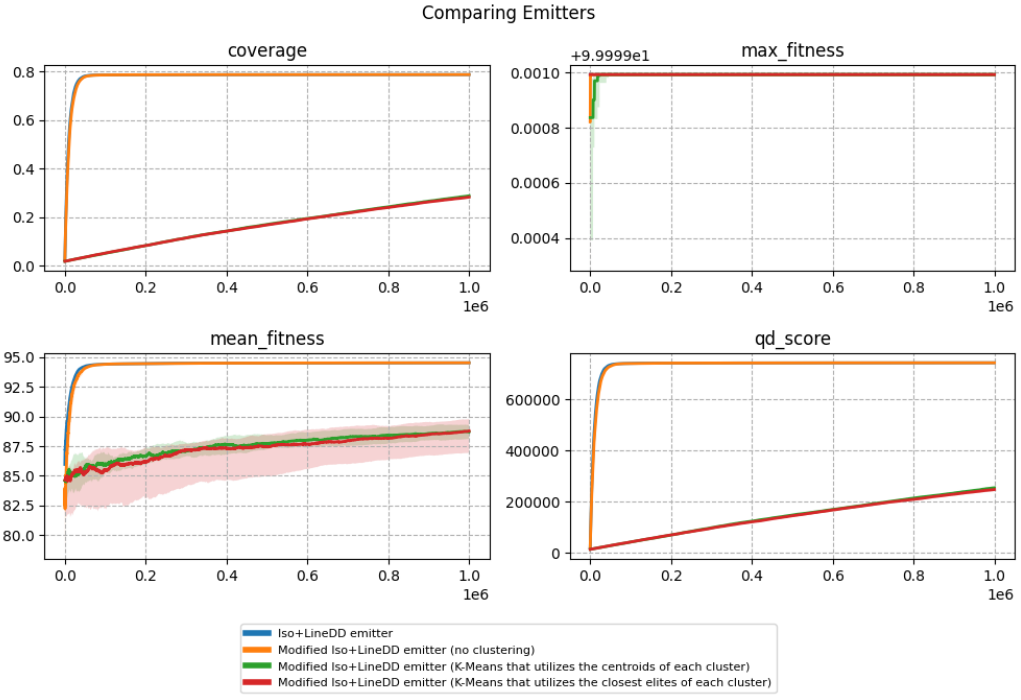
Number of clusters: 2^4



Number of clusters: 2^6

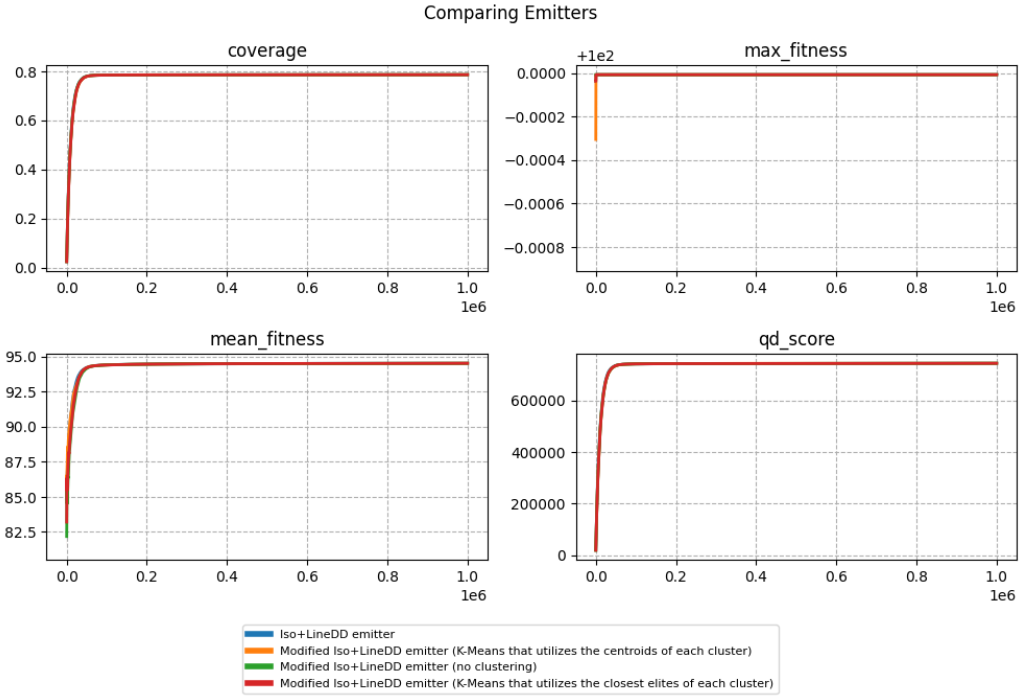


Number of clusters: 2^8

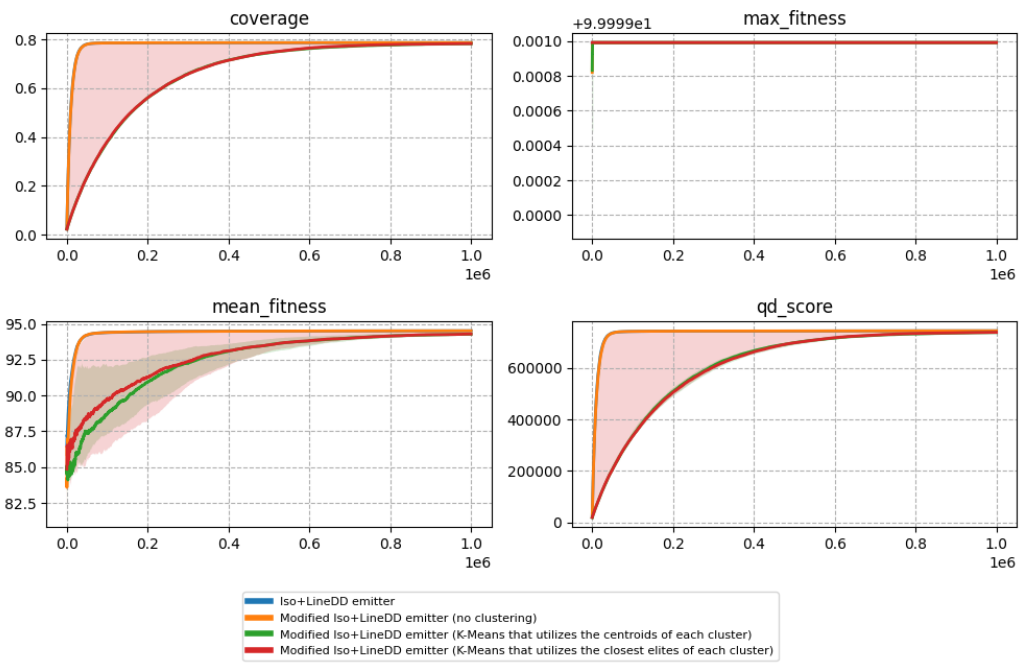


Scenario 1B: 2-D Robotic Arm with 2^4 Elites

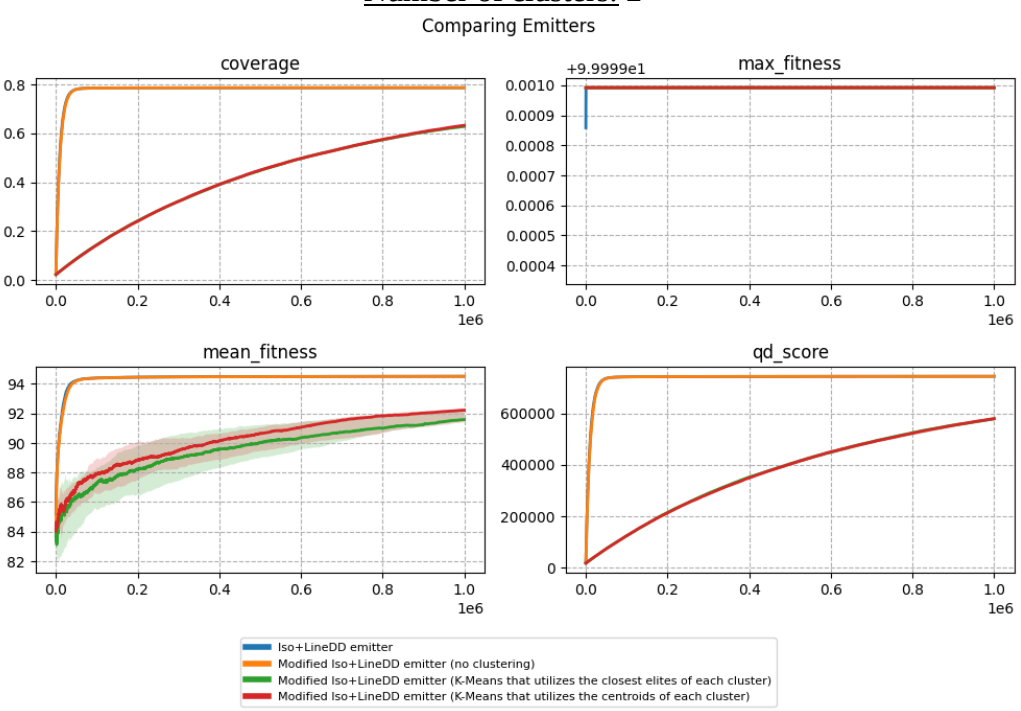
Number of clusters: 2^2



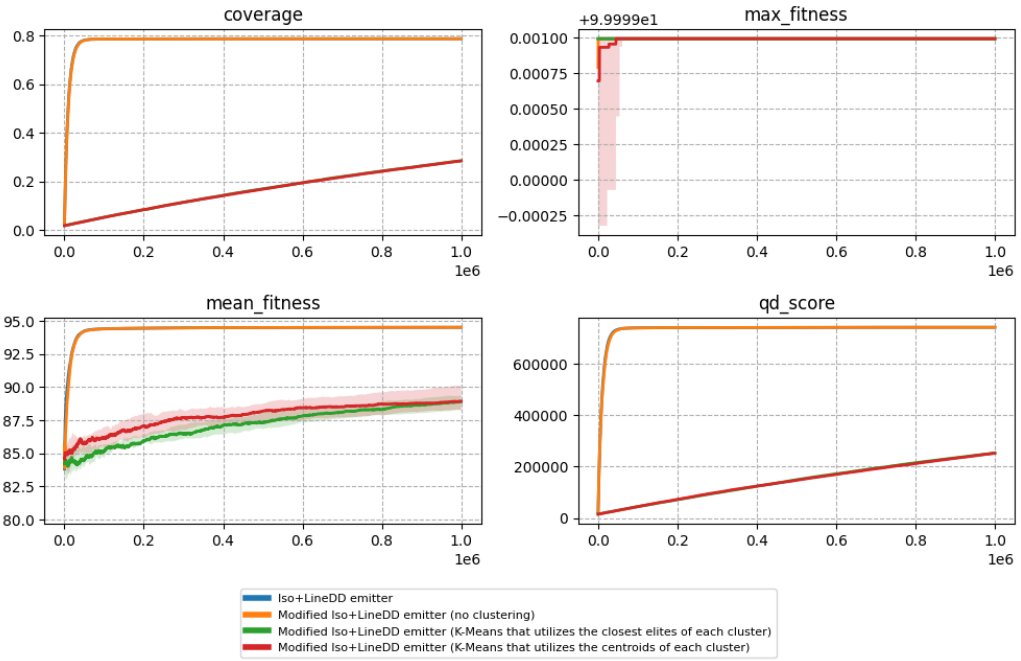
Number of clusters: 2^4



Number of clusters: 2^6

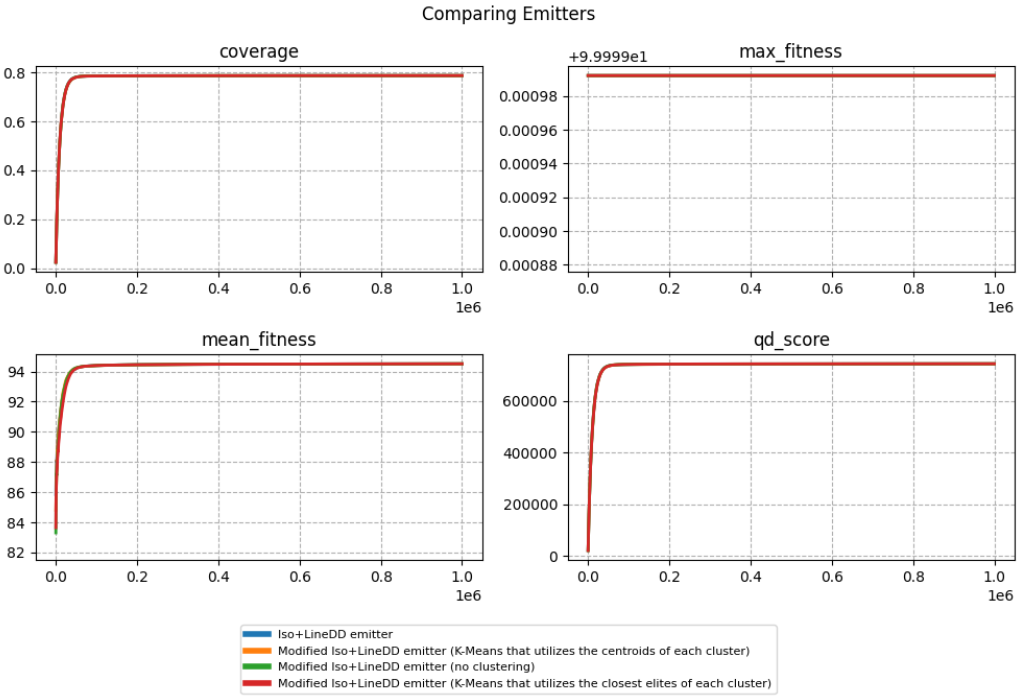


Number of clusters: 2^8

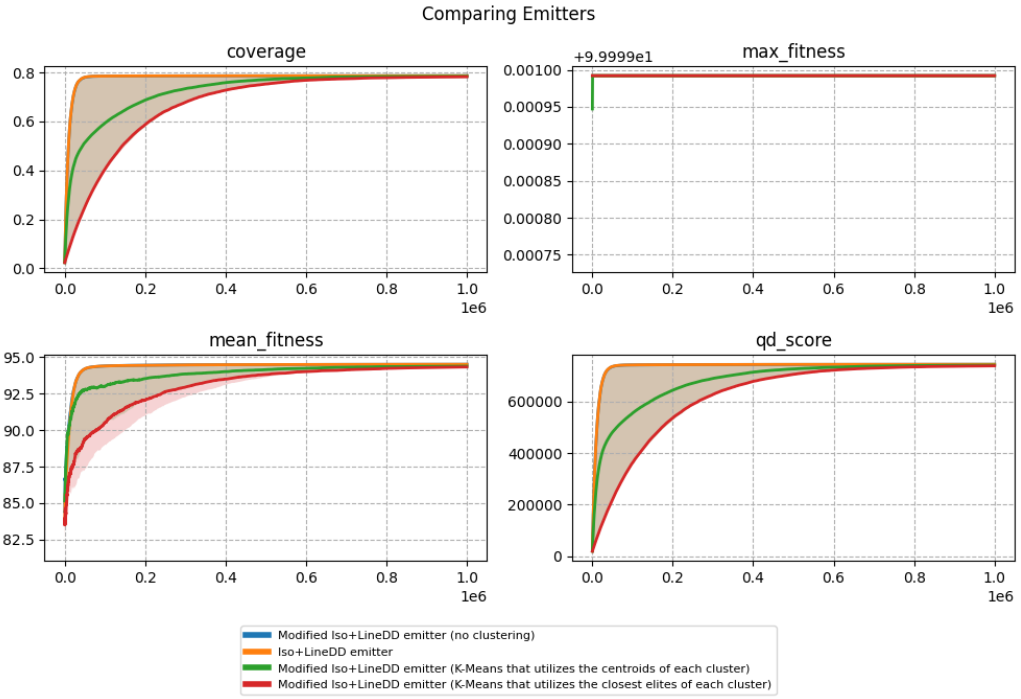


Scenario 1C: 2-D Robotic Arm with 2^6 Elites

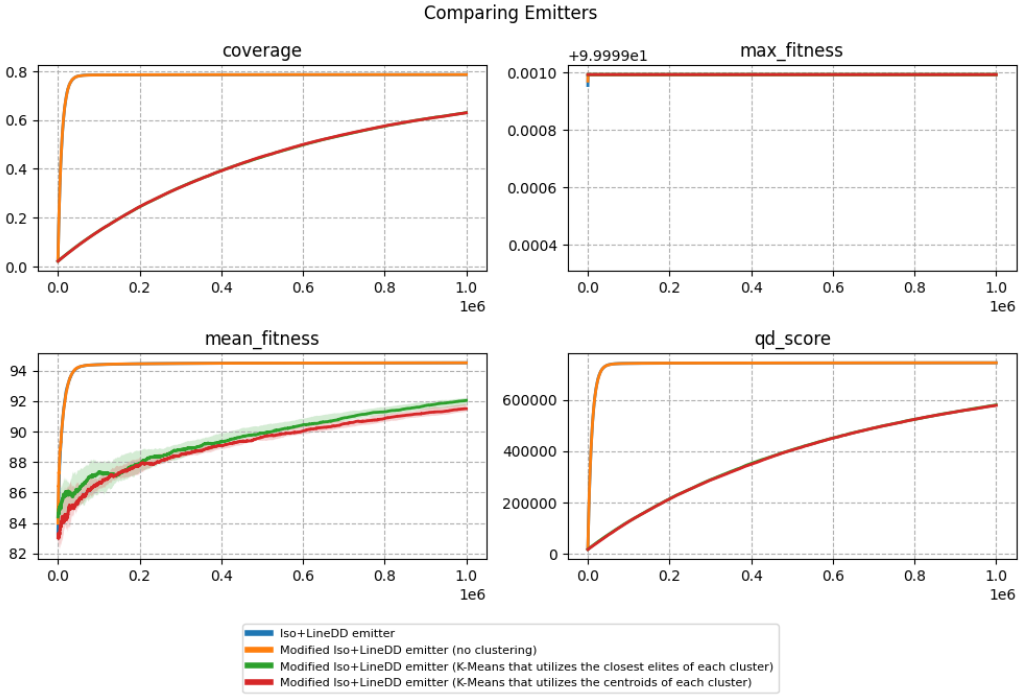
Number of clusters: 2^2



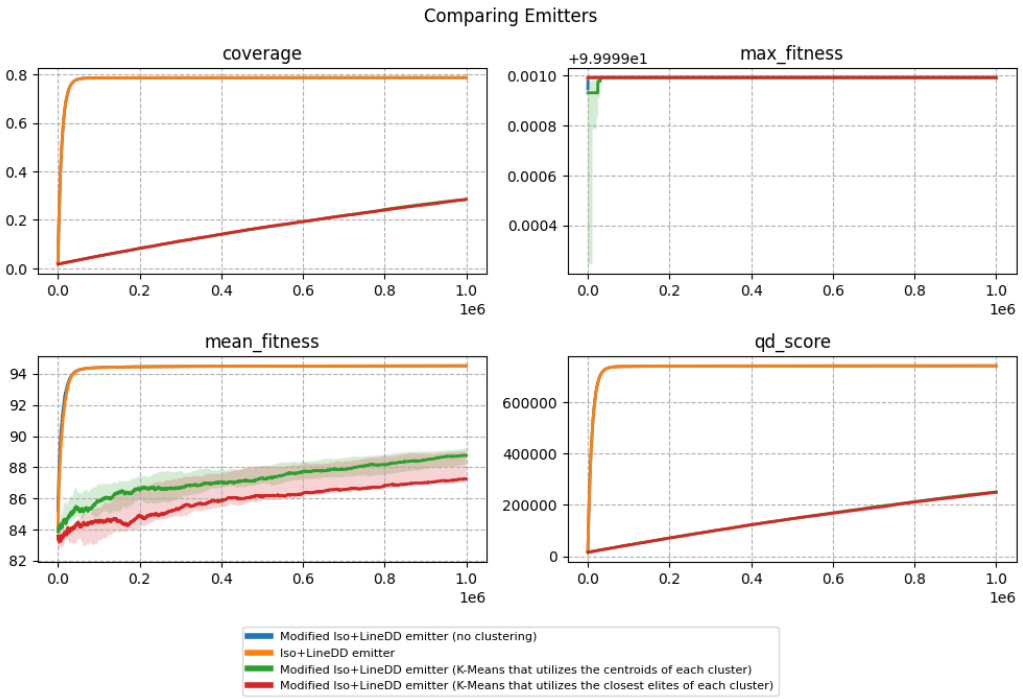
Number of clusters: 2^4



Number of clusters: 2^6



Number of clusters: 2^8



4.2.2. Scenario 2 with 100x100 Grid Size for 2-D Rastrigin Function

The second scenario, referred to as Scenario 2, involves a two-dimensional Rastrigin problem. The analysis concentrates on a problem domain characterized by a 100x100 grid size.

Similar to Scenario 1, this scenario's results indicate that the proposed emitters struggle to match the efficacy of the Iso+LineDD emitter.

In Scenario 2A, when the number of clusters was set to 4 the performance across all metrics was analogous to that of the baseline emitter. This suggests, as before, that the benefits of K-Means and the proposed strategies cannot have a significant impact when a limited number of clusters is used.

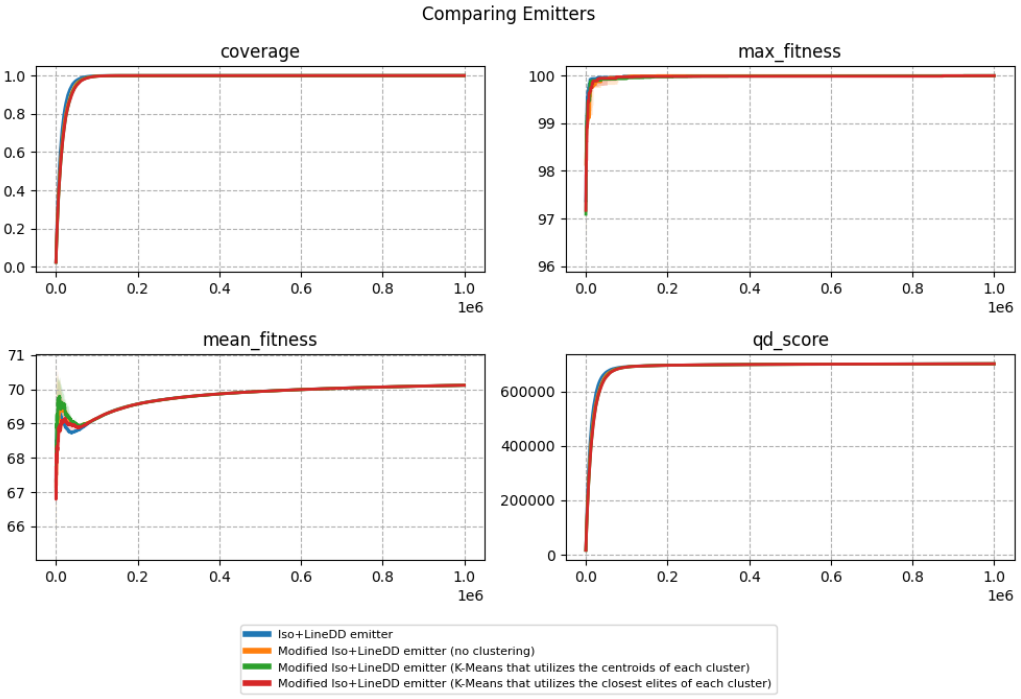
Moving to the assignment of 16 clusters in the same scenario, the proposed emitters were able to achieve metrics that were similar to the baseline emitter yet again. However, **the convergence to the highest scores was delayed for both strategies, with Strategy 2 converging slightly faster**. In particular, the mean fitness of the proposed strategies follows the same trend as the original emitter initially, but this trend ultimately declines after a period of time, resulting in decreased mean fitness values. In this regard, **Strategy 2 still outperforms Strategy 1.**

As the number of clusters produced by the K-Means algorithm increases, the performance of the proposed strategies deteriorates even further. As seen in the plots, the coverage is substantially lower, falling short of achieving the same maximal fitness as Iso+LineDD. Likewise, for the QD result. As far as the mean fitness scores are concerned, the proprosed emitters exhibit similar behavior to the assignment of 16 clusters.

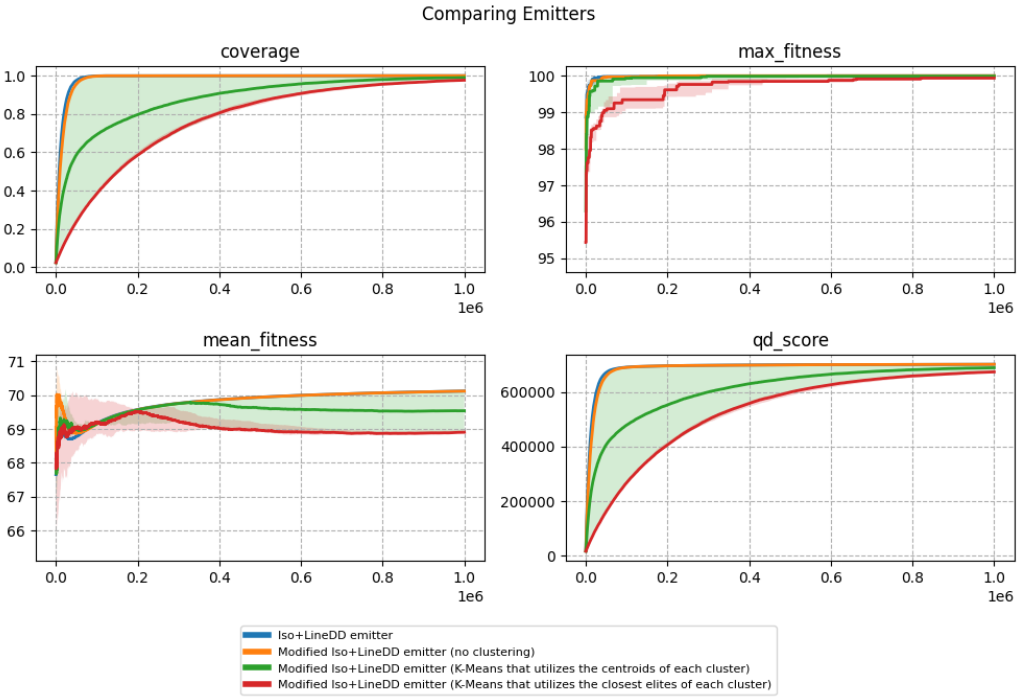
When comparing Scenario 2A to both Scenario 2B and 2C, with batch sizes of 16 and 64 elites in B1 respectively, it was shown that the proposed emitters exhibit the same behavior as in Scenario 2A. However, there was one exception where this was not the case. To be more specific, in a particular experiment in scenario 2B in which there are 16 clusters, the emitter employing Strategy 2 performs similarly to the emitter employing Iso+LineDD across all metrics, and significantly outperforms the emitter employing Strategy 1.

Scenario 2A: 2-D Rastrigin Function with 2^2 Elites

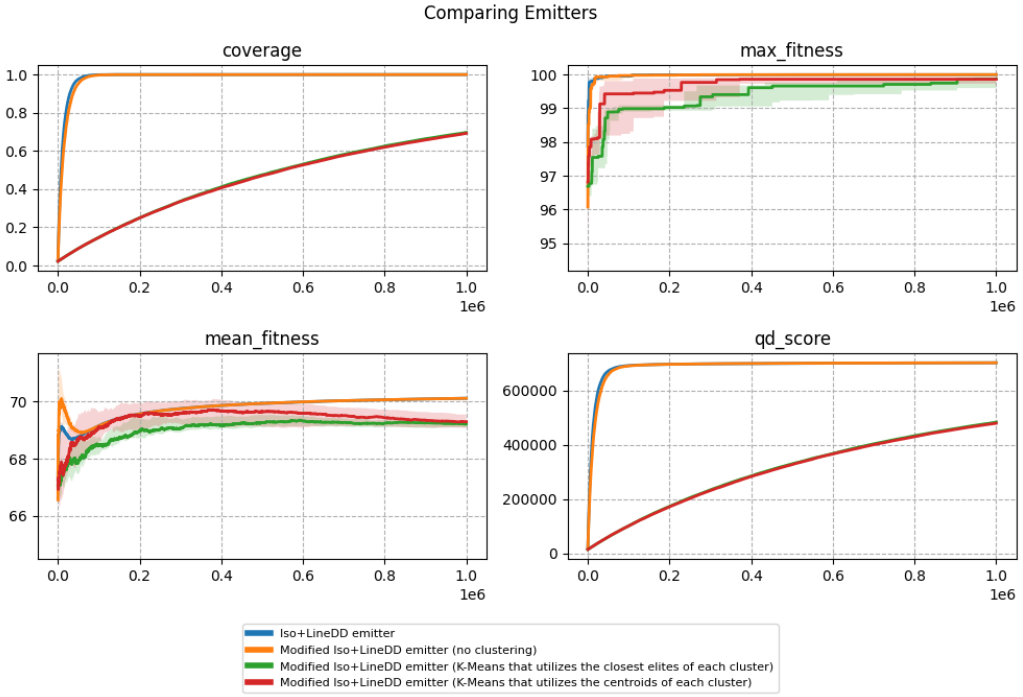
Number of clusters: 2^2



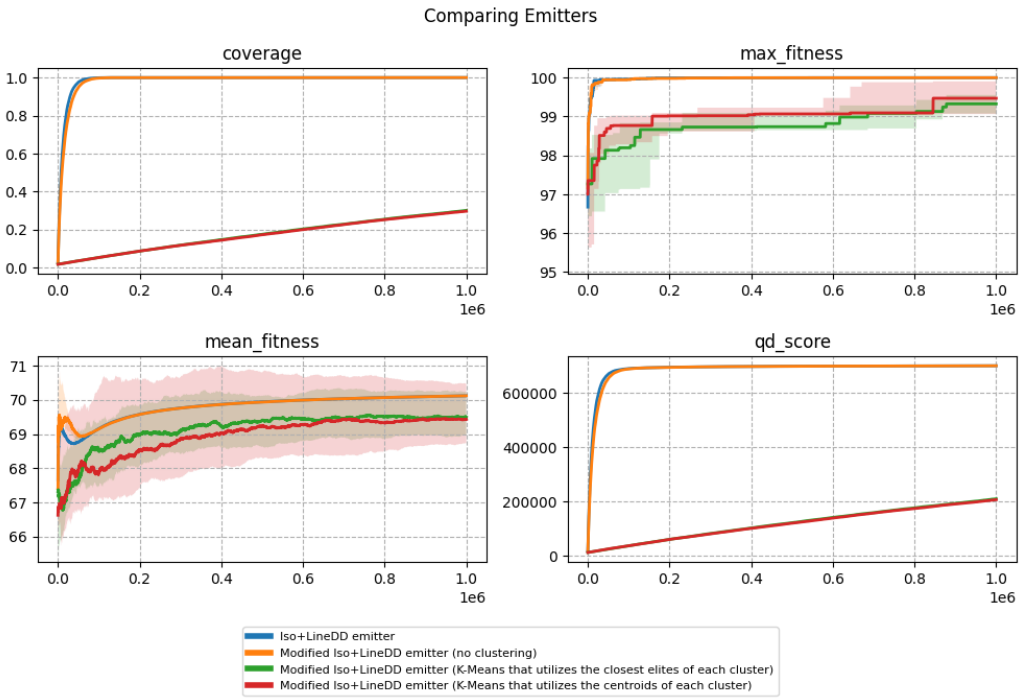
Number of clusters: 2^4



Number of clusters: 2^6

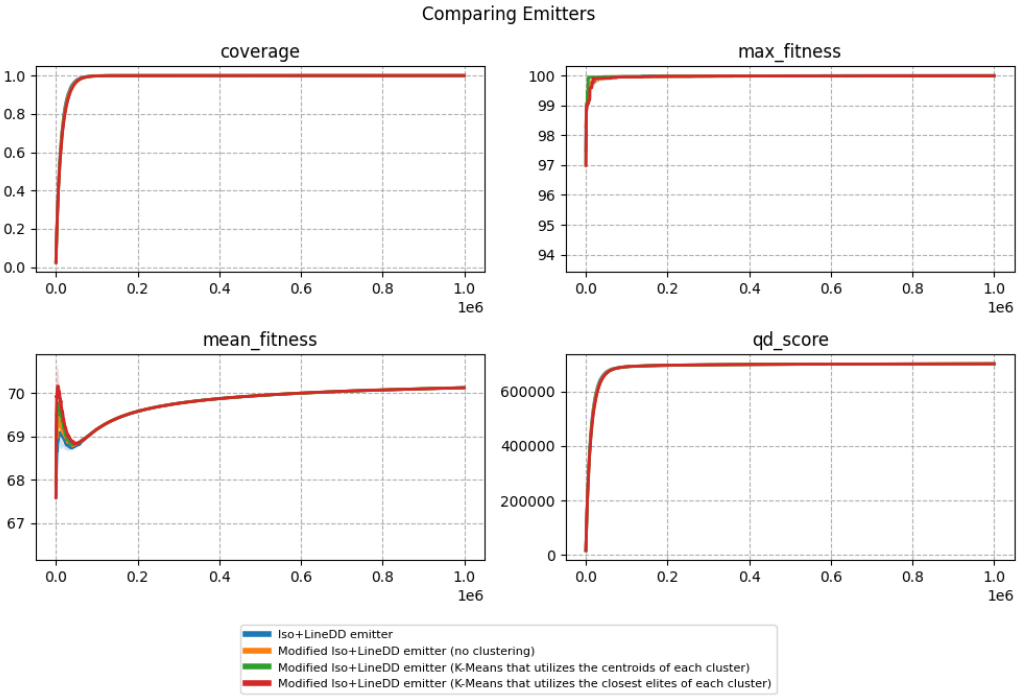


Number of clusters: 2^8

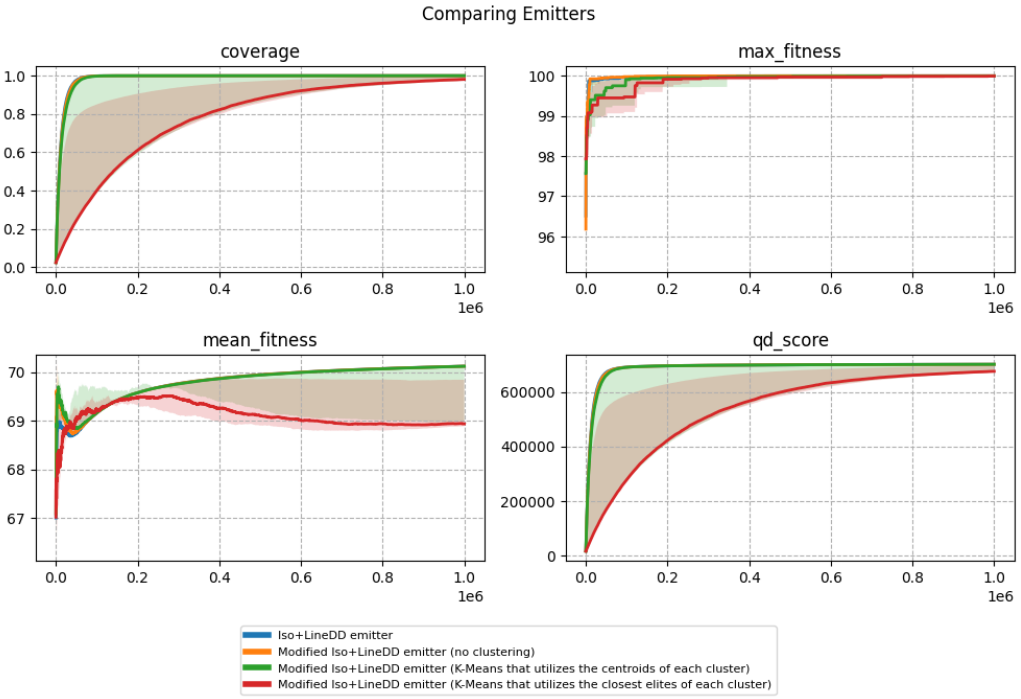


Scenario 2B: 2-D Rastrigin Function with 2^4 Elites

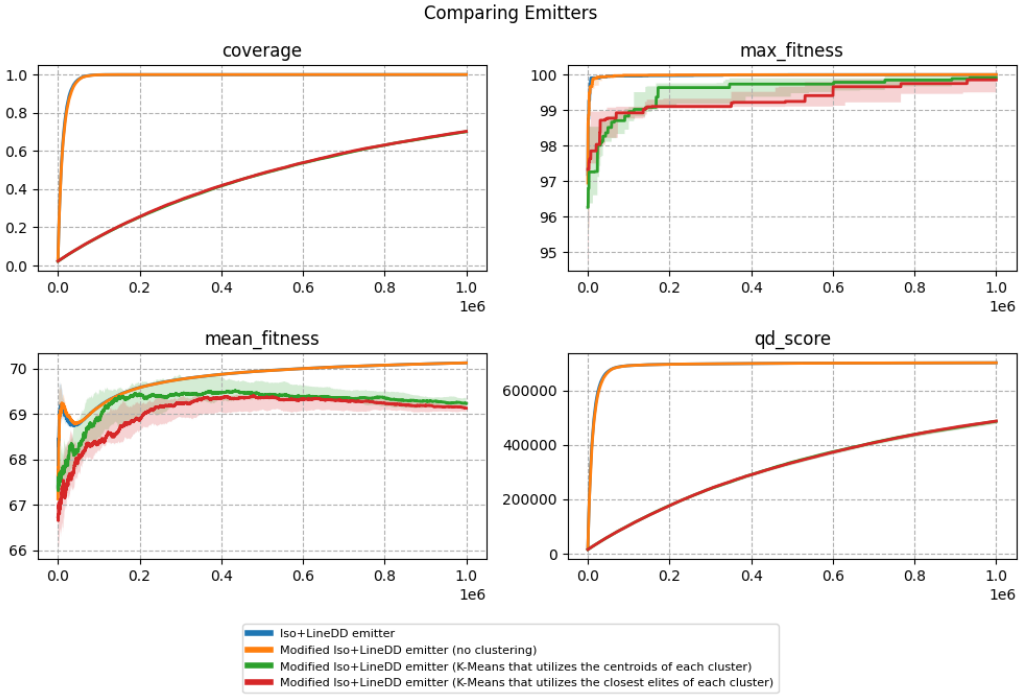
Number of clusters: 2^2



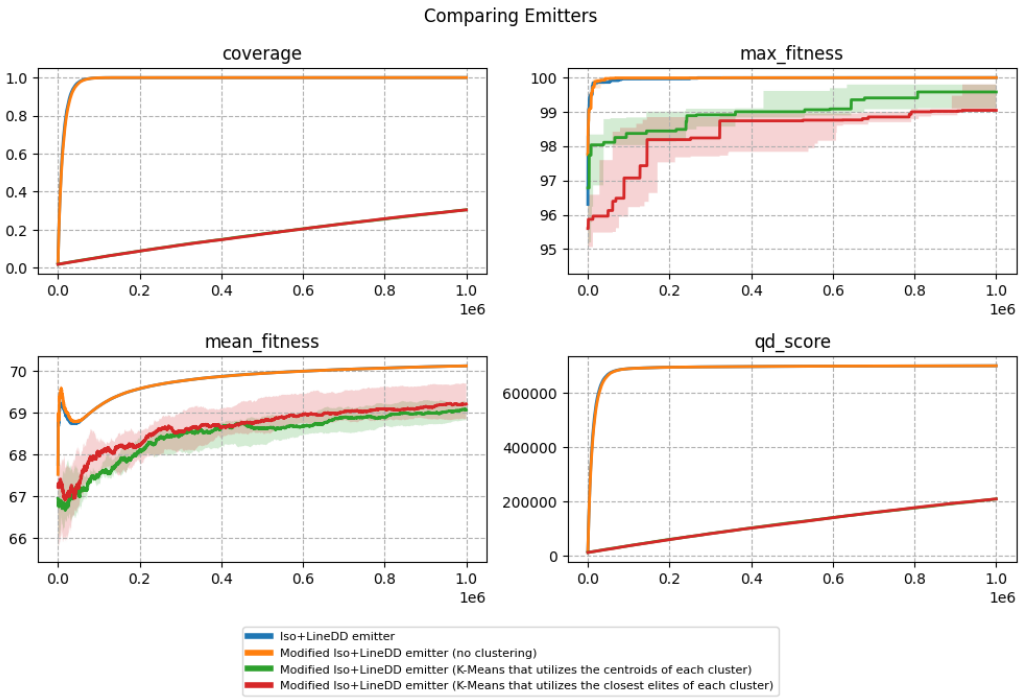
Number of clusters: 2^4



Number of clusters: 2^6

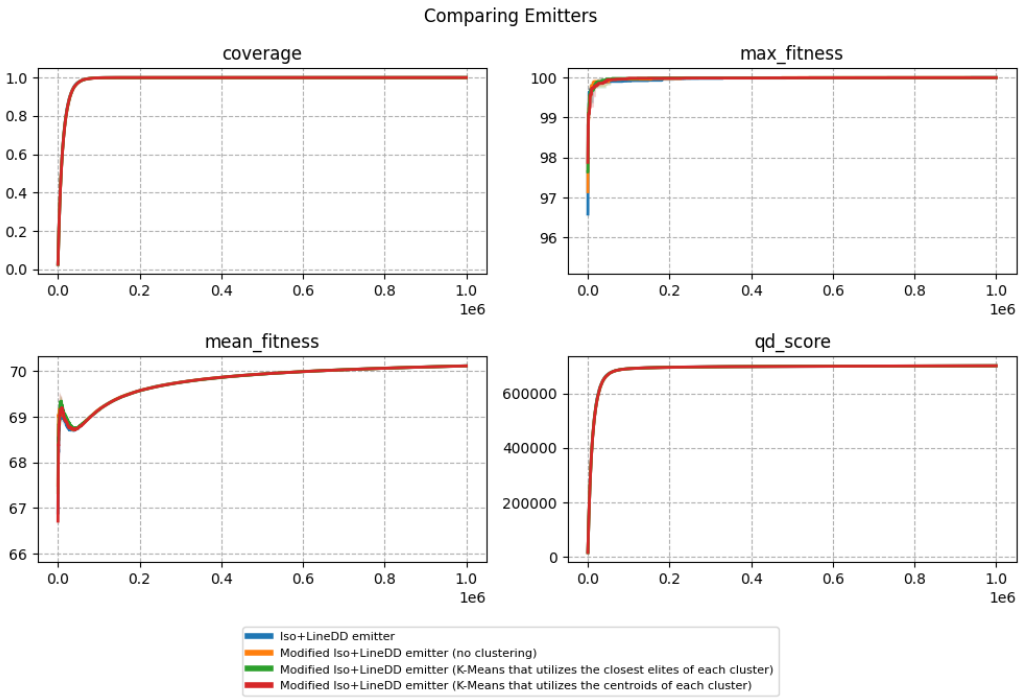


Number of clusters: 2^8

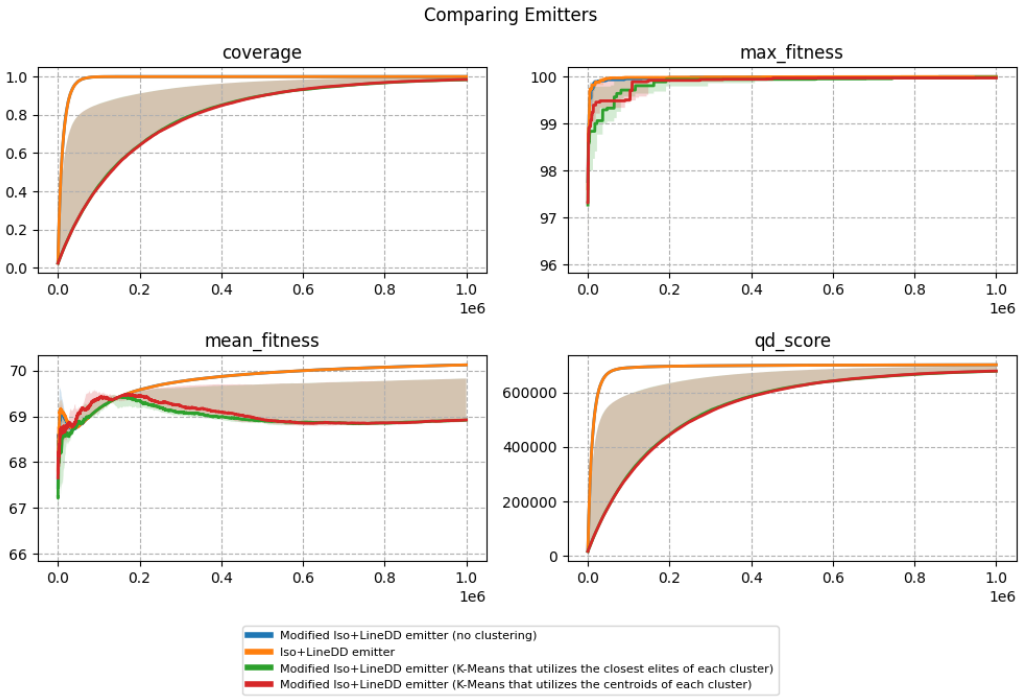


Scenario 2C: 2-D Rastrigin Function with 2^6 Elites

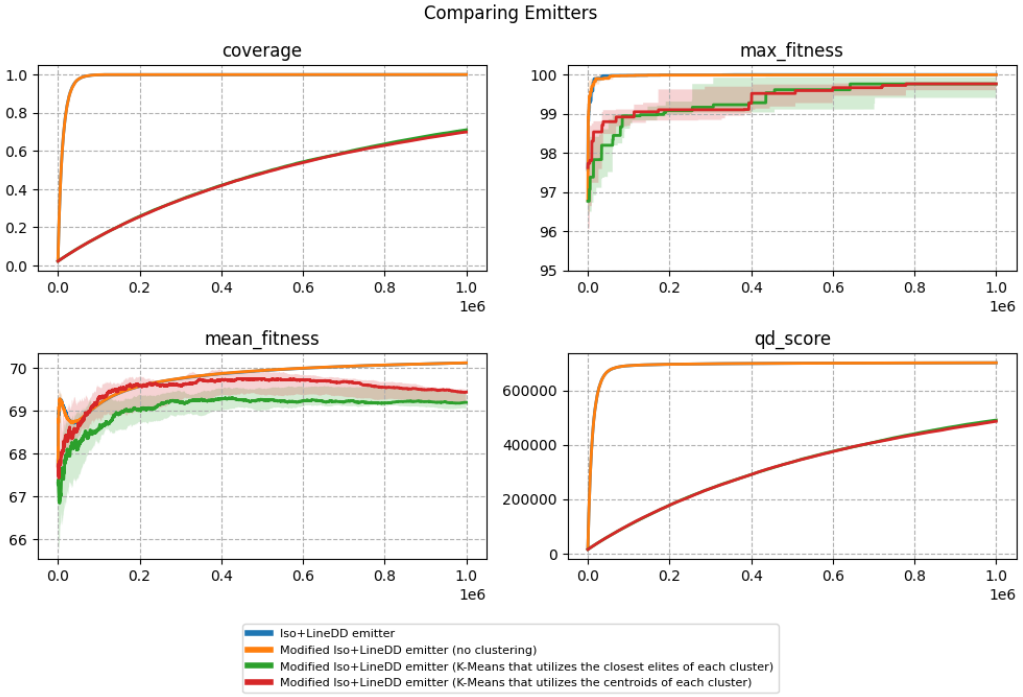
Number of clusters: 2^2



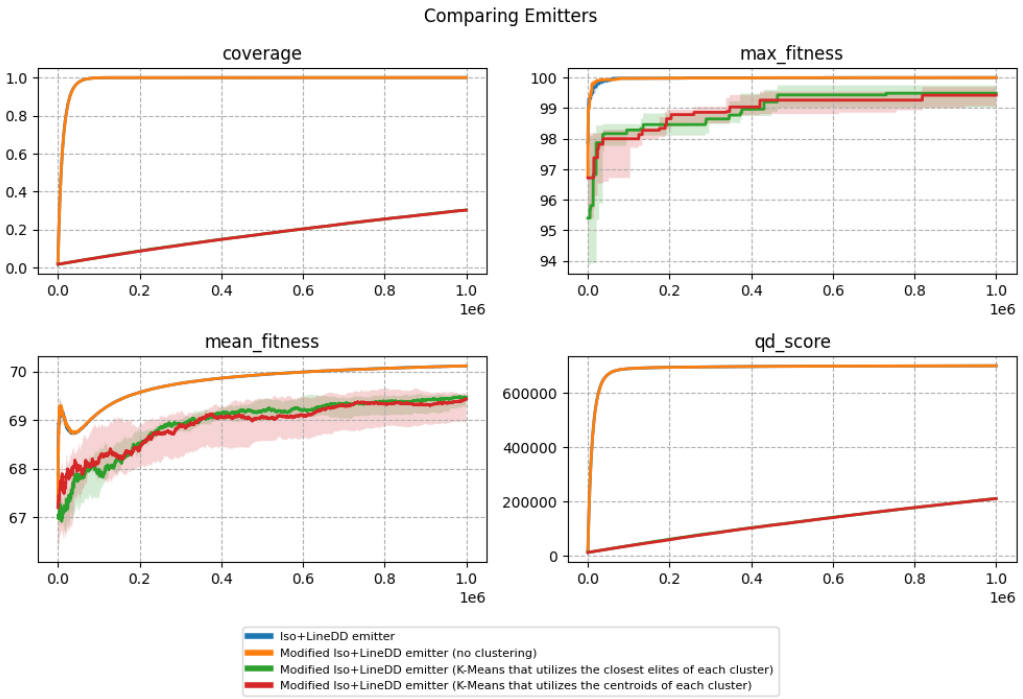
Number of clusters: 2^4



Number of clusters: 2^6



Number of clusters: 2^8



4.2.3. Scenario 3 with 400x400 Grid Size for 2-D Robotic Arm Problem

Scenario 3 deals with the 2-D Robotic Arm problem with a grid dimension of 400x400, thereby expanding the problem's search space. **Despite the change in grid size, many of the observations from Scenario 1 (grid size of 100x100) still hold true**.

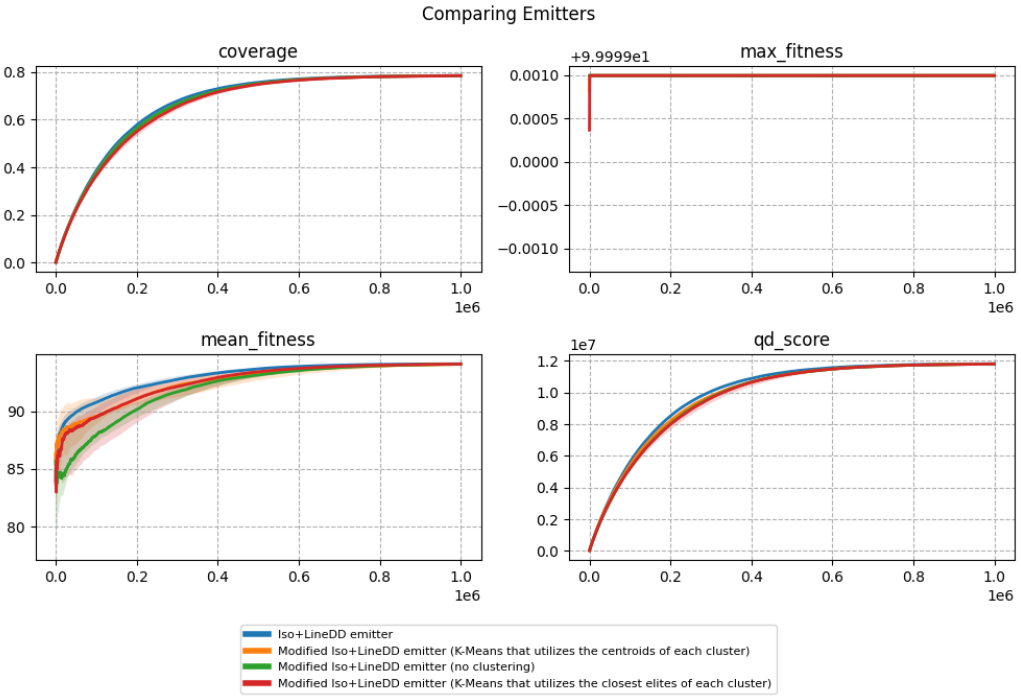
When the number of clusters is relatively small in Scenario 3A, the efficacy of the proposed emitters matches up to that of the Iso+LineDD emitter. However, unlike for the 100x100 grid size, we can see a slight impact of K-Means in the proposed strategies, as **the convergence time of the mean fitness score is slightly slower when compared to the baseline emitter**, **even for this small number of clusters**. One possible reason for this minor change could be attributed to the fact that the emitters are now dealing with a more complex search space. Another reason could be that the emitters following the proposed strategies are more sensitive to their initial placement in the search space due to the increased complexity.

As the number of clusters increases, the effectiveness of the proposed strategies falls short of that of the Iso+LineDD emitter in all metrics yet again, with the exception of max fitness score, which remains unchanged across all emitters. This could suggest that the proposed emitters are concentrating more on exploiting their respective sub-regions, resulting in a reduced coverage and delayed convergence time for the mean fitness score, while still achieving a similar maximum fitness score to the Iso+LineDD emitter.

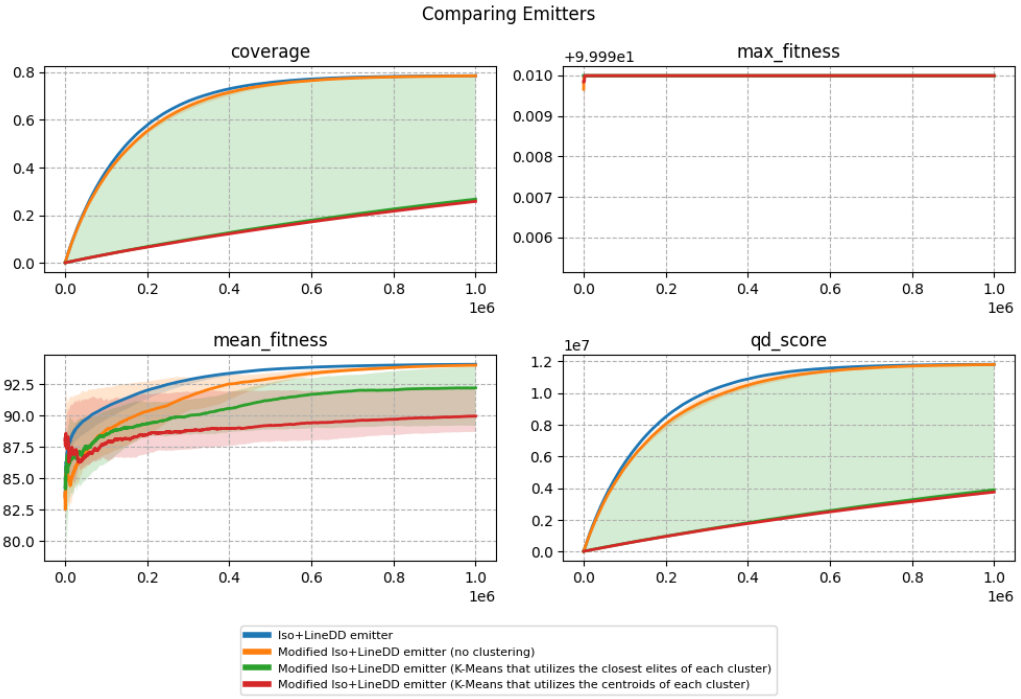
When the number of clusters is small, the efficacy of the emitters in Scenarios 3B and 3C is nearly the same. Interestingly, the emitter without K-Means clustering experiences a faster convergence to the highest mean value when compared with Scenario 3A. In addition, **the maximum fitness remains consistent across all scenarios, regardless of the number of elites in B1.** However, the mean fitness performance differs depending on the batch size of B1 and the number of clusters. **In the case of 16 clusters, Strategy 2 converges faster than Strategy 1 for both 16 and 64 elites (scenarios 3B and 3C) in B1**. Moreover**, for 16 number of clusters in scenario 3C, Strategy 2 also demonstrates significantly better performance in coverage and QD scores**. The performance trends for larger clusters match those observed in Scenario 3A, with the proposed strategies underperforming when compared to the Iso+LineDD emitter.

Scenario 3A: 2-D Robotic Arm with 2^2 Elites

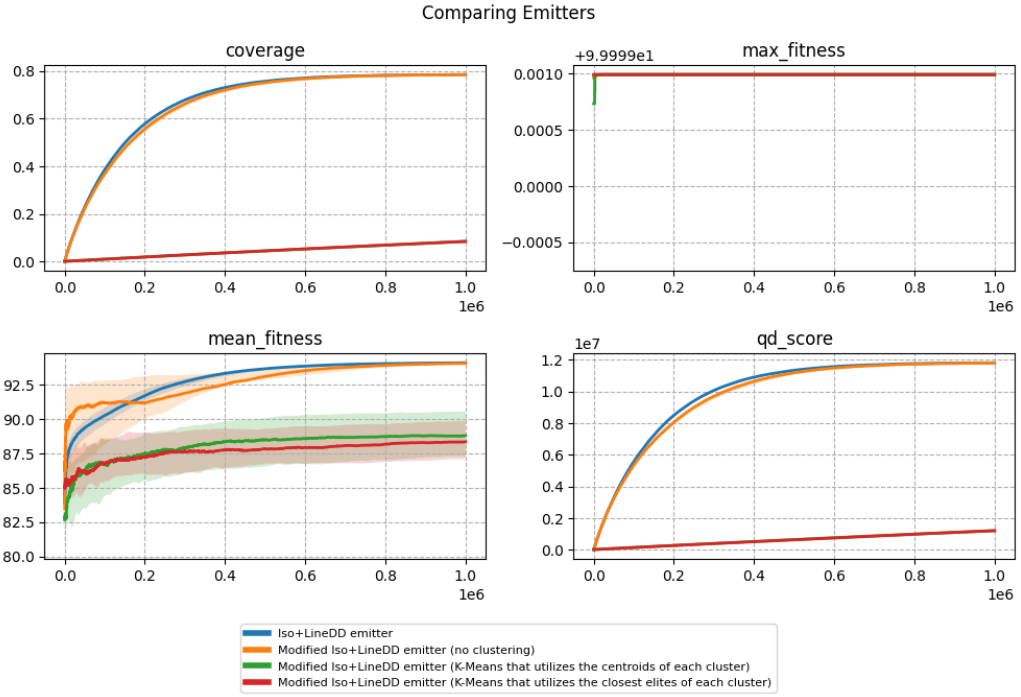
Number of clusters: 2^2



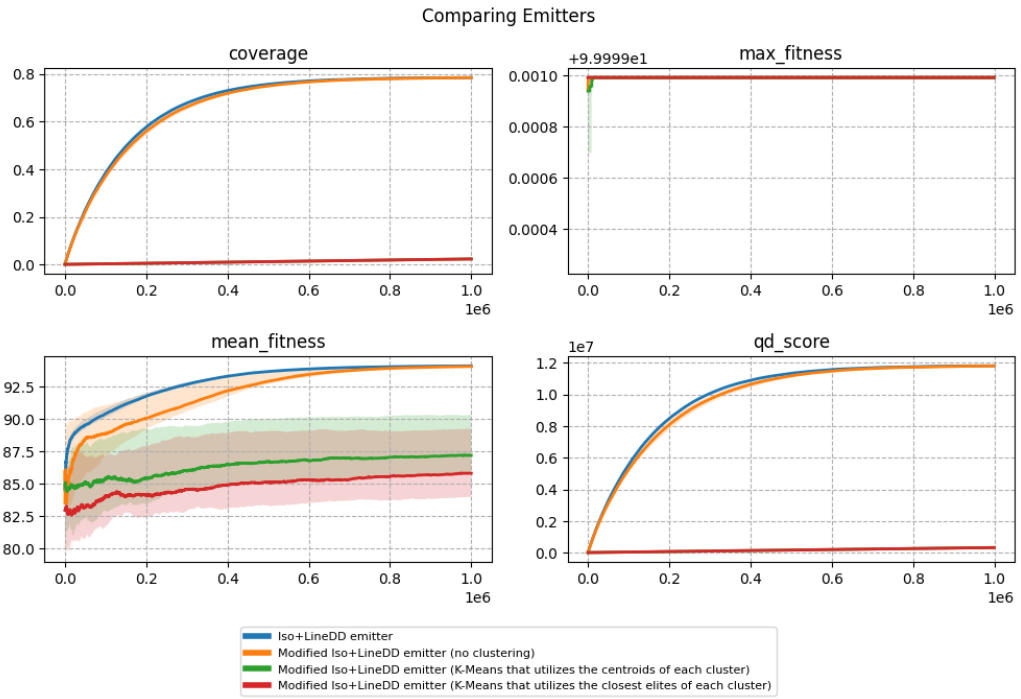
Number of clusters: 2^4



Number of clusters: 2^6

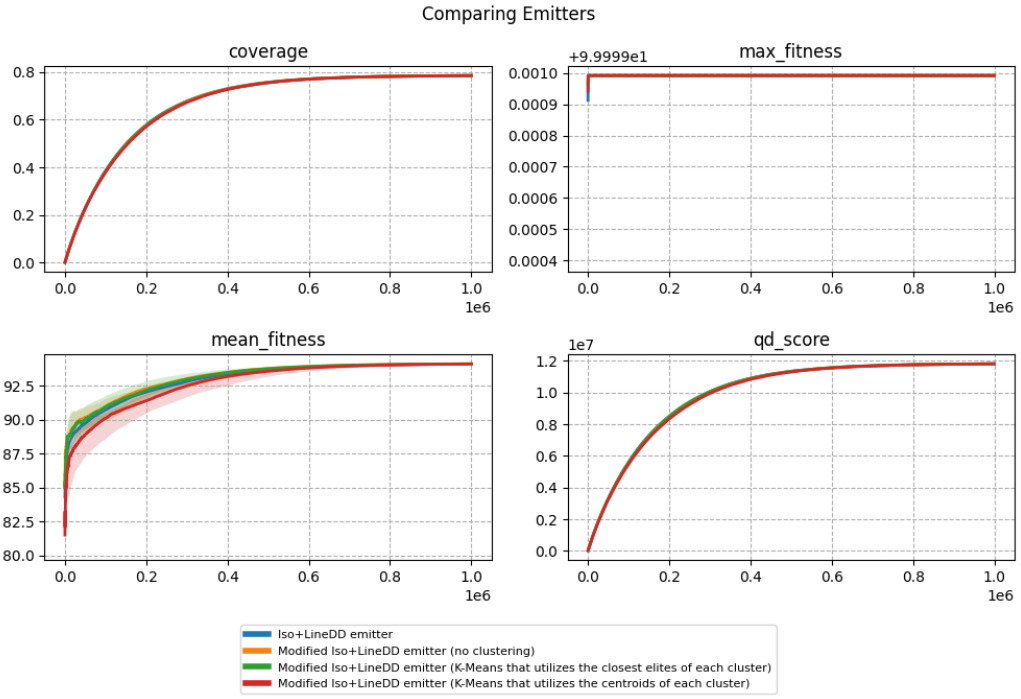


Number of clusters: 2^8

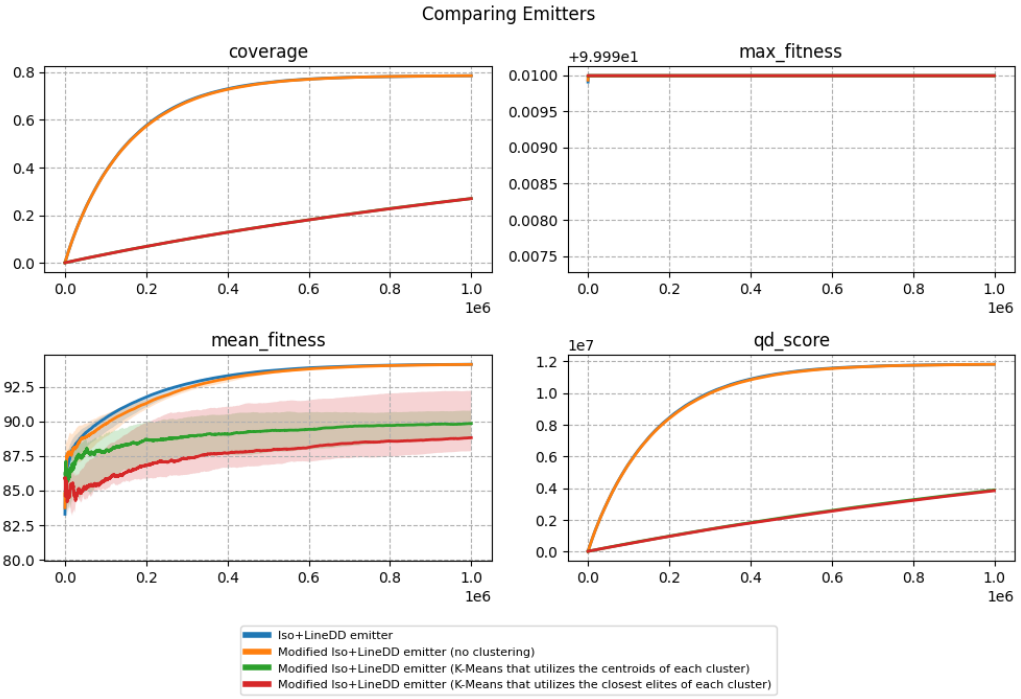


Scenario 3B: 2-D Robotic Arm with 2^4 Elites

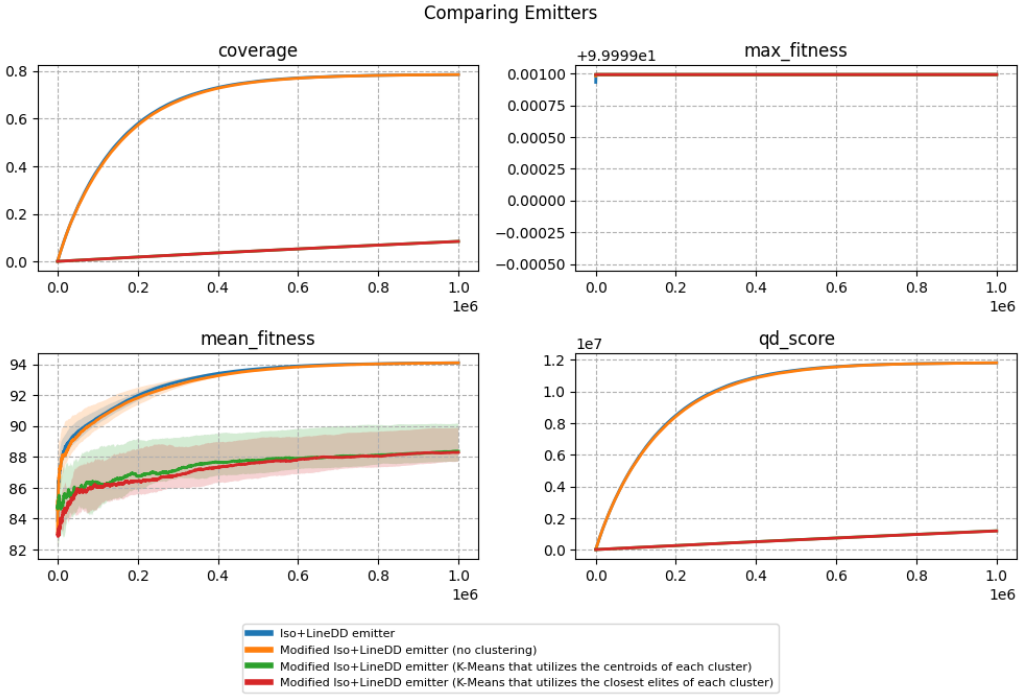
Number of clusters: 2^2



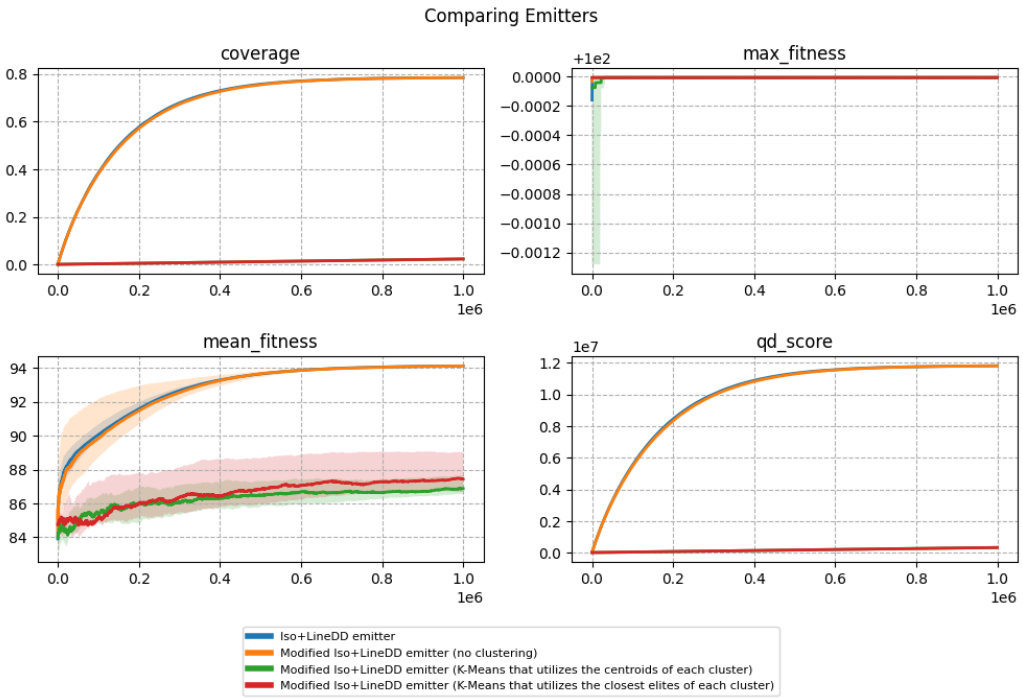
Number of clusters: 2^4



Number of clusters: 2^6

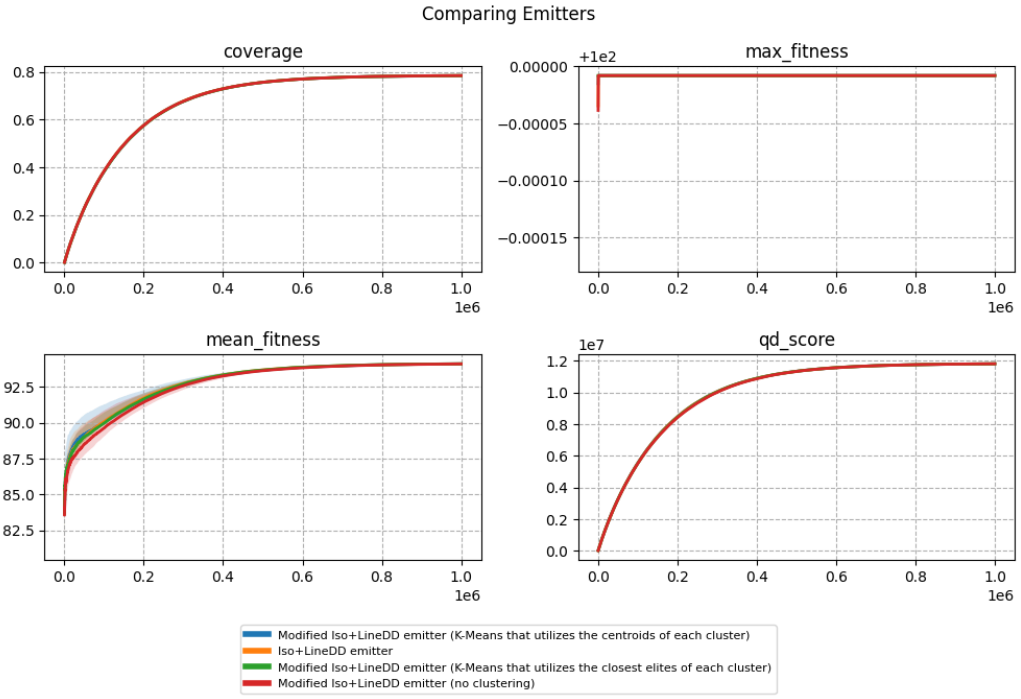


Number of clusters: 2^8

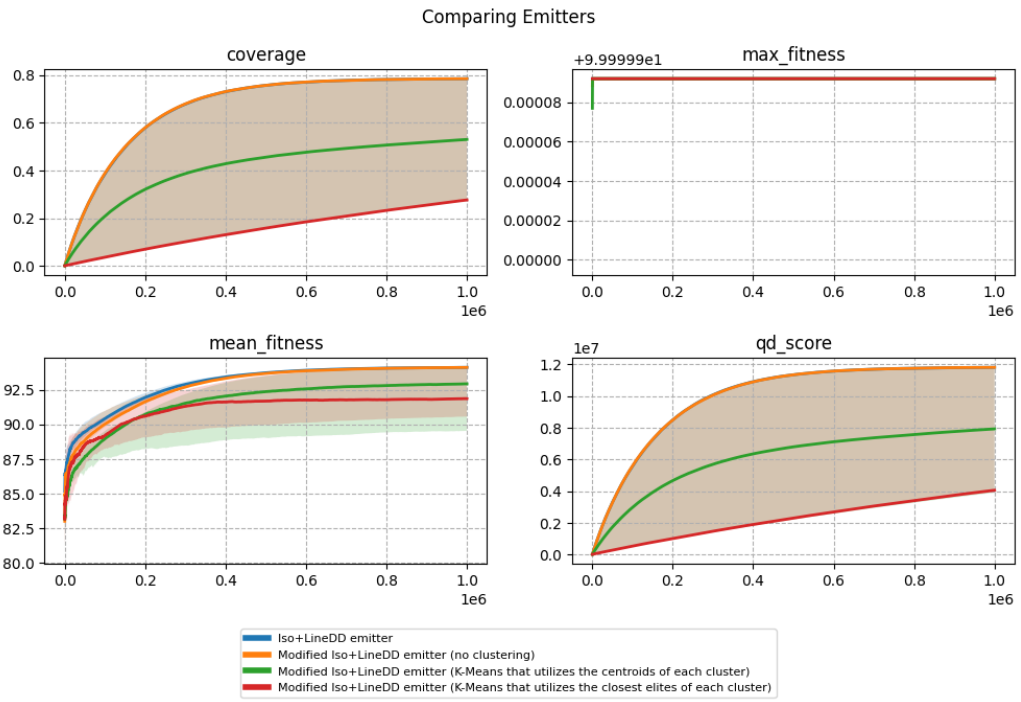


Scenario 3C: 2-D Robotic Arm with 2^6 Elites

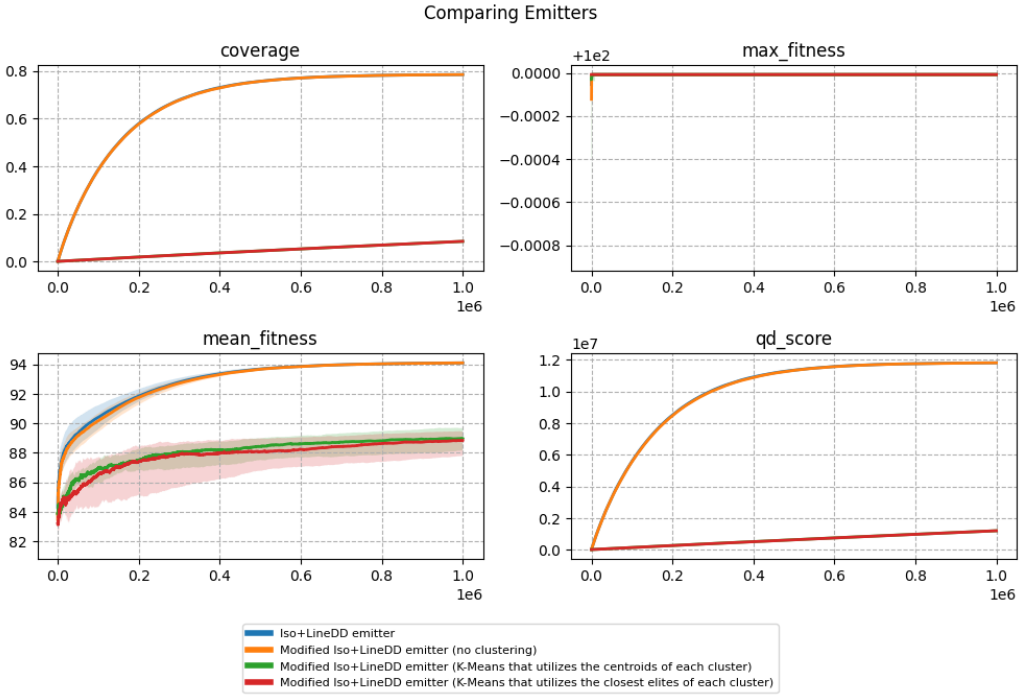
Number of clusters: 2^2



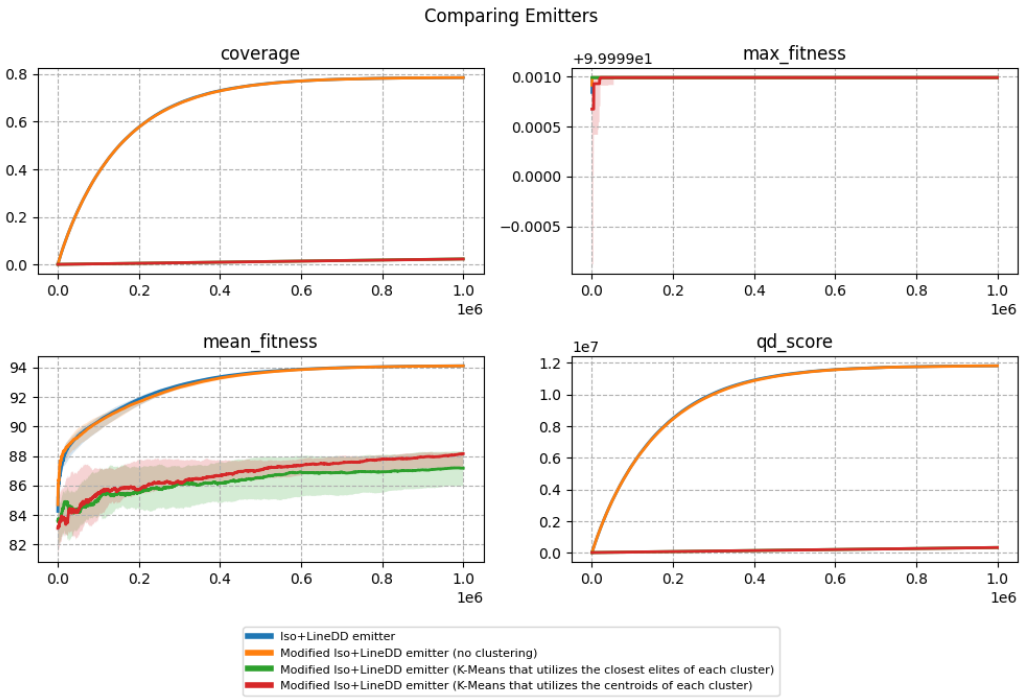
Number of clusters: 2^4



Number of clusters: 2^6



Number of clusters: 2^8



4.2.4. Scenario 4 with 400x400 Grid Size for 2-D Rastrigin Function

In Scenario 4, the 2-D Rastrigin Function is evaluated on a 400x400 grid size. This scenario produced a number of insightful observations.

In each scenario (4A, 4B, and 4C), the results demonstrated that the metrics for the small number of clusters were very similar across all emitters. Nonetheless, **for the first time across all the experiments, the proposed strategies began with a mean fitness score that exceeded that of the Iso+LineDD emitter**. This advantage diminished as the number of iterations increased, most likely due to K-Means' limited impact when dealing with a small number of clusters. Regardless, it hinted at the possibility for better results in higher number of clusters.

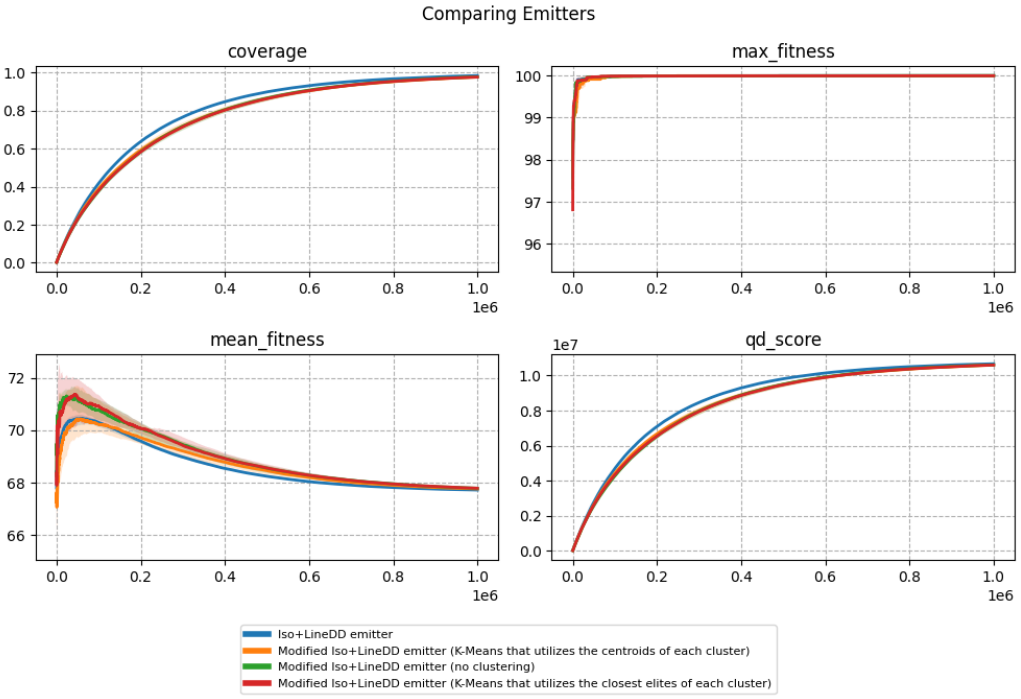
Furthermore, **the mean fitness score for the proposed strategies was considerably higher than that of the Iso+LineDD emitter, for larger number of clusters, regardless of the batch size in B1**. In some scenarios, such as Scenario 4A, Strategy 1 performed better than Strategy 2, while in others, such as Scenario 4B, Strategy 2 performed better. However, both strategies yielded fairly similar results. This marked an important progress, as **it is the first time the proposed strategies have demonstrated a higher mean fitness than the Iso+LineDD emitter**. This improvement may be related to the increased complexity of the search space, which may have made the Iso+LineDD emitter less effective, allowing the proposed strategies to find better mean fitness scores in this scenario. This improvement may also be attributed to the Rastrigin function, which is characterized by multiple local optima. The function may be more favorable towards the proposed emitters' search strategy, allowing them to investigate space more efficiently.

However, this also had an effect on the proposed emitters' max fitness score metrics. **While they managed to still find the highest maximum fitness or a score very close to the maximum fitness of the Iso+LineDD emitter, their convergence rate was slightly slower**.

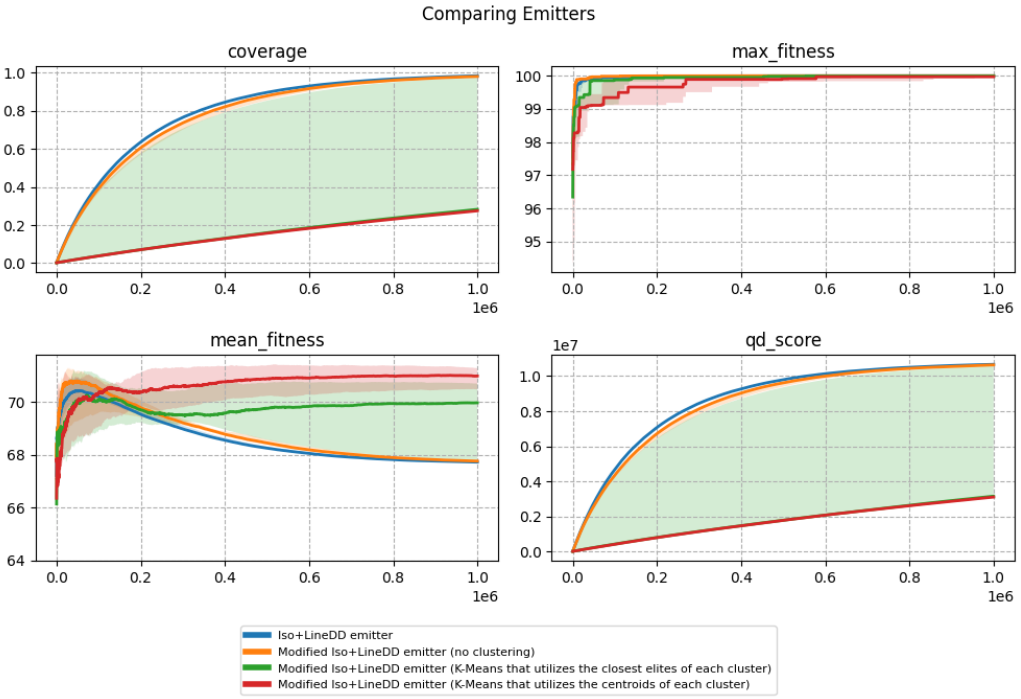
Aside from these findings, no other significant differences were observed compared to the previous scenarios.

Scenario 4A: 2-D Rastrigin Function with 2^2 Elites

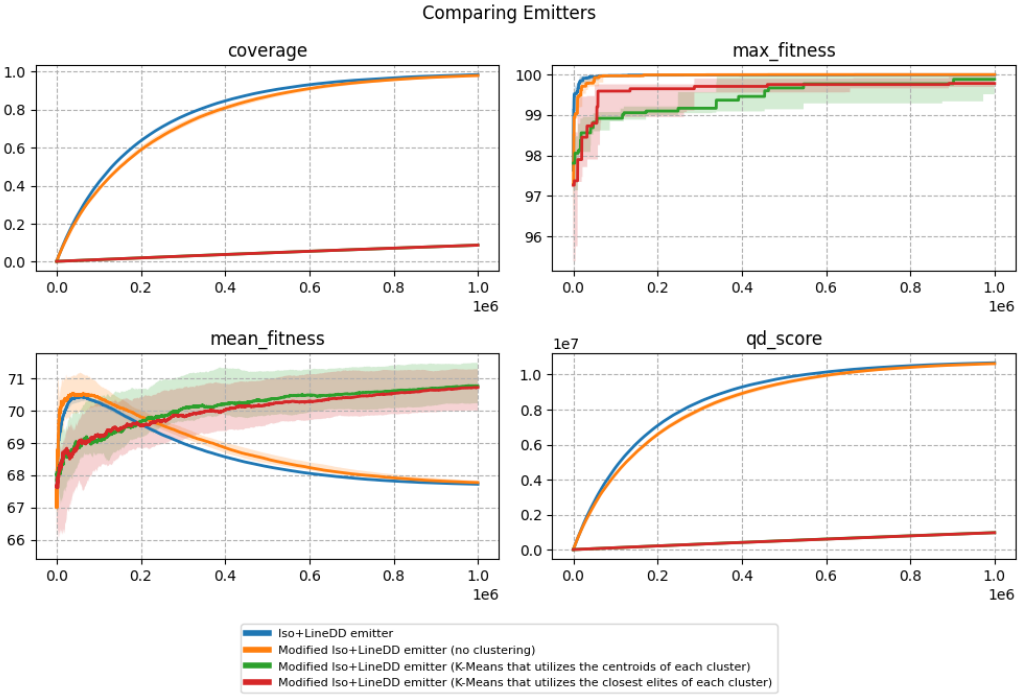
Number of clusters: 2^2



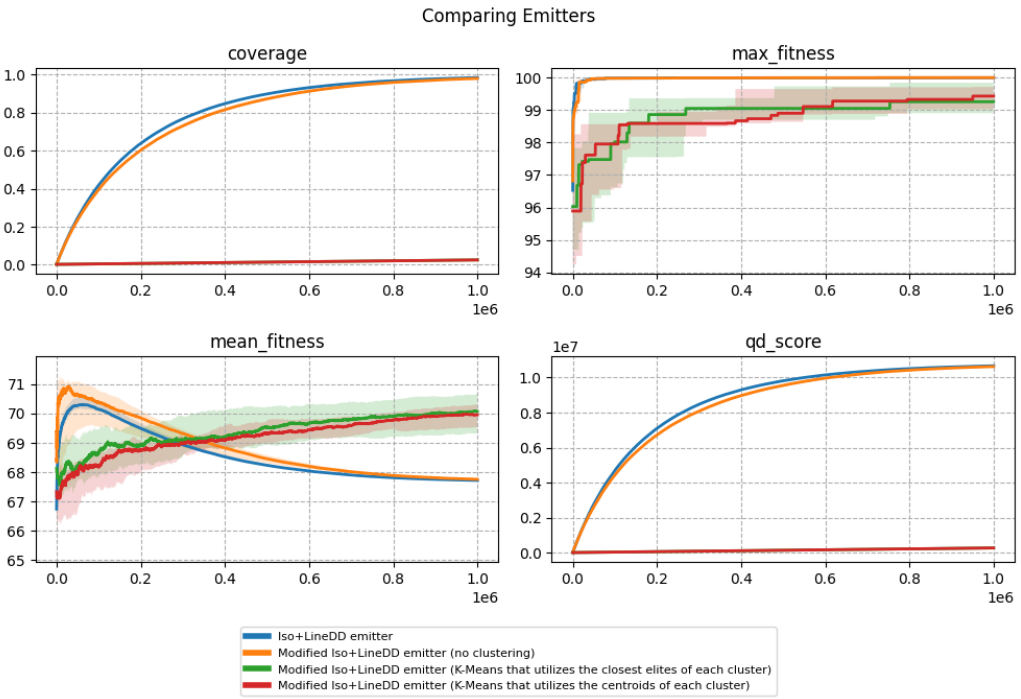
Number of clusters: 2^4



Number of clusters: 2^6

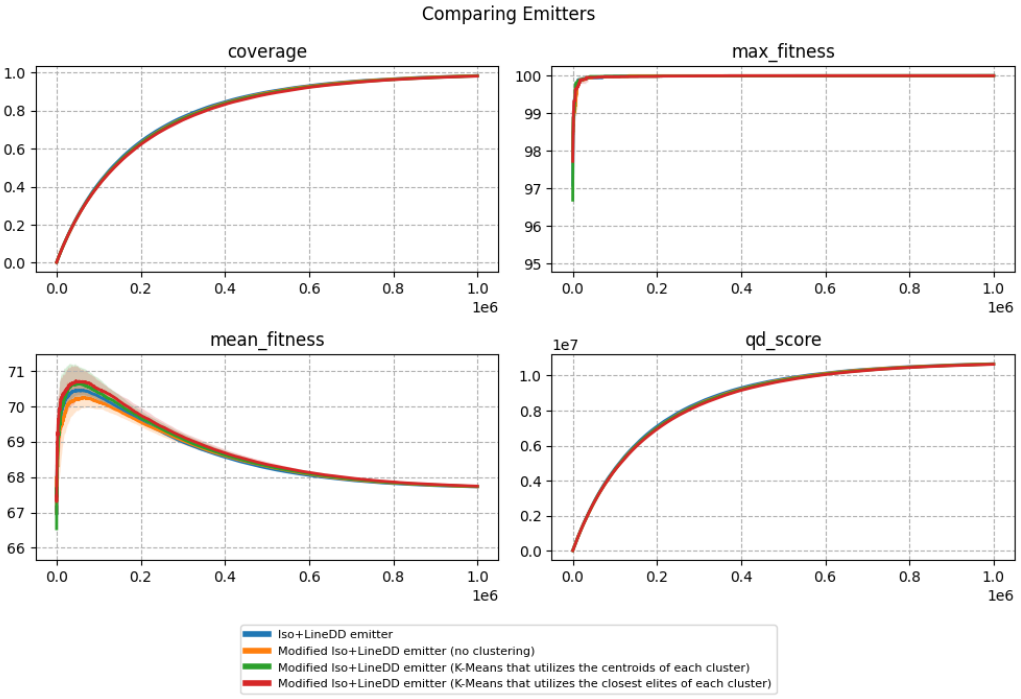


Number of clusters: 2^8

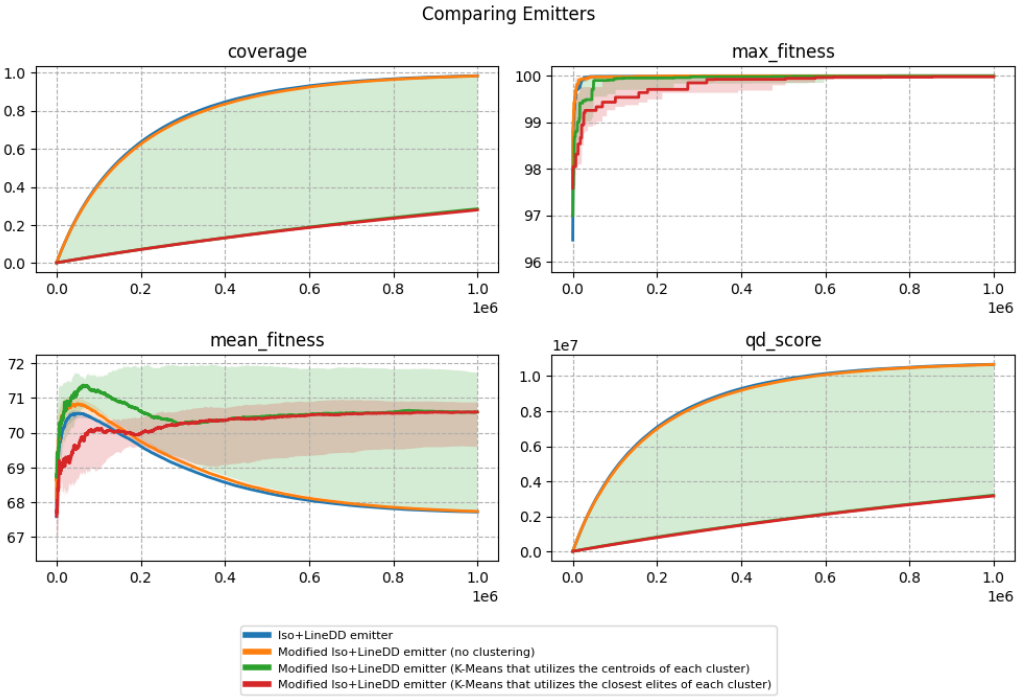


Scenario 4B: 2-D Rastrigin Function with 2^4 Elites

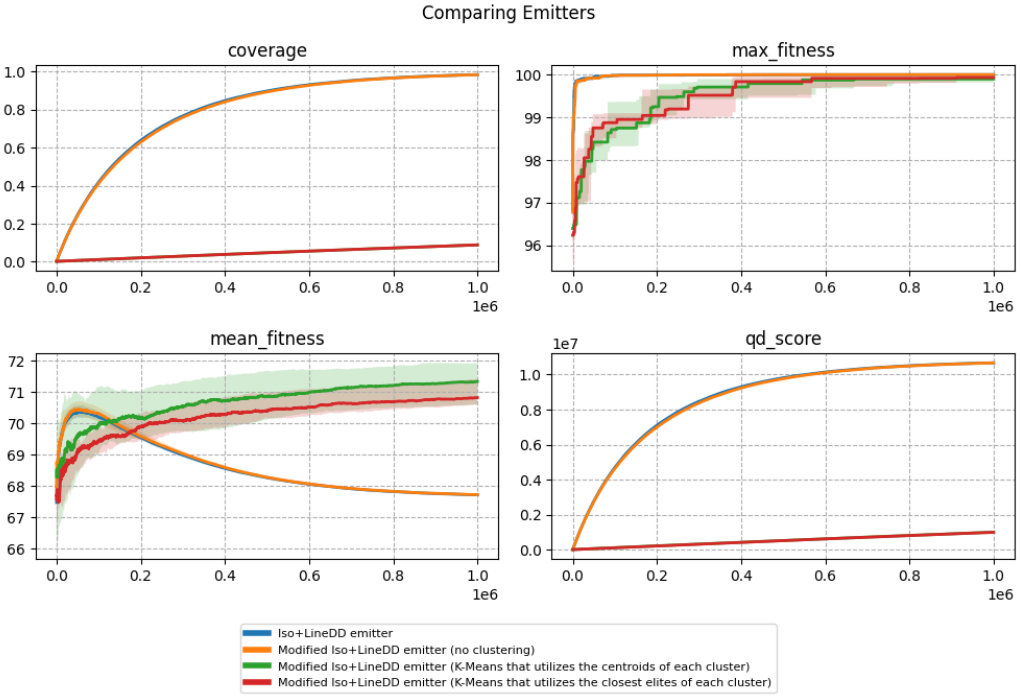
Number of clusters: 2^2



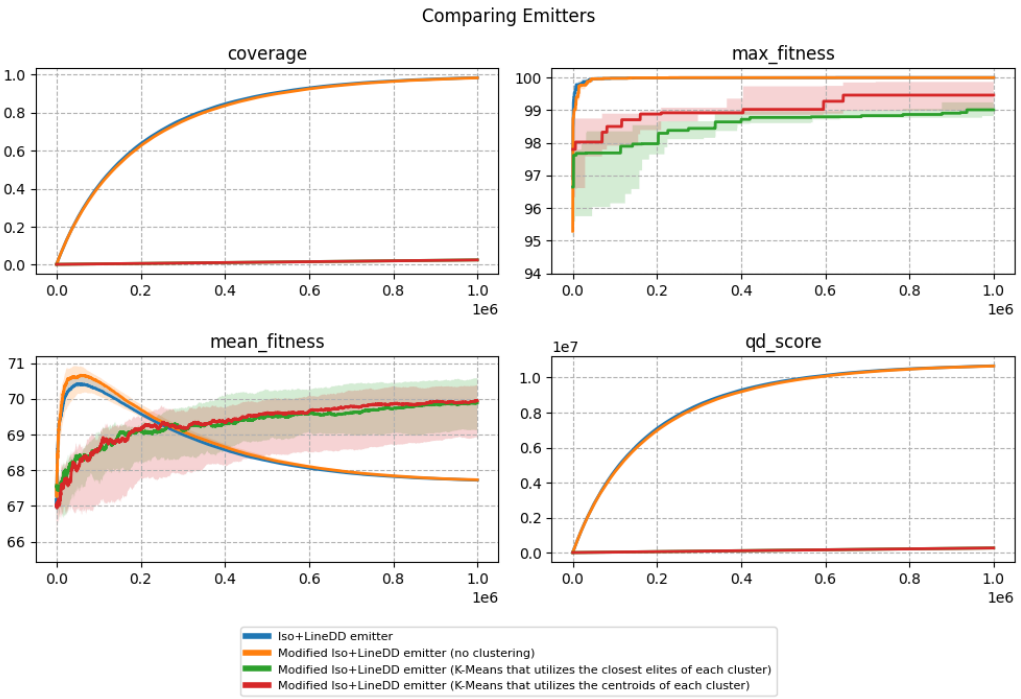
Number of clusters: 2^4



Number of clusters: 2^6

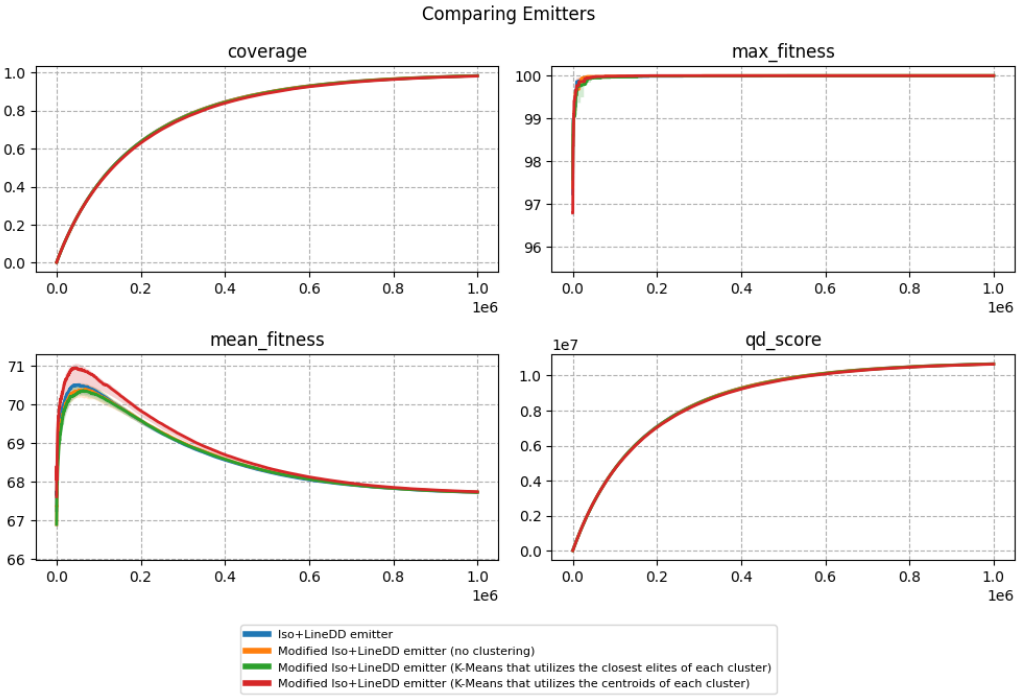


Number of clusters: 2^8

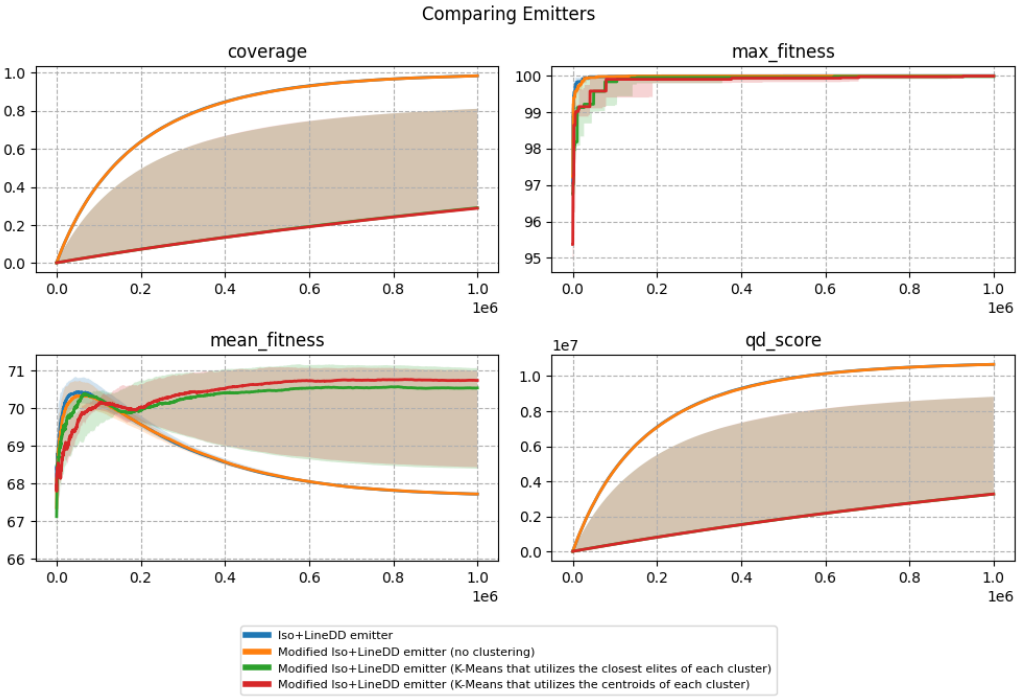


Scenario 4C: 2-D Rastrigin Function with 2^6 Elites

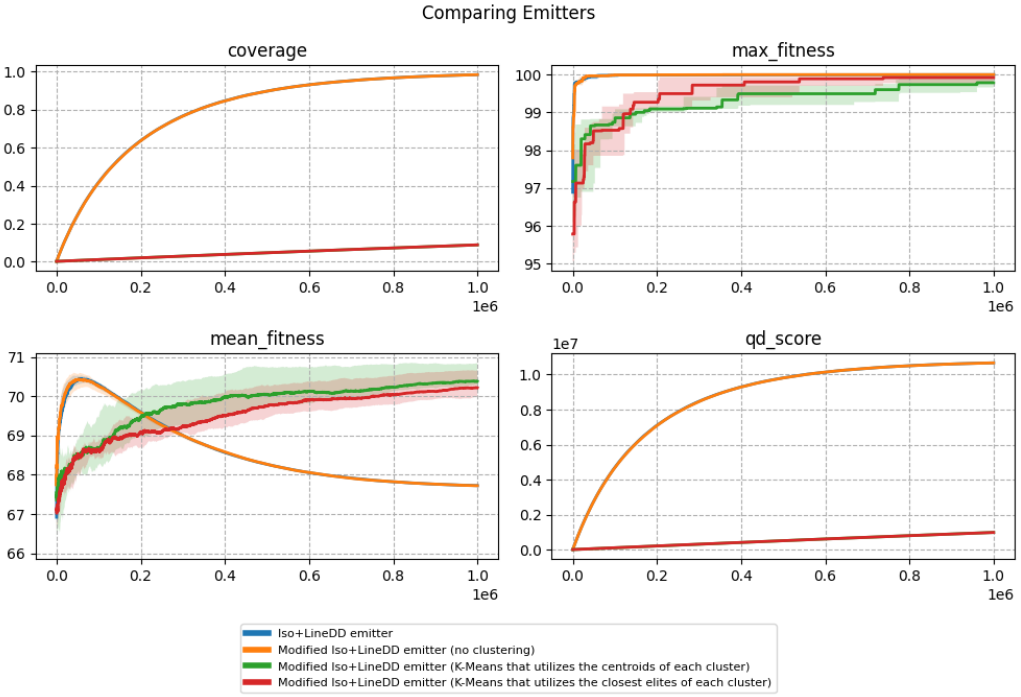
Number of clusters: 2^2



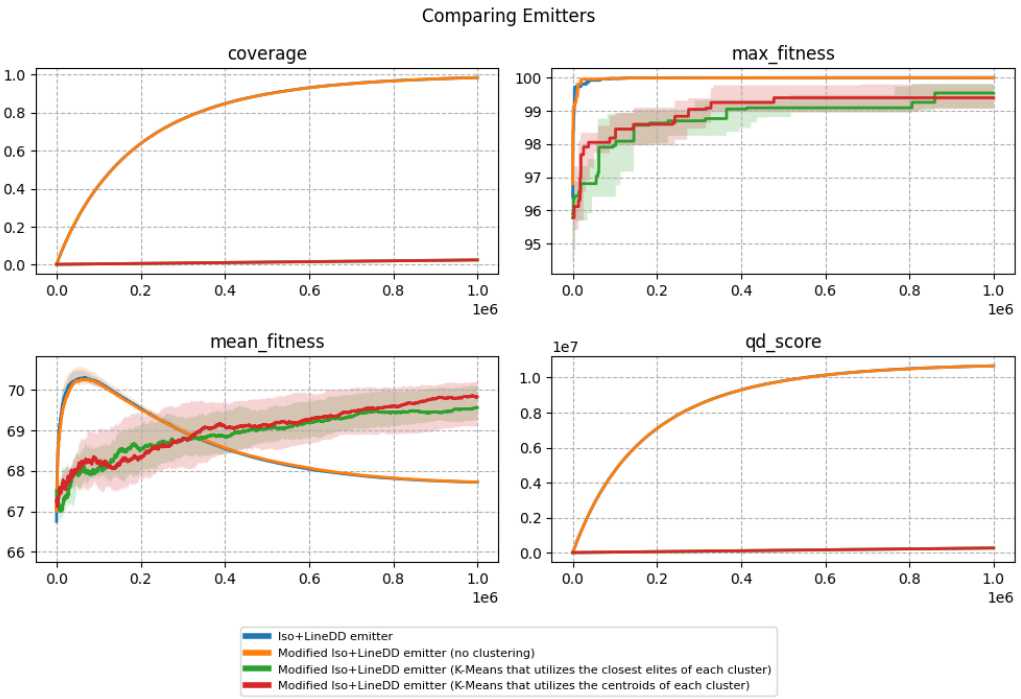
Number of clusters: 2^4



Number of clusters: 2^6



Number of clusters: 2^8



4.3. Results for Higher-Dimensional Problem Domains

In this section, the discussion for the experimental results continues for higher dimensional problem domains, specifically addressing Scenarios 5 - 8, which involve 10-dimensional and 100-dimensional problems. We evaluate the performance of the proposed strategies, Perturbation Towards Elites Closest to Cluster Centers (Strategy 1) and Perturbation Towards Cluster Centroids (Strategy 2), along with the Modified Iso+LineDD without K-Means Clustering, by comparing them against the baseline Iso+LineDD emitter. The evaluation focuses again on key performance metrics, including Coverage, Max Fitness, Mean Fitness, and QD Score. Same with previous experiments, the emitter with the highest QD score in each experiment is denoted with the color blue, followed by the second highest in orange, the third highest in green, and the fourth highest in red. Through these set of experiments, we aim to gain further insights into the effectiveness of the proposed strategies in more complex, higher-dimensional problem domains

4.3.1. Scenarios 5 - 6 with 100x100 Grid Size for 10-D Problem Domains

In this section, the results for both Scenario 5 and Scenario 6 are discussed together, as they did not produce any significant differences when compared to each other. These scenarios deal with 10-dimensional problem domains, specifically the 10-dimensional Robotic Arm problem and the 10-dimensional Rastrigin Function.

As also observed in previous scenarios, **when the number of clusters is small, the metrics for the Iso+LineDD emitter and the proposed emitters are nearly identical**. However, when tested with a larger number of clusters, the performance of the proposed strategies varied.

In the case of the 10-dimensional Robotic Arm problem (Scenario 5), the proposed strategies failed to match the performance of the Iso+LineDD emitter for most metrics, except for a few specific cases. From the plots, it can be seen that the proposed strategies managed to achieve the same max fitness as the baseline emitter in all cases, except when the number of clusters was 256; in this case, Strategy 2 failed to reach the same max fitness, and its convergence was much slower.

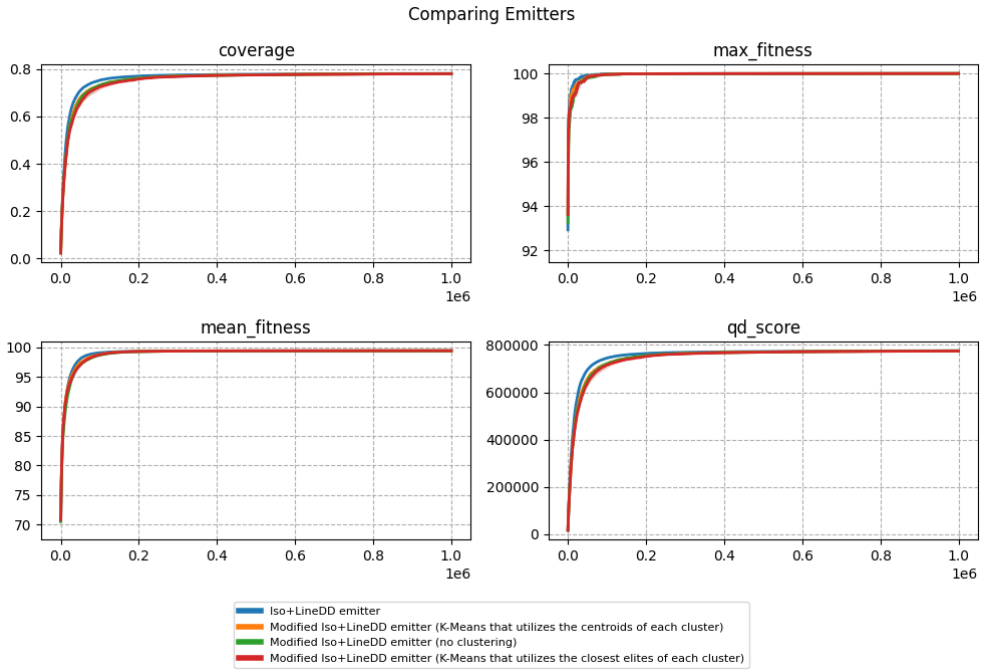
For the 10-dimensional Rastrigin Function (Scenario 6), the performance of the proposed strategies was mixed. Strategy 2 matched the highest max fitness of the Iso+LineDD emitter in Scenario 6A, while Strategy 1 did not. The opposite occurred in Scenario 6B, where Strategy 1 matched the highest max fitness, but Strategy 2 did not. In other cases, the proposed emitters failed to achieve the same max fitness as the baseline emitter.

The proposed emitters managed to match the performance of the Iso+LineDD emitter in other metrics on a few occasions for both scenarios. This occurred in Scenario 5B with 16 clusters (Strategy 1), Scenario 6A with 16 clusters (Strategy 2), and Scenario 6B with 16 clusters (Strategy 1). Here, our strategies matched the baseline emitter's performance in coverage, average fitness, and QD scores. But outside of these instances, the proposed emitters showed slower convergence and lower scores for these scenarios.

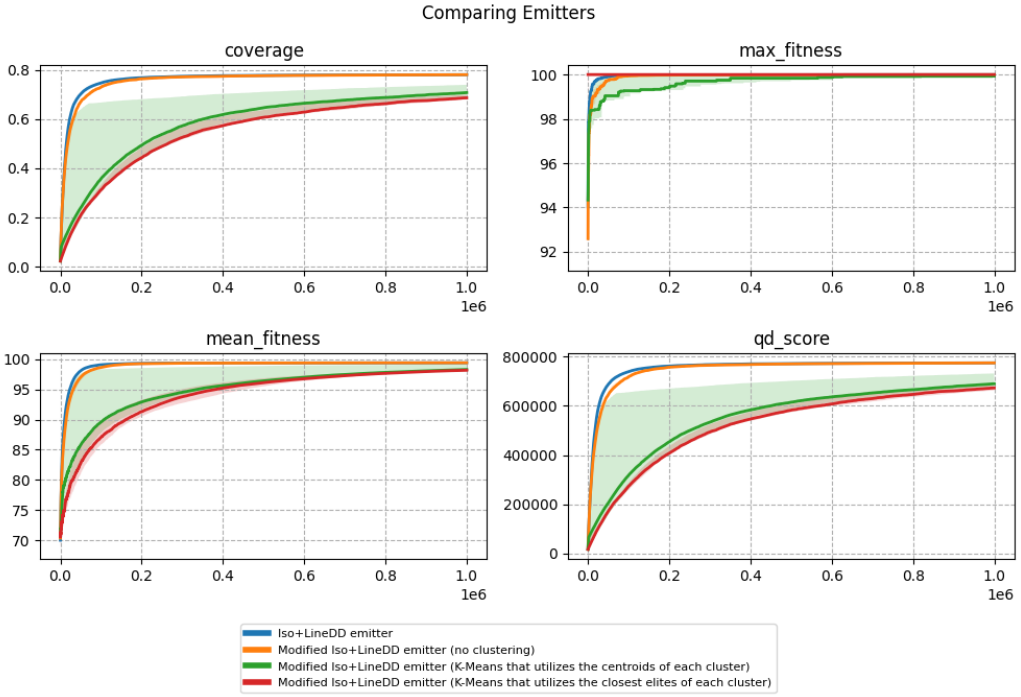
The difference in performance between Scenarios 5 and 6 and our proposed emitters can be attributed to several factors. One possibility is that 10-dimensional problem domains are more complex than 2-dimensional problem domains, which could result in more difficult search spaces. This makes it more challenging for our proposed emitters to match the Iso+LineDD emitter's effectiveness. The greater complexity of these problem domains may also imply that our proposed emitters are more affected by their initial placement in the search space, which could result in varying performance outcomes, as observed in the different scenarios. There is also the possibility that our strategies are better tailored towards particular problem domain types, or that the selected parameters for these experiments, such as cluster number may not be optimal for the 10-dimensional problem domains.

Scenario 5A: 10-D Robotic Arm with 2^2 Elites

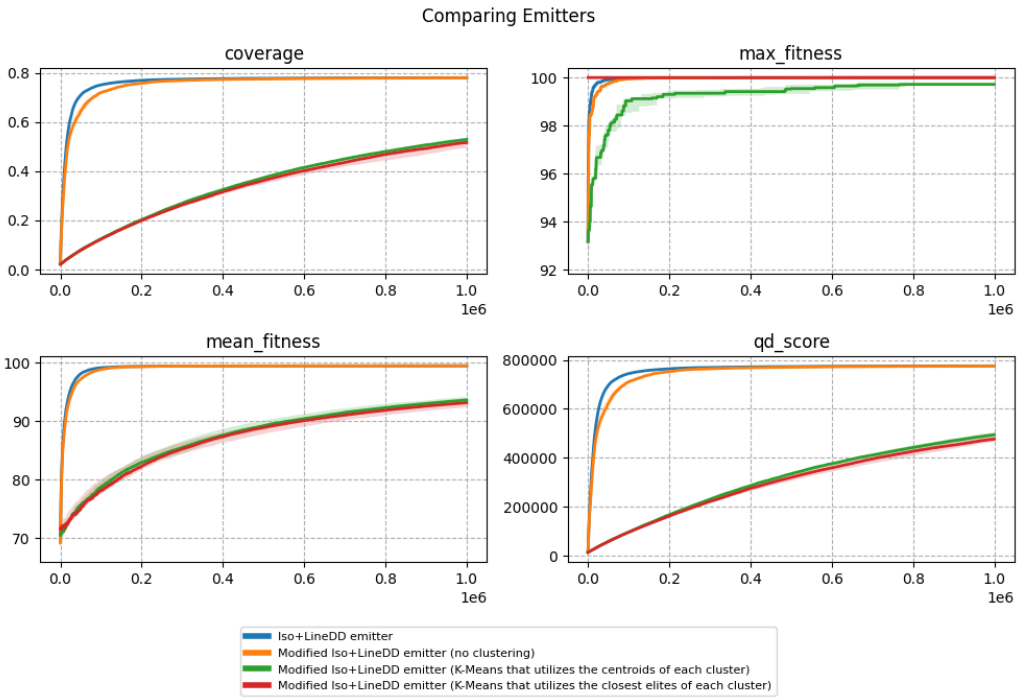
Number of clusters: 2^2



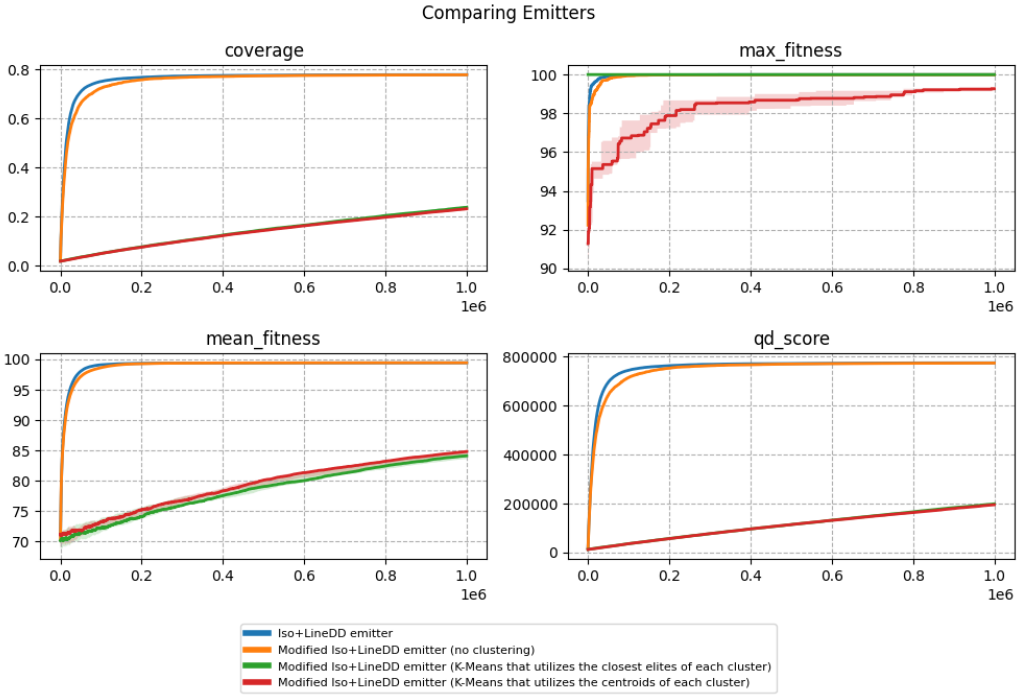
Number of clusters: 2^4



Number of clusters: 2^6

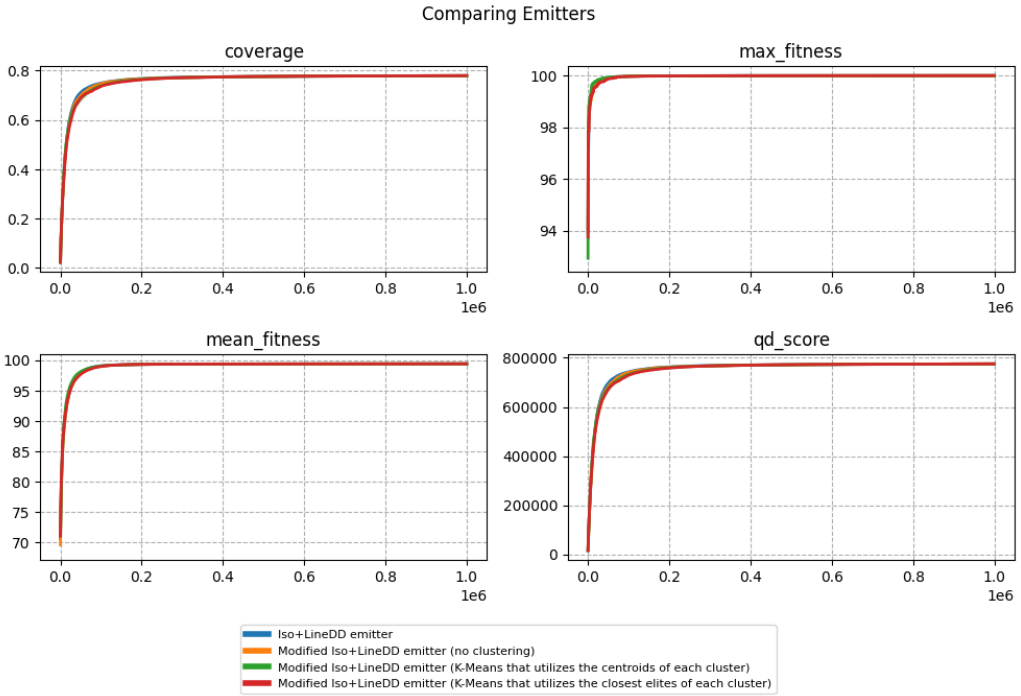


Number of clusters: 2^8

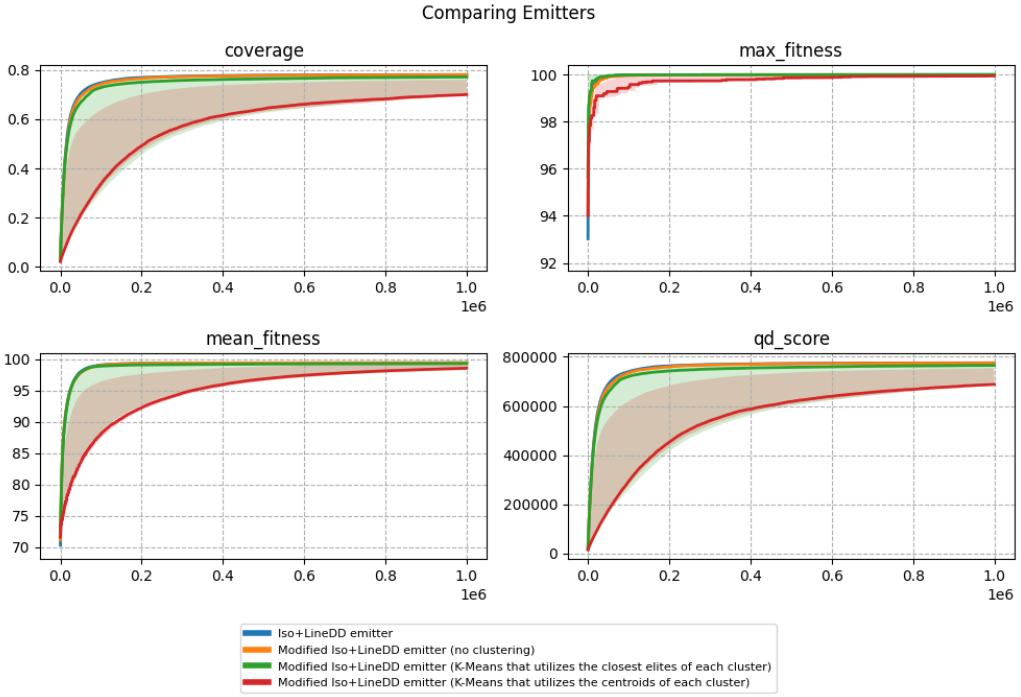


Scenario 5B: 10-D Robotic Arm with 2^4 Elites

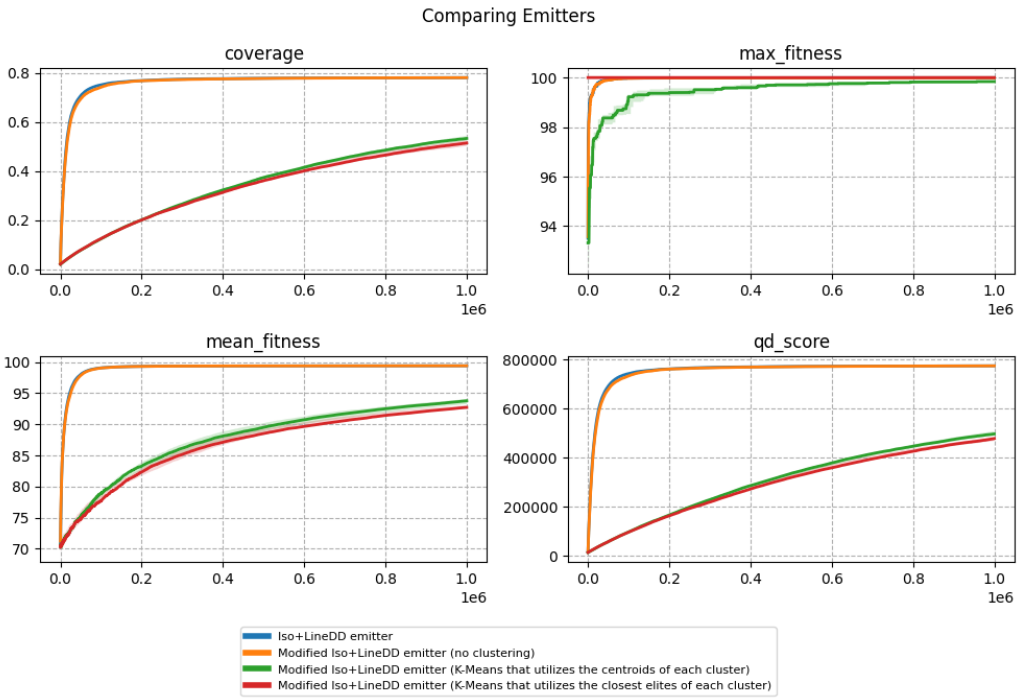
Number of clusters: 2^2



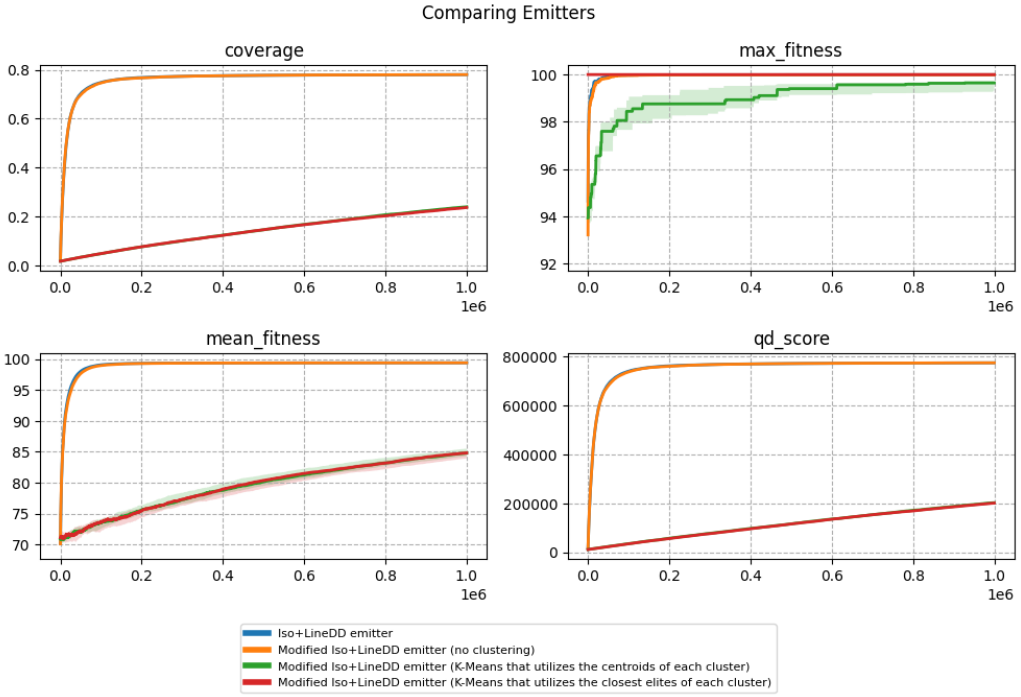
Number of clusters: 2^4



Number of clusters: 2^6

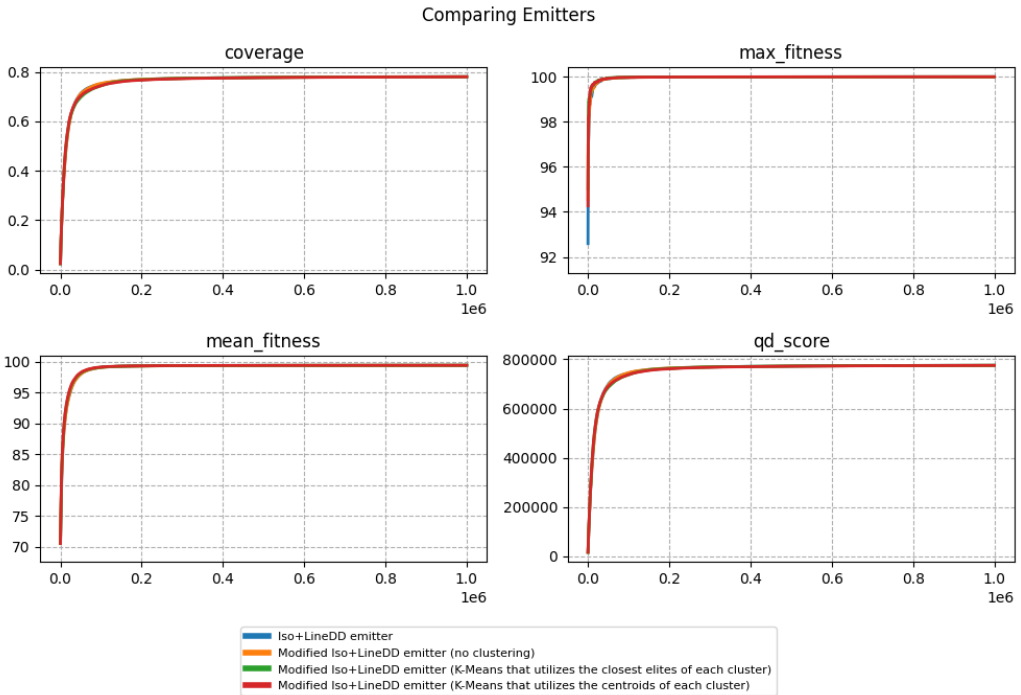


Number of clusters: 2^8

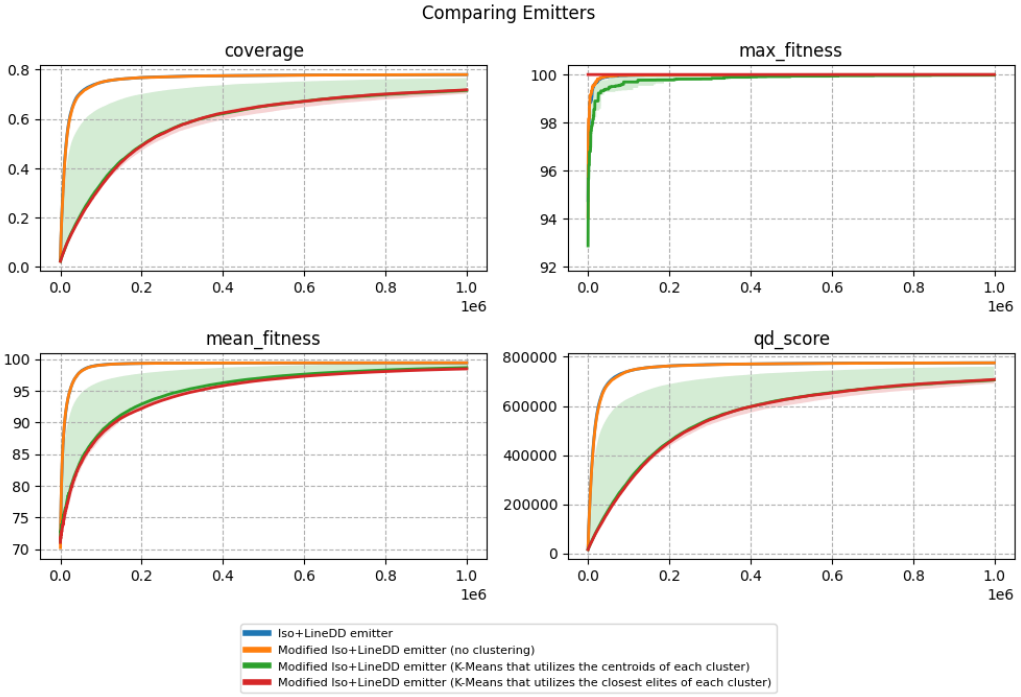


Scenario 5C: 10-D Robotic Arm with 2^6 Elites

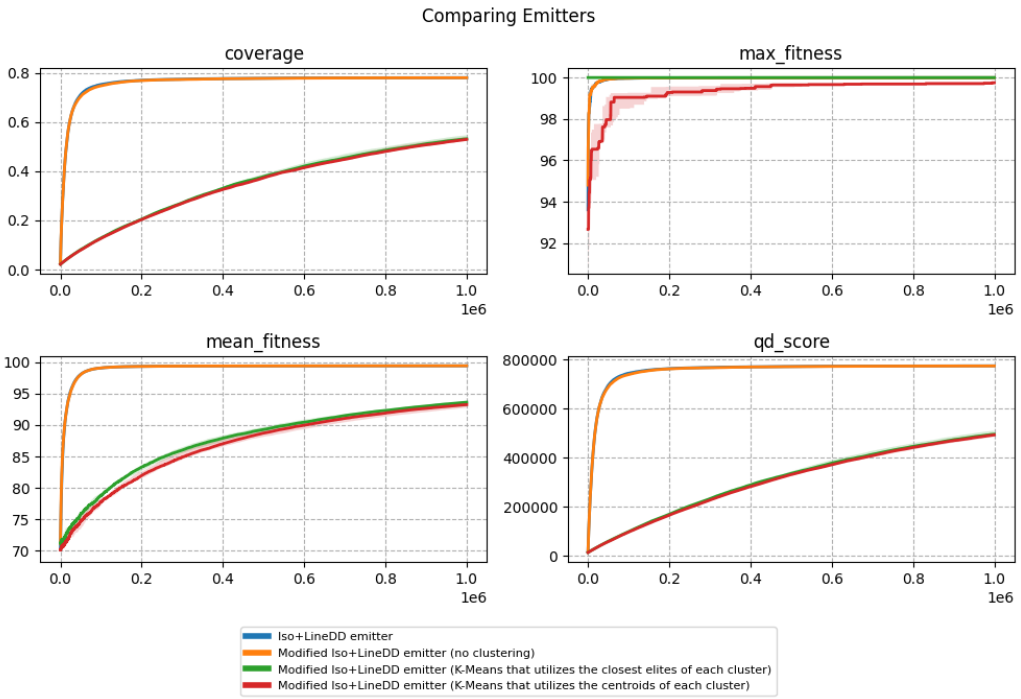
Number of clusters: 2^2



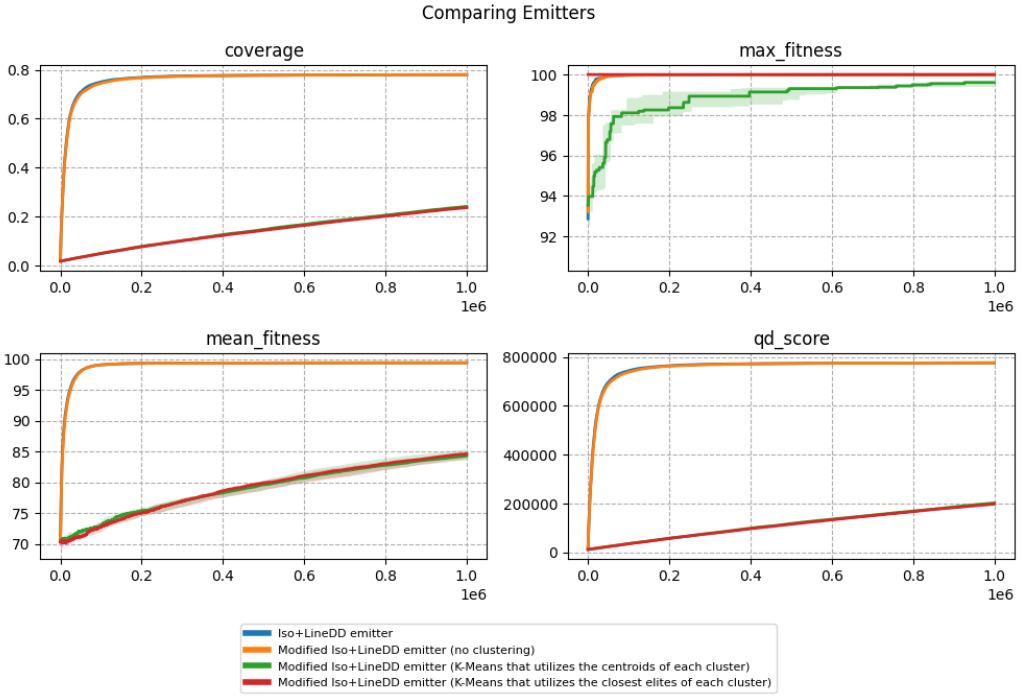
Number of clusters: 2^4



Number of clusters: 2^6

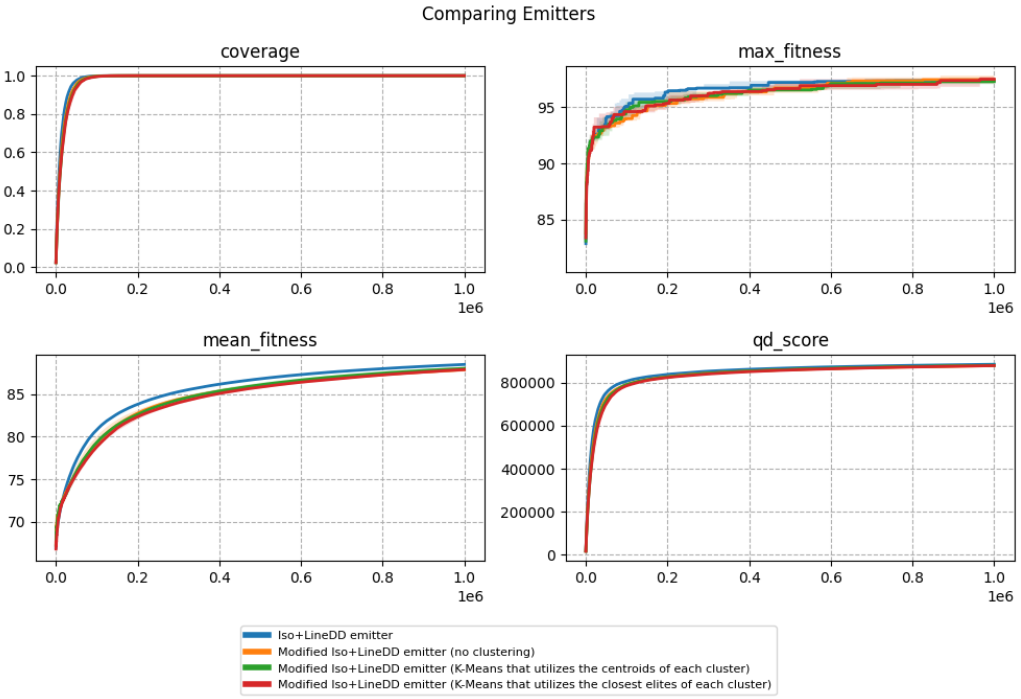


Number of clusters: 2^8

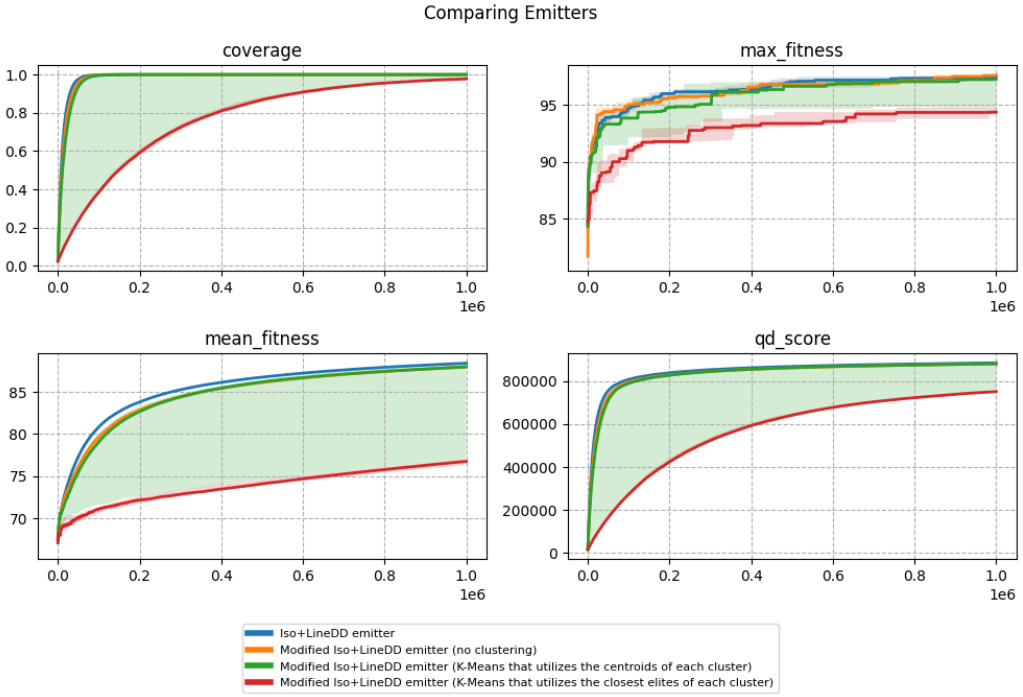


Scenario 6A: 10-D Rastrigin Function with 2^2 Elites

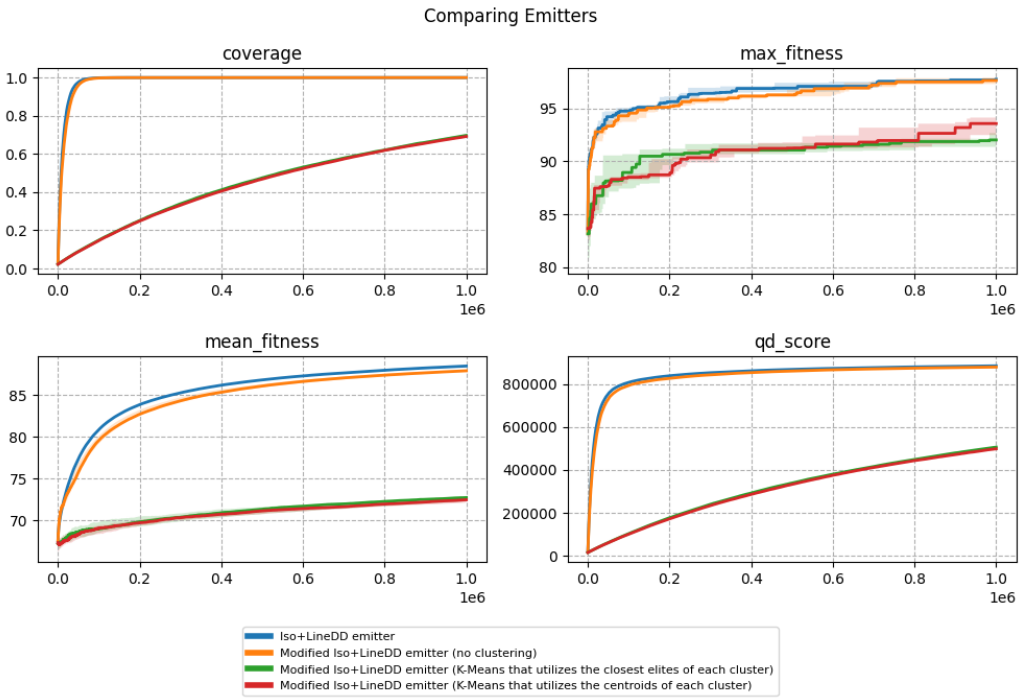
Number of clusters: 2^2



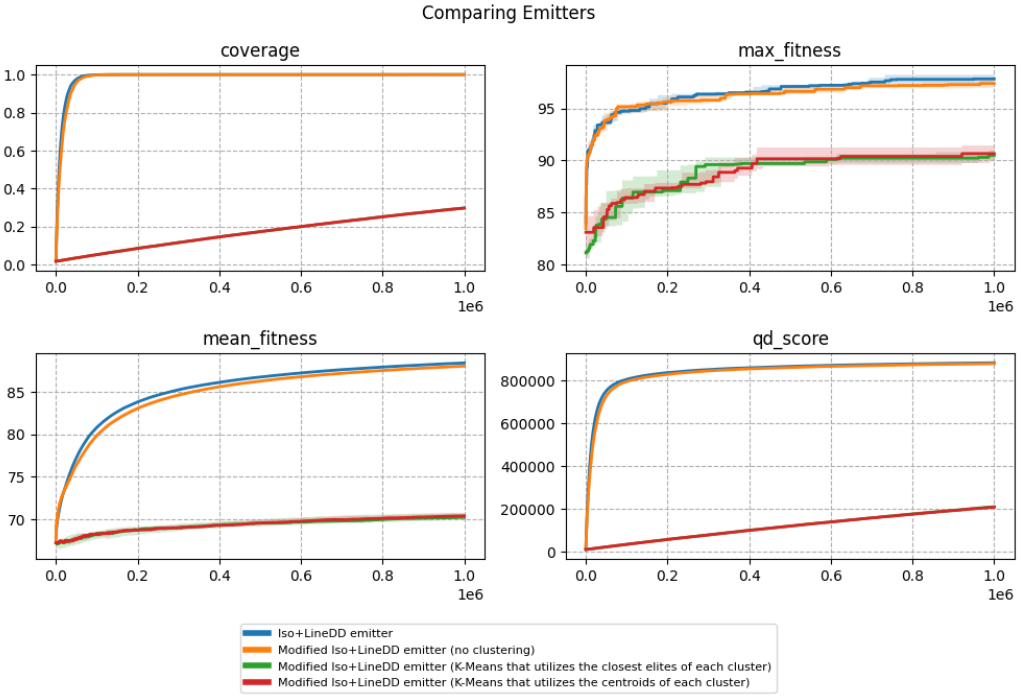
Number of clusters: 2^4



Number of clusters: 2^6

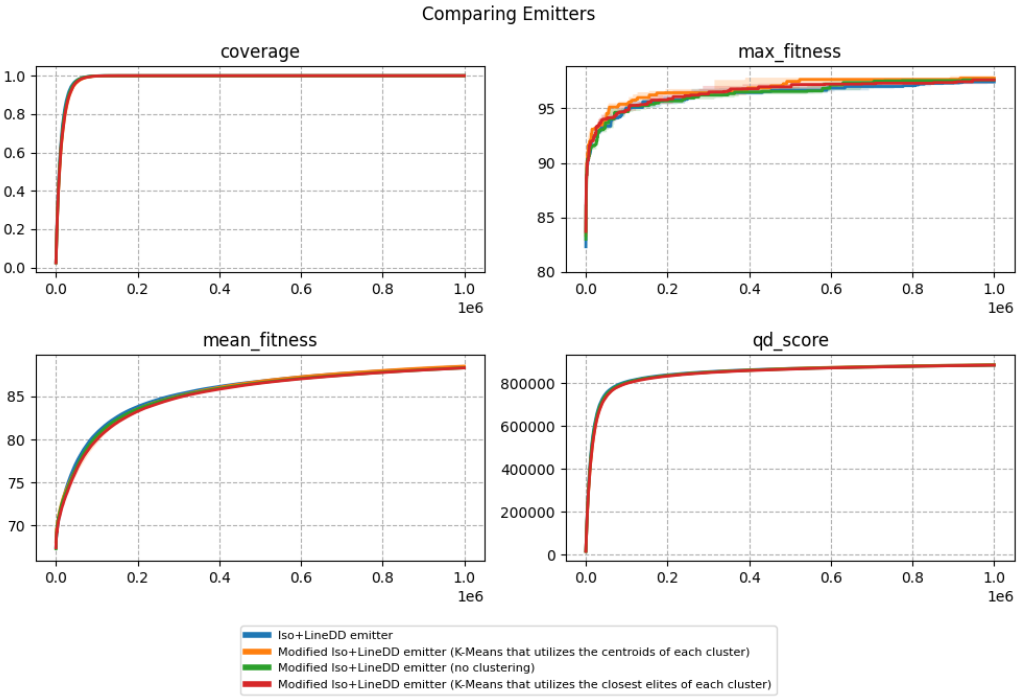


Number of clusters: 2^8

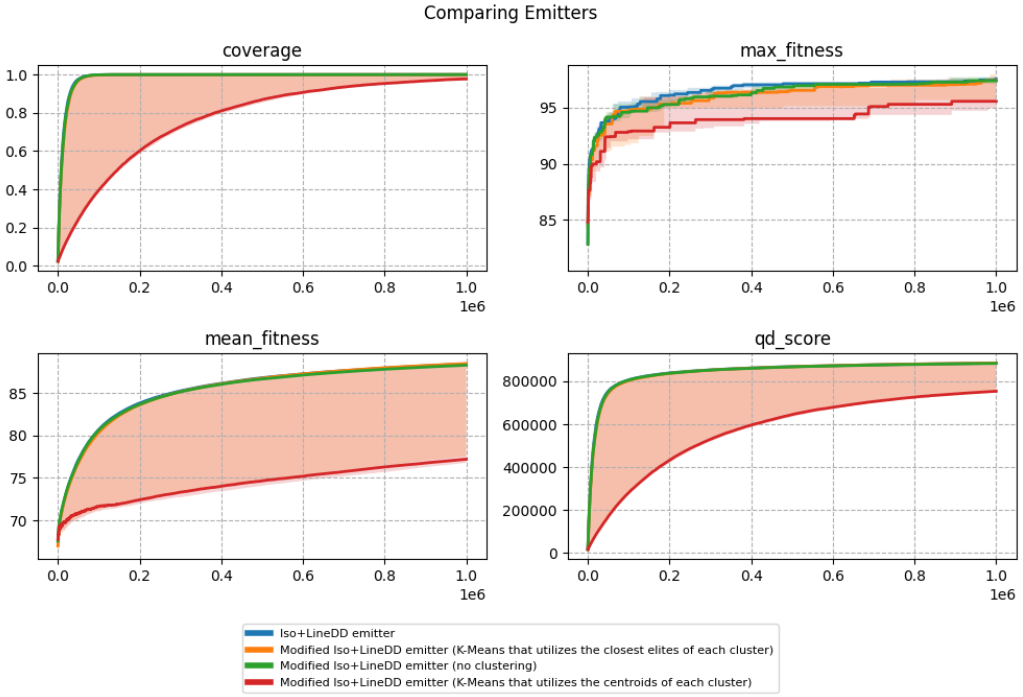


Scenario 6B: 10-D Rastrigin Function with 2^4 Elites

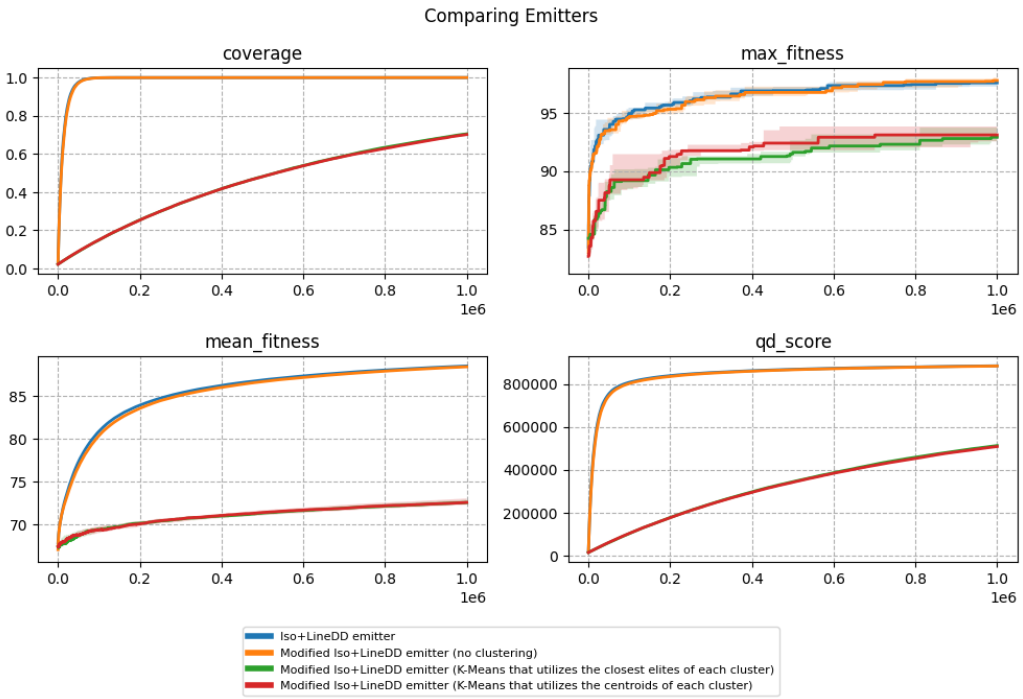
Number of clusters: 2^2



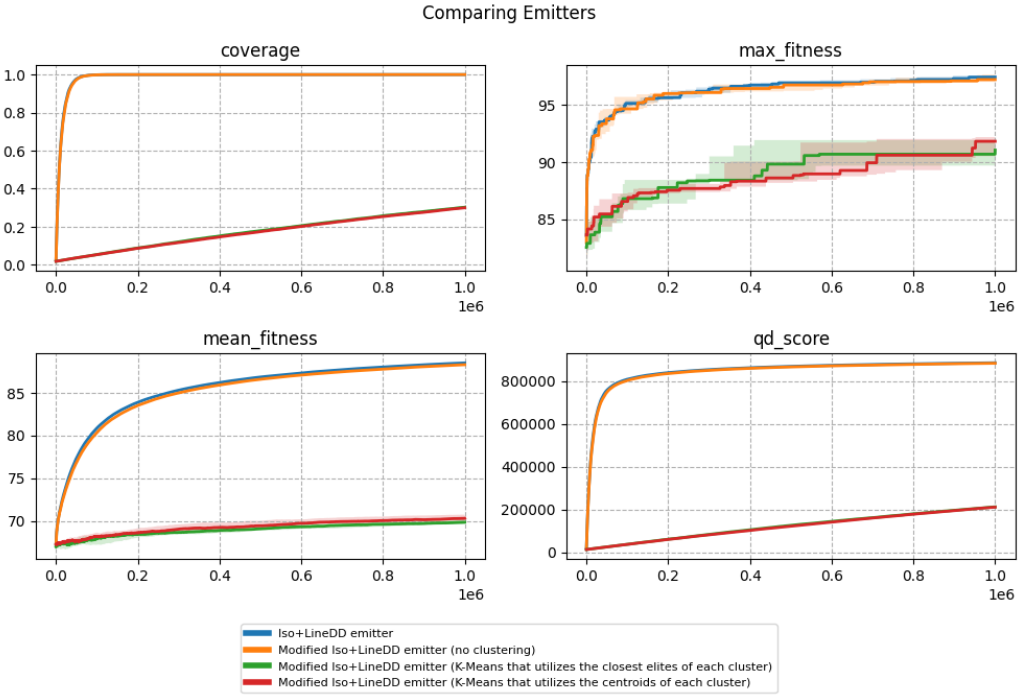
Number of clusters: 2^4



Number of clusters: 2^6

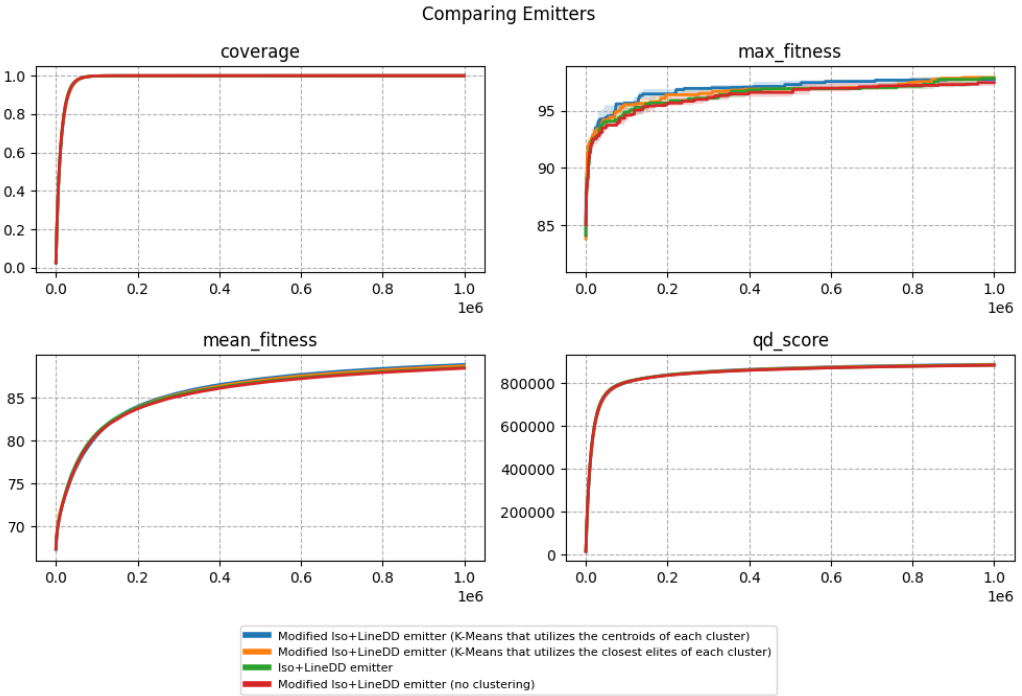


Number of clusters: 2^8

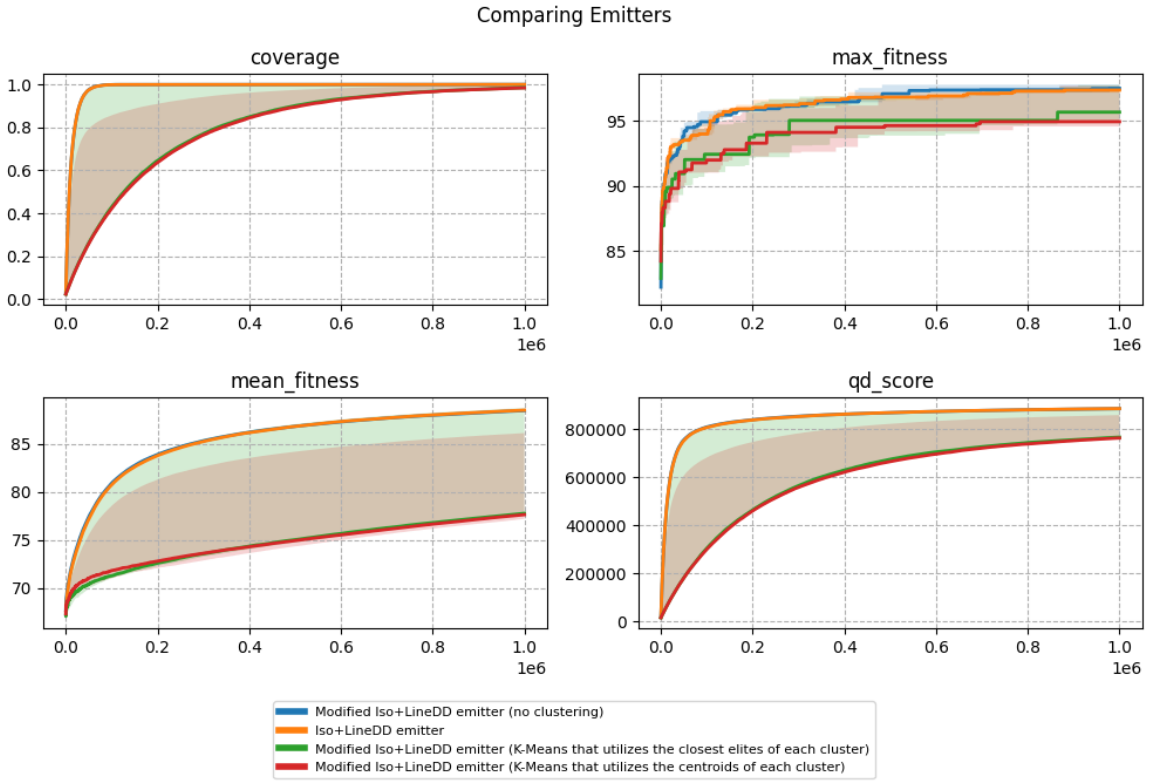


Scenario 6C: 10-D Rastrigin Function with 2^6 Elites

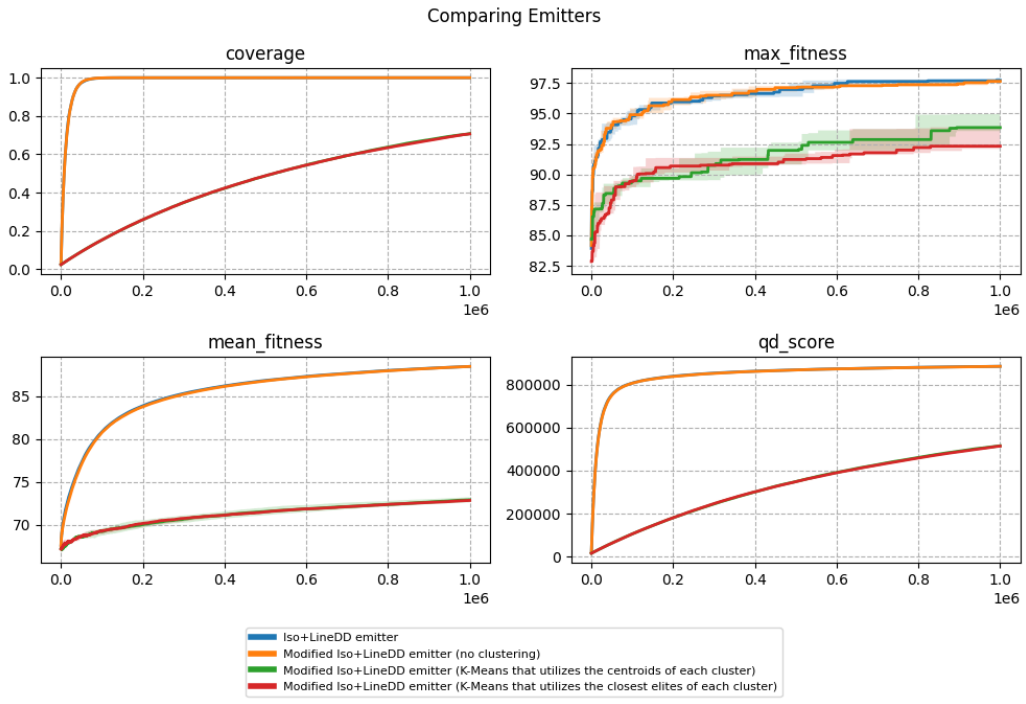
Number of clusters: 2^2



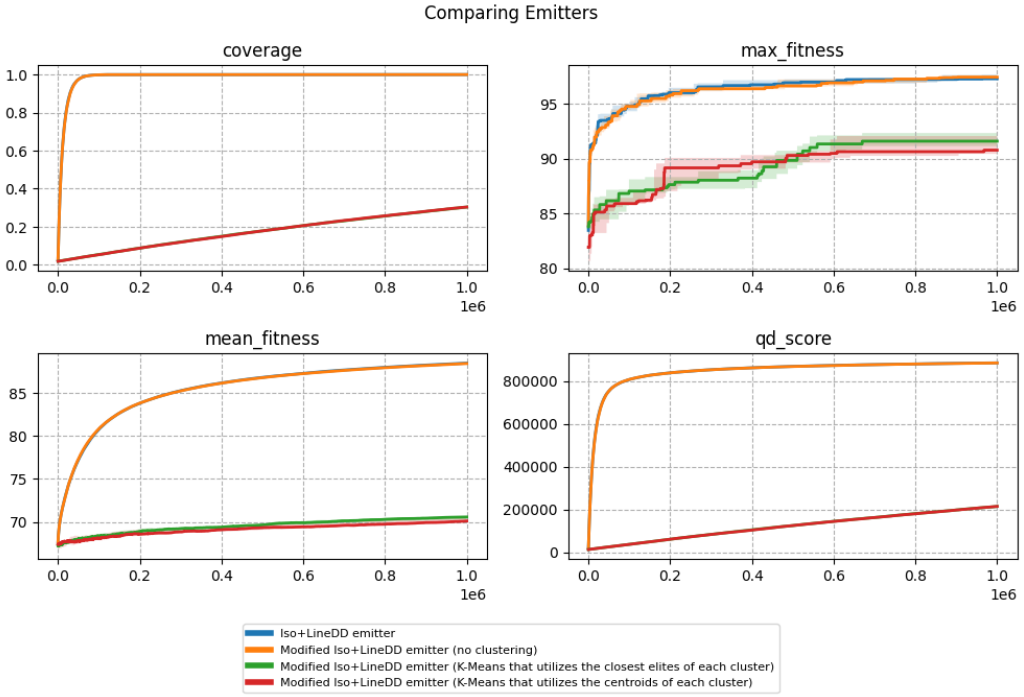
Number of clusters: 2^4



Number of clusters: 2^6



Number of clusters: 2^8



4.3.2. Scenario 7 - 8 with 100x100 Grid Size for 100-D Problem Domains

Both Scenario 7 and 8 with 100x100 grid size for 100-D problem domains, offered additional insights into the performance, scalability, and robustness of the strategies.

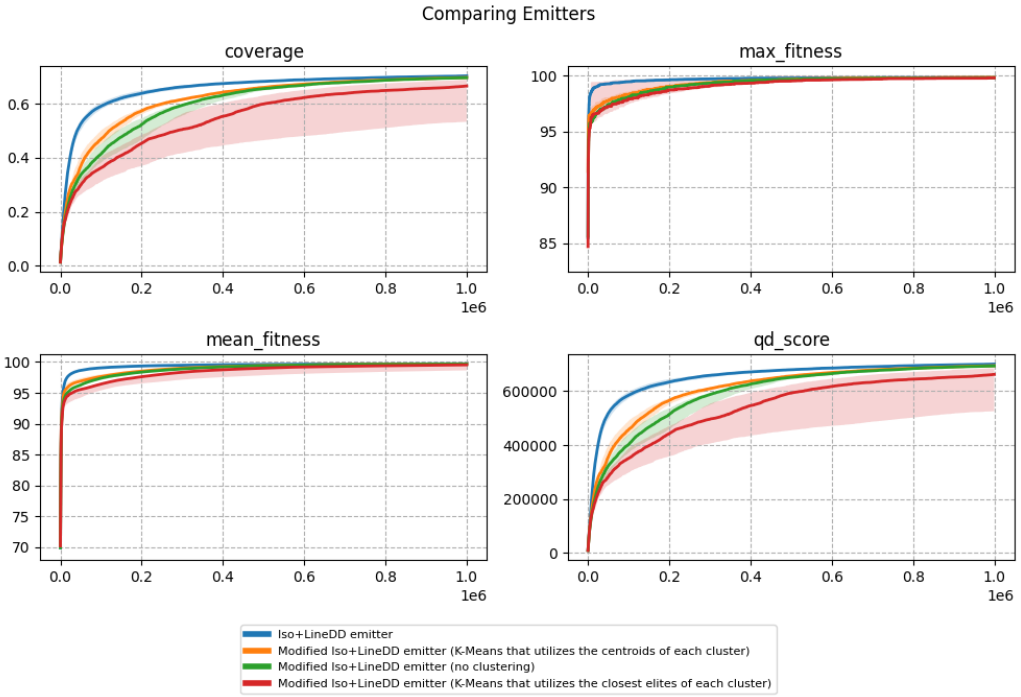
From the plots, it is evident that the impact of K-Means becomes more pronounced, even for the small numbers of clusters. **The performance of the Iso+LineDD emitter was matched in most of the cases by the proposed strategies for Scenario 7 across all the metrics**. In particular, **Strategy 2 was substantially more effective in all cases**, as it converged to the highest scores faster, and in some cases achieved the highest scores. Interestingly, **for Scenario 7C, Strategy 2 managed to slightly outperform the Iso+LineDD emitter**, scoring a higher QD score. This can be attributed to the fact that Strategy 2 may be better at investigating and exploiting the various sub regions of this more complex search space, resulting in a more diverse set of high-performing solutions. Even though the difference is minimal, the slightly greater QD score is an important improvement when compared to the previous experiments. **For the small number of clusters in Scenario 8, the proposed strategies had similar behavior to the Iso+LineDD emitter, although the latter performed better**. This might suggest that the K-Means clustering did not have a significant impact on these small numbers of clusters for the 100-D Rastrigin problem.

Another noteworthy observation on the results is that, **for a larger number of clusters in Scenario 7, the emitter that follows Strategy 1 appears to achieve maximum fitness quicker than the iso+LineDD emitter from the very start**. However, this was not the case for the emitter employing strategy 2, as it was unable to locate the same maximum fitness as the other emitters and instead discovered a lower value. This was not the case for Scenario 8 as **the proposed strategies found lower max fitness value than the Iso+LineDD emitter**. It should be noted that between the 2 strategies, Strategy 2 managed to find higher max fitness.

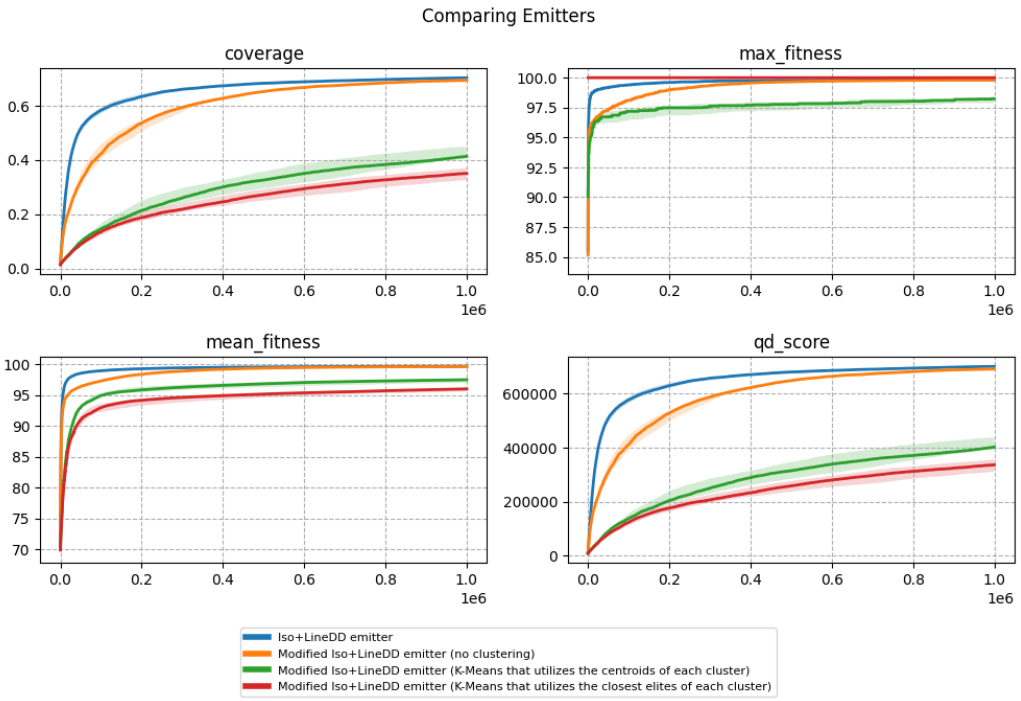
Regarding the other metrics for larger numbers of clusters, in Scenario 7**, the mean fitness of the proposed strategies is lower than the Iso+LineDD emitter but remains relatively high for all numbers of clusters**. The coverage and QD score do not remain high in Scenario 7; as the number of clusters increases, the coverage and QD score decrease. For Scenario 8, the proposed strategies do not manage to match the scores of the Iso+LineDD emitter**. In contrast to Strategy 7, the proposed strategies achieve a relatively high coverage and QD score, but the mean fitness is much lower.** This could indicate that the proposed strategies are finding a diverse set of solutions, but these solutions are not as optimal as those found by the Iso+LineDD emitter. Additionally, Strategy 2 still outperforms Strategy 1 in this case.

Scenario 7A: 100-D Robotic Arm with 2^2 Elites

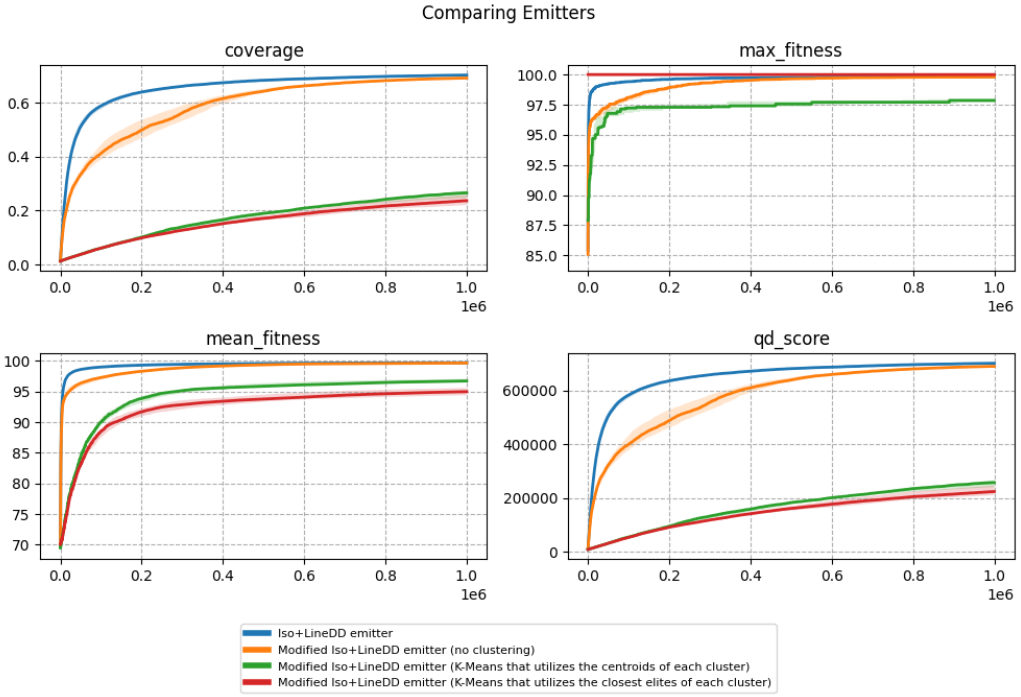
Number of clusters: 2^2



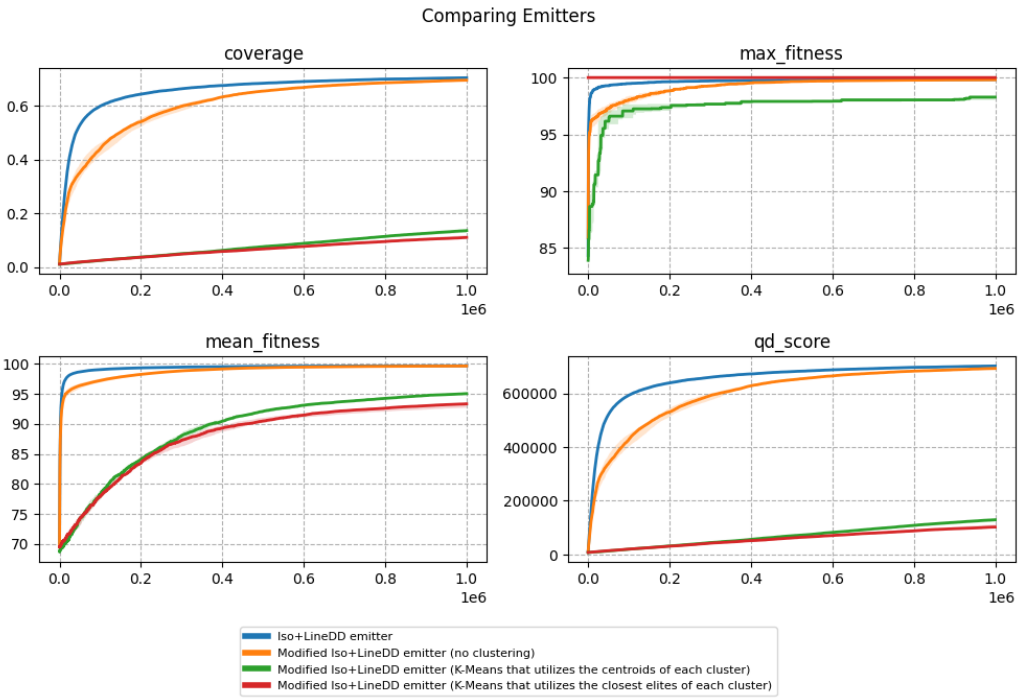
Number of clusters: 2^4



Number of clusters: 2^6

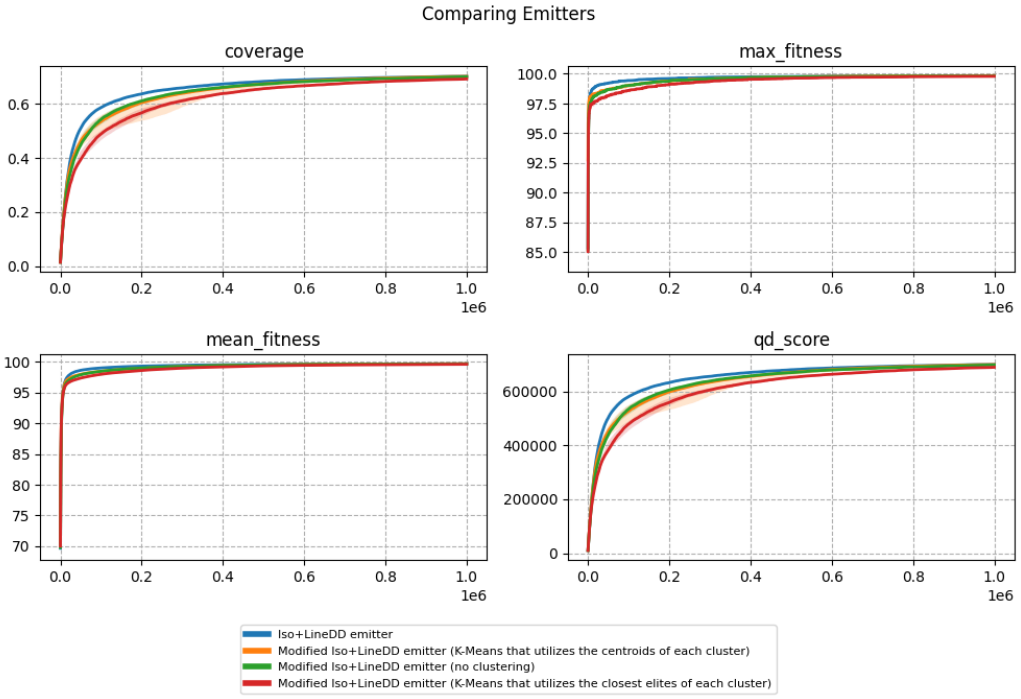


Number of clusters: 2^8

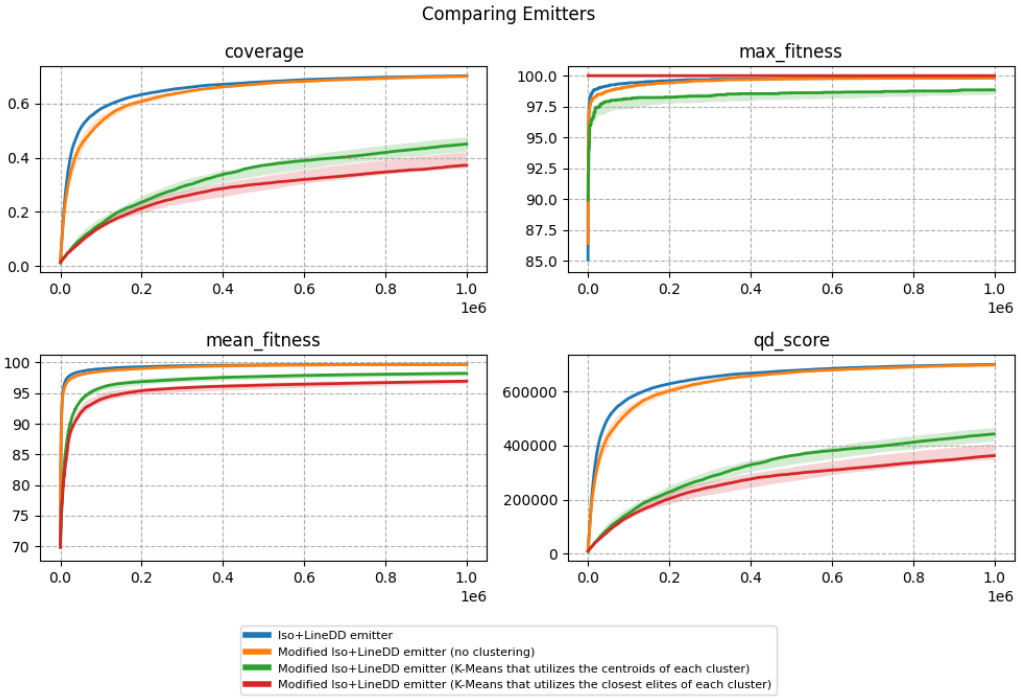


Scenario 7B: 100-D Robotic Arm with 2^4 Elites

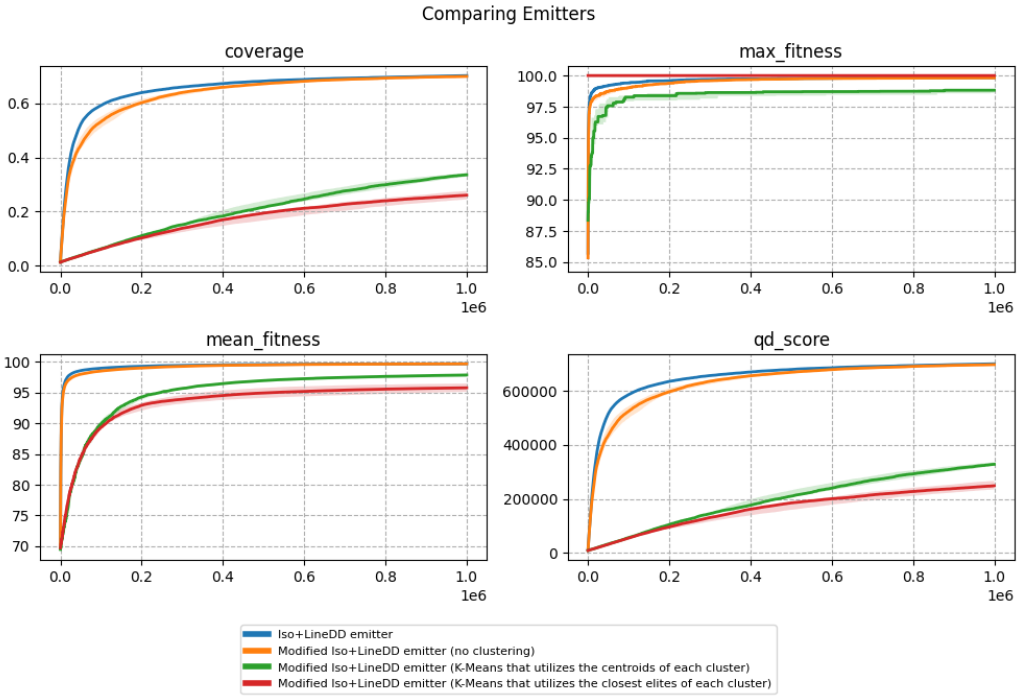
Number of clusters: 2^2



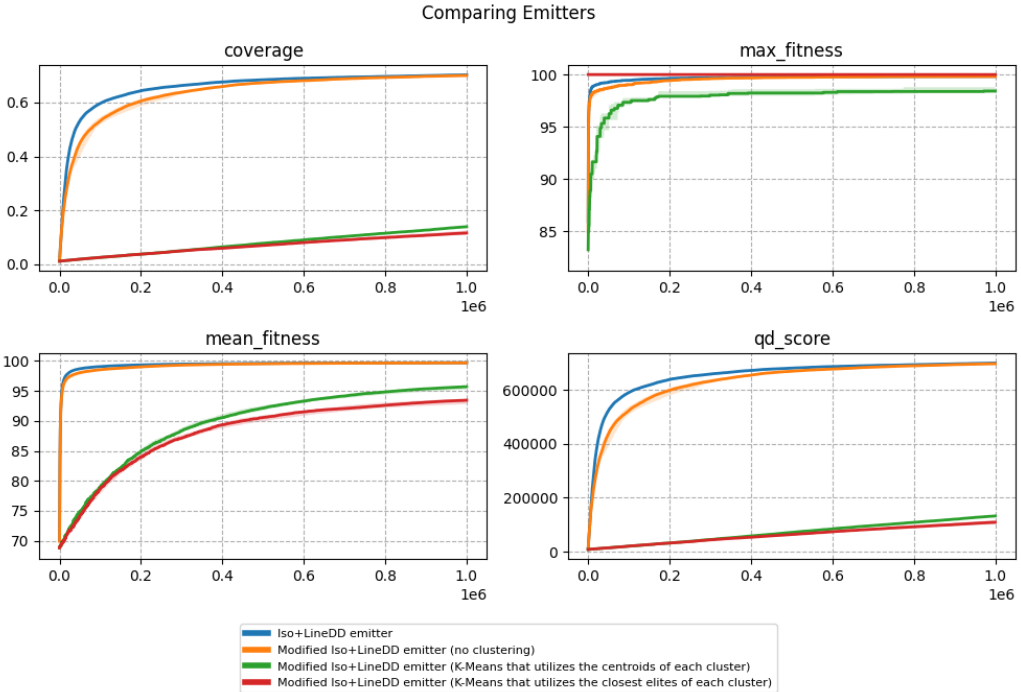
Number of clusters: 2^4



Number of clusters: 2^6

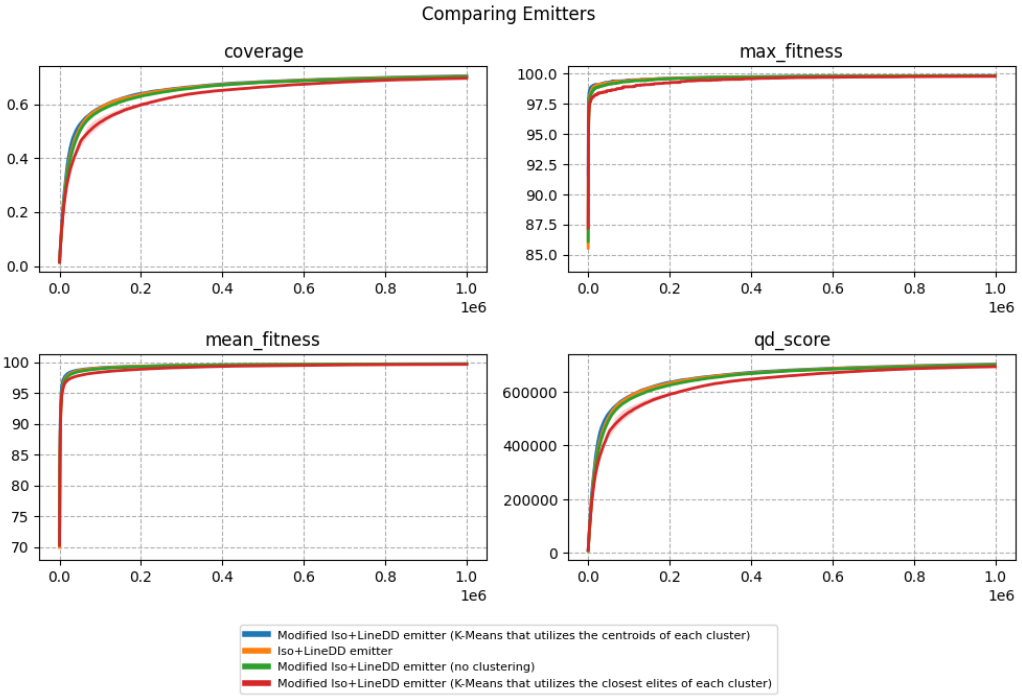


Number of clusters: 2^8

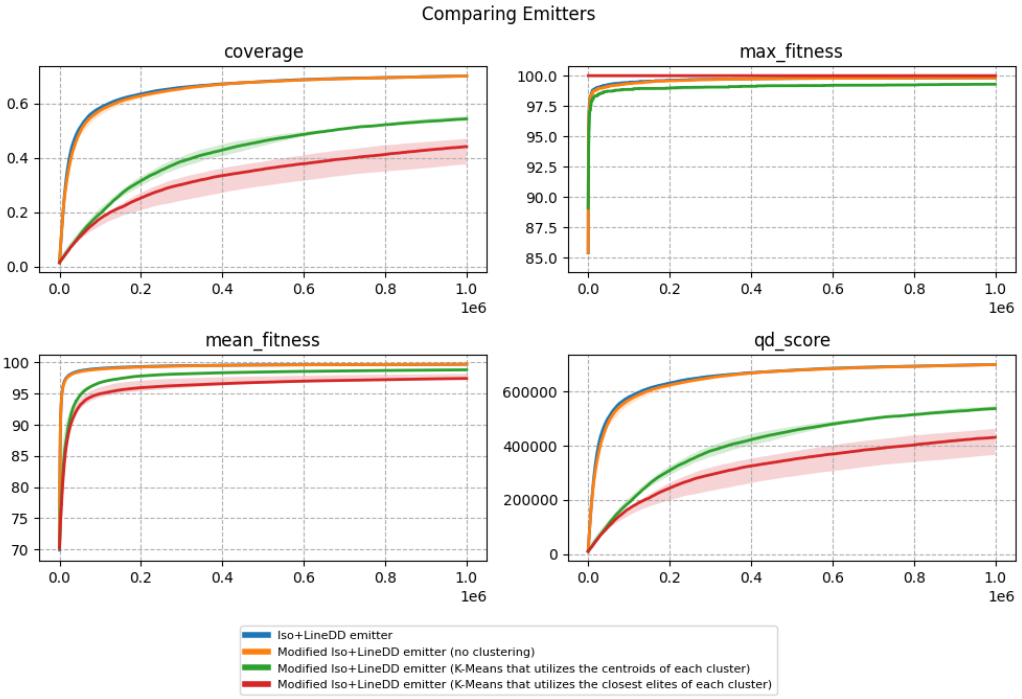


Scenario 7C: 100-D Robotic Arm with 2^6 Elites

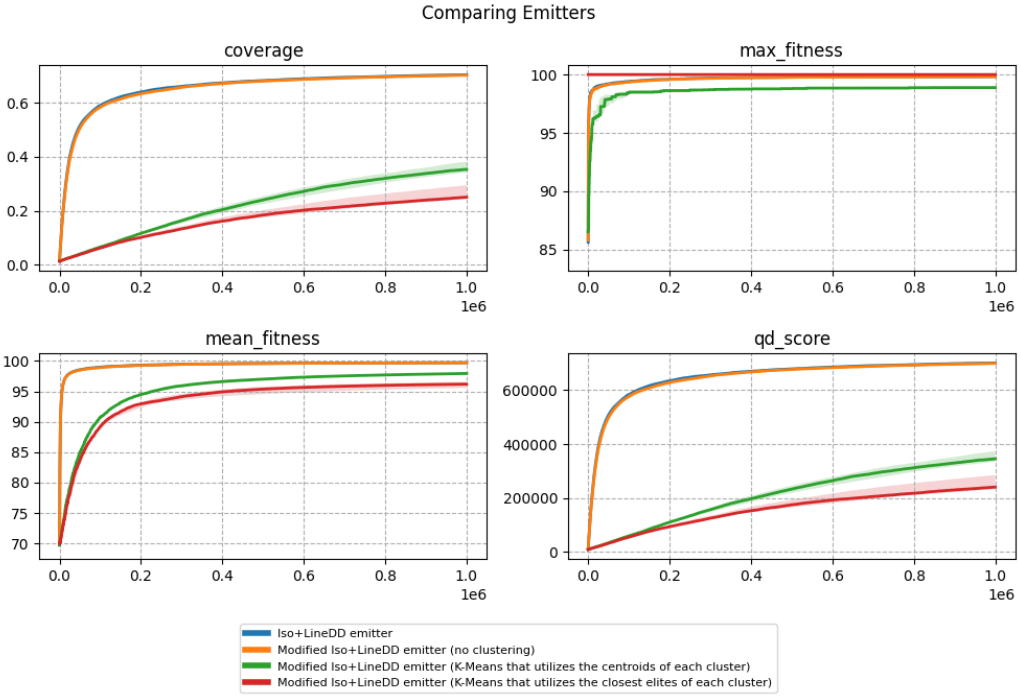
Number of clusters: 2^2



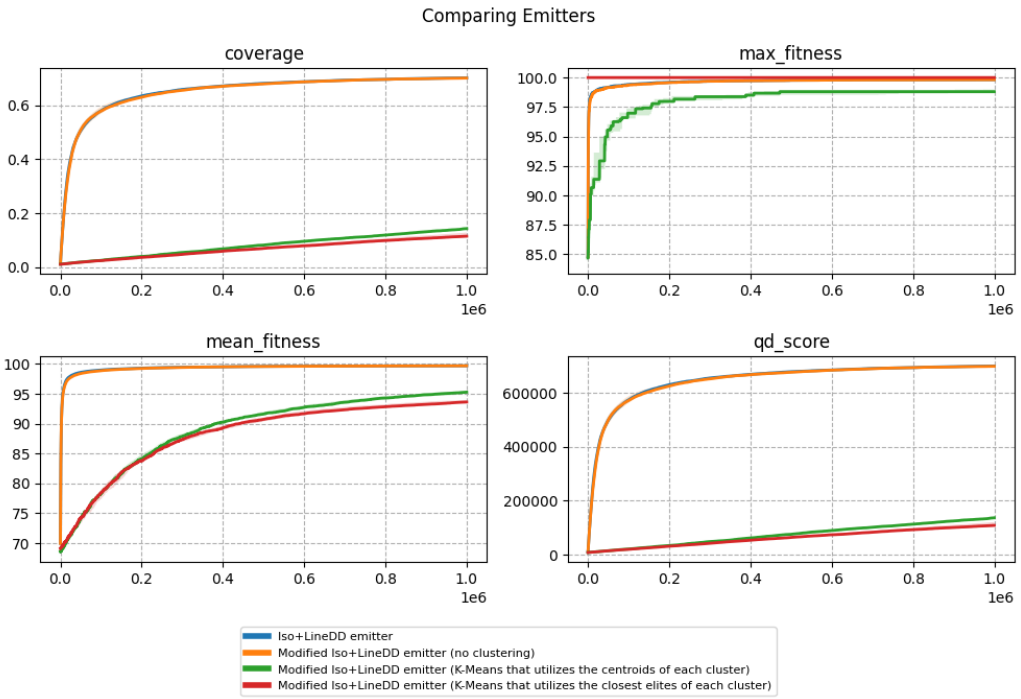
Number of clusters: 2^4



Number of clusters: 2^6

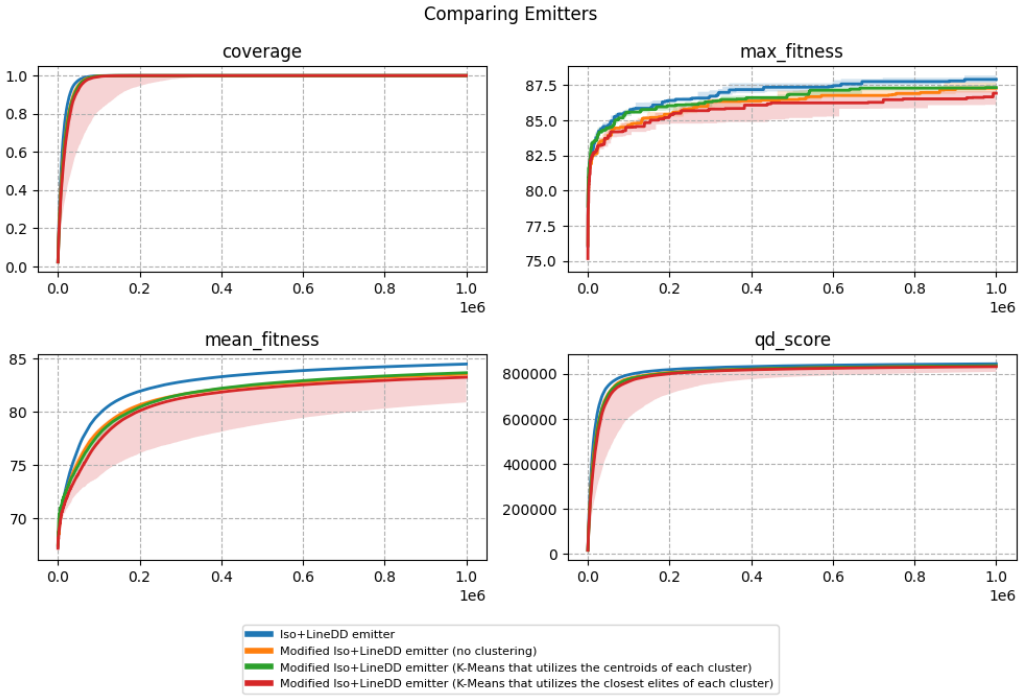


Number of clusters: 2^8

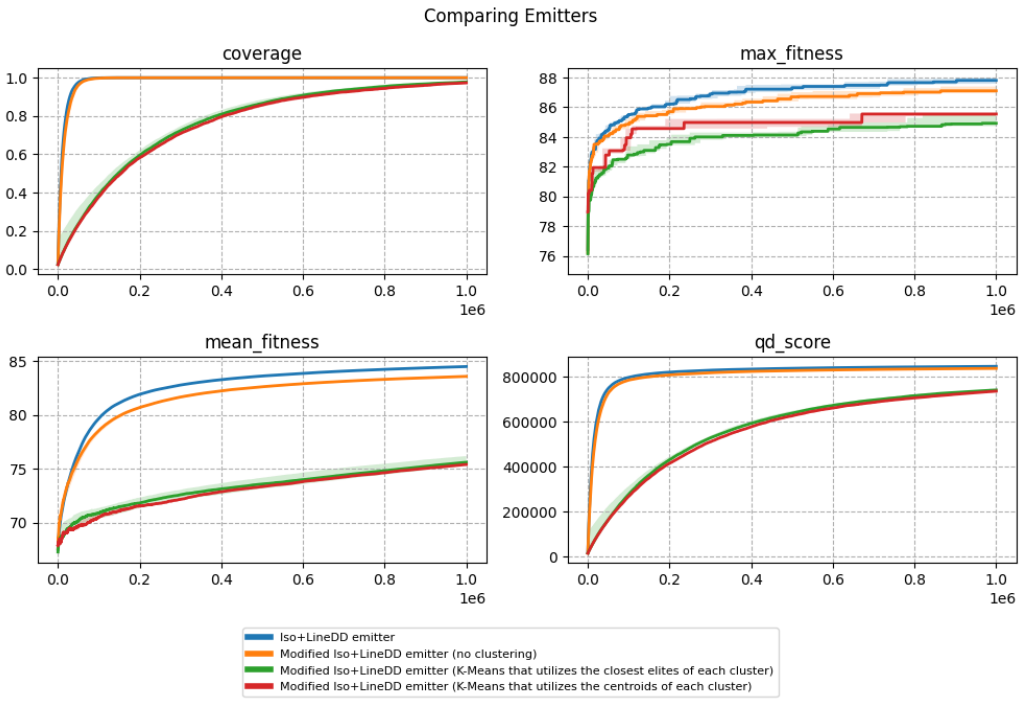


Scenario 8A: 100-D Rastrigin Function with 2^2 Elites

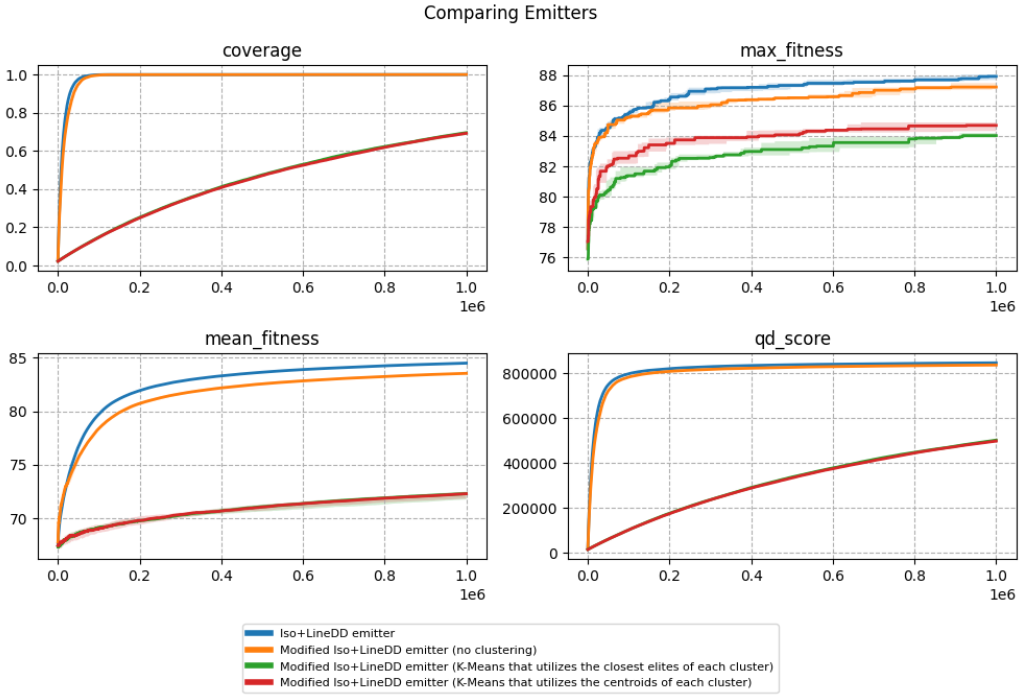
Number of clusters: 2^2



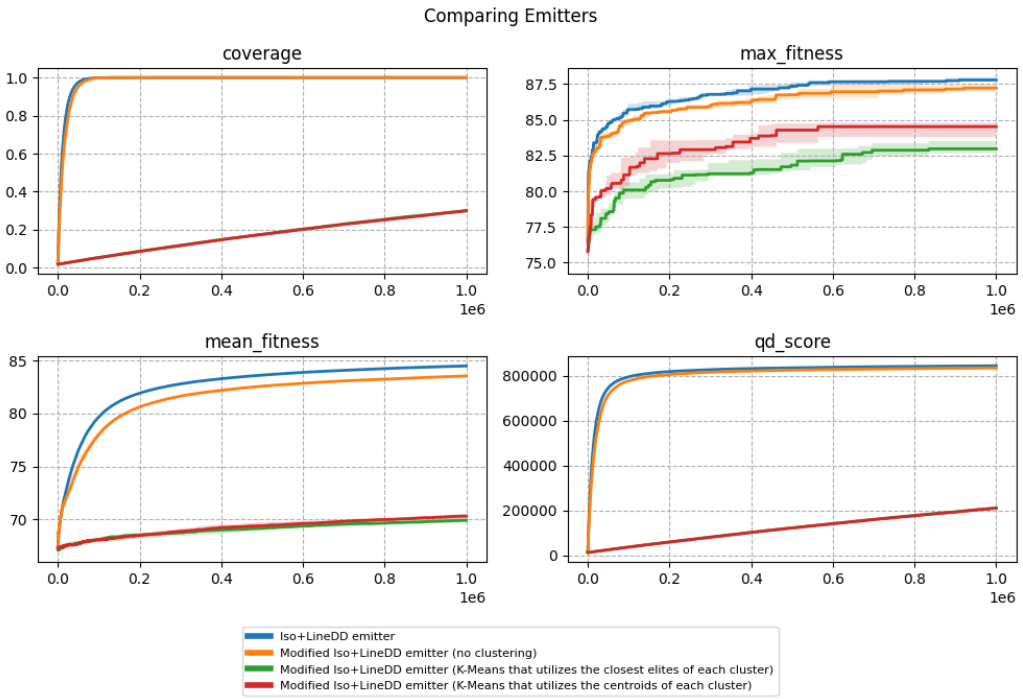
Number of clusters: 2^4



Number of clusters: 2^6

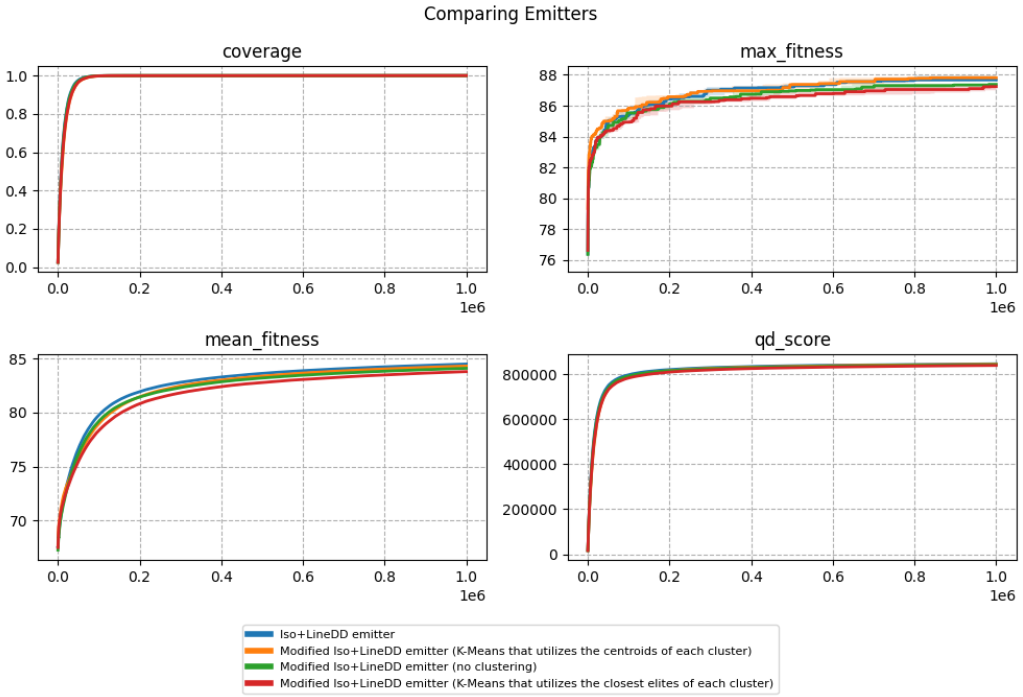


Number of clusters: 2^8

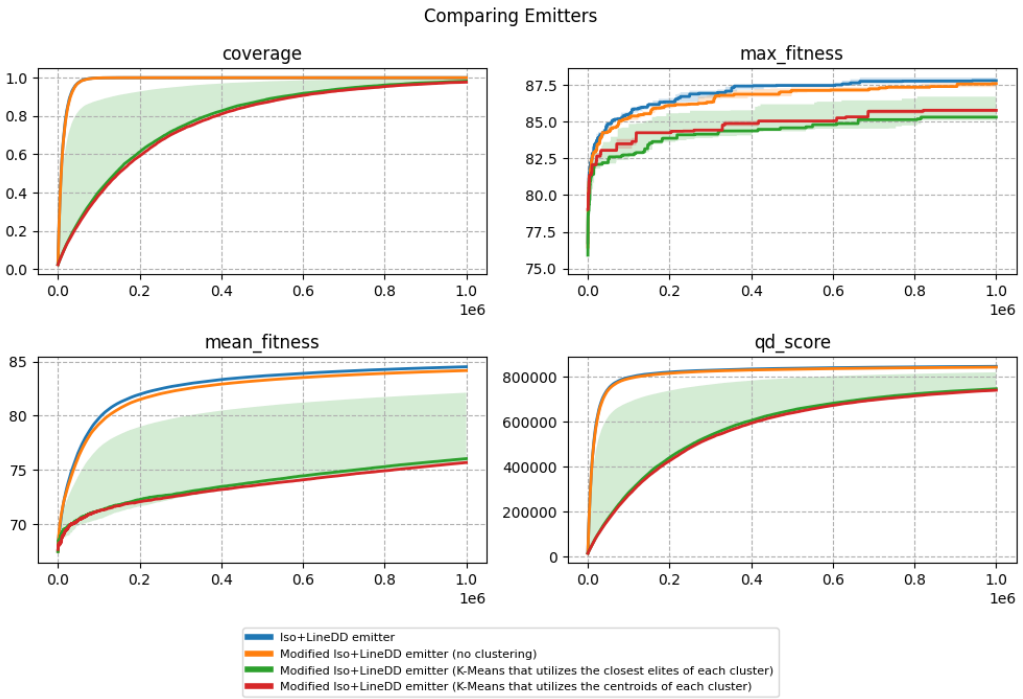


Scenario 8B: 100-D Rastrigin Function with 2^4 Elites

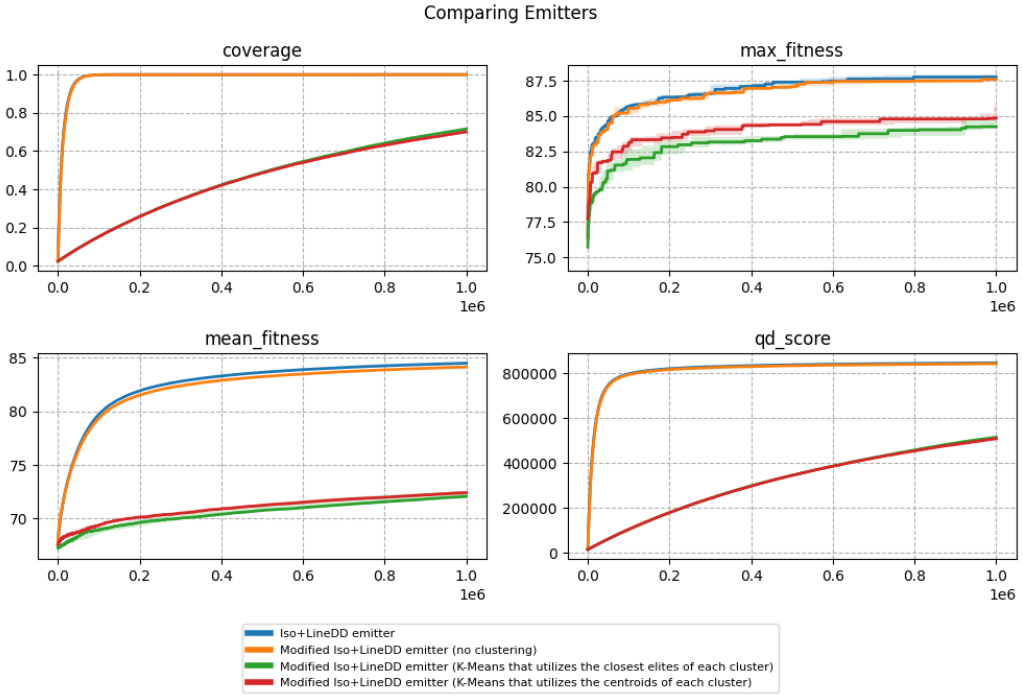
Number of clusters: 2^2



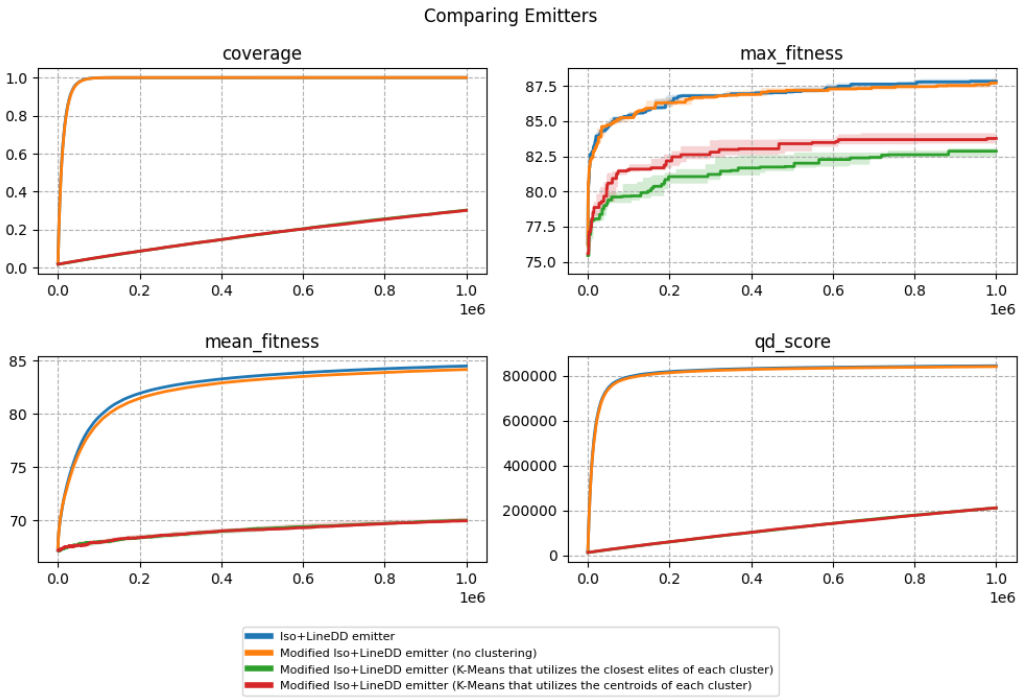
Number of clusters: 2^4



Number of clusters: 2^6

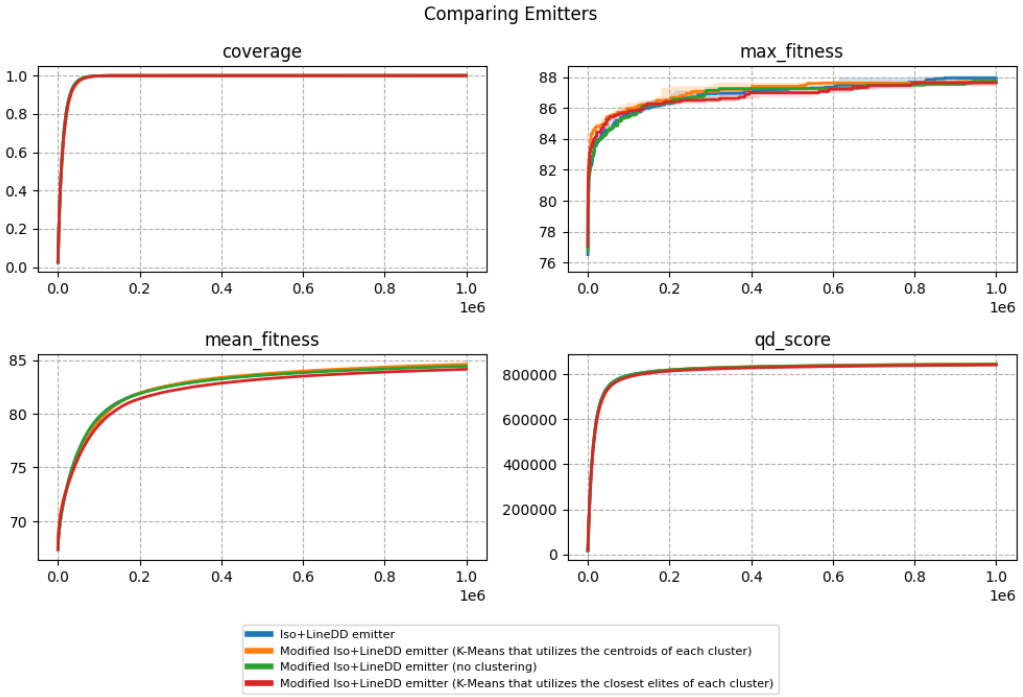


Number of clusters: 2^8

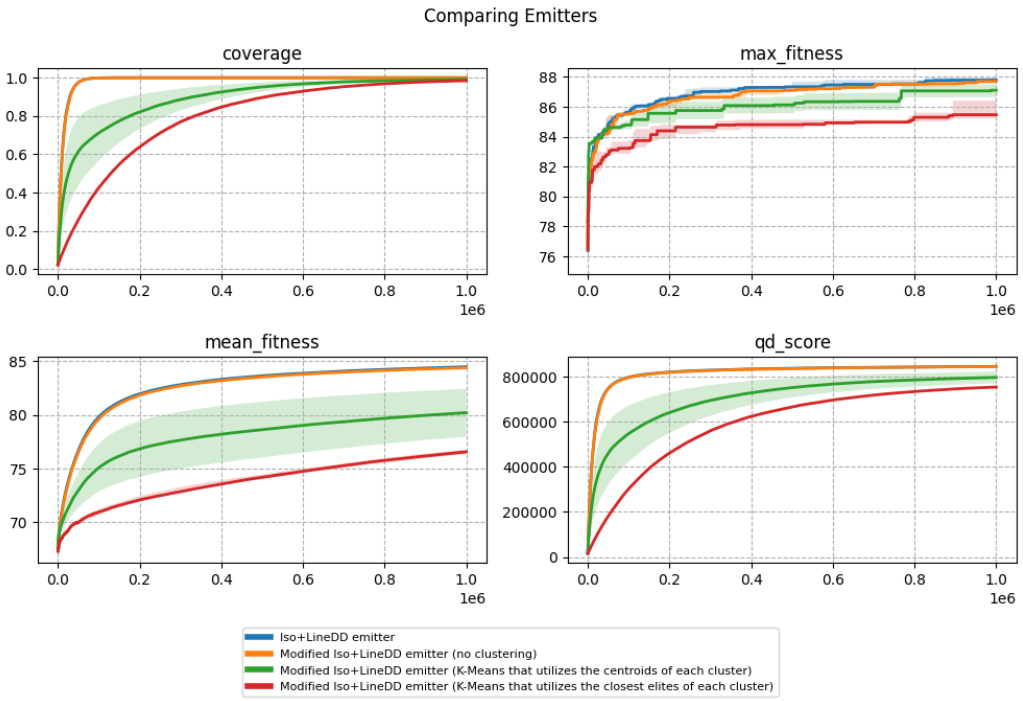


Scenario 8C: 100-D Rastrigin Function with 2^6 Elites

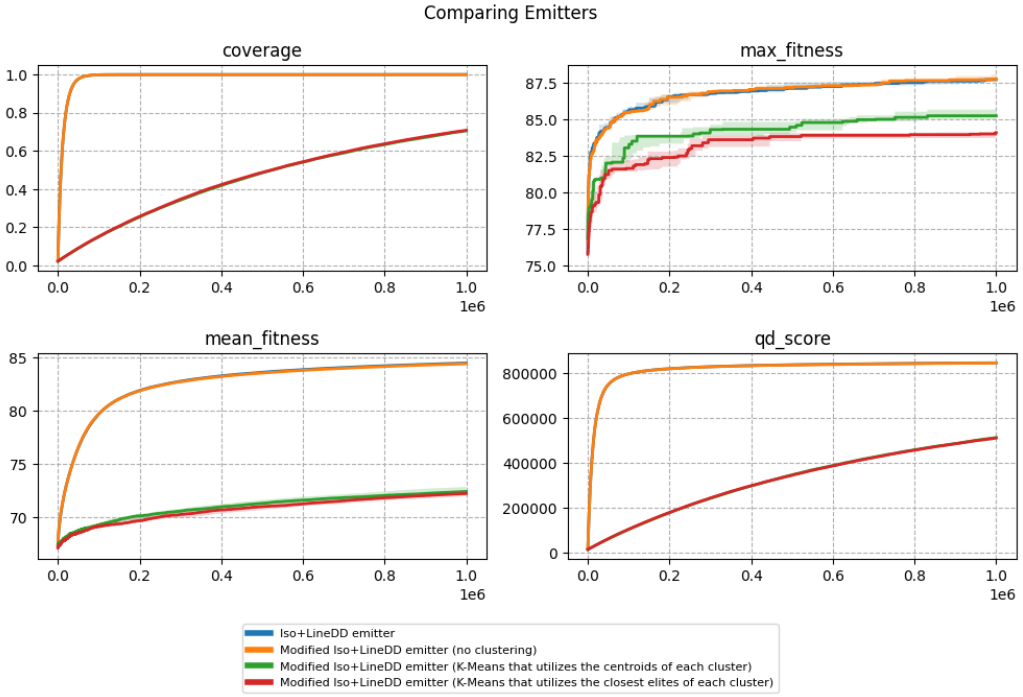
Number of clusters: 2^2



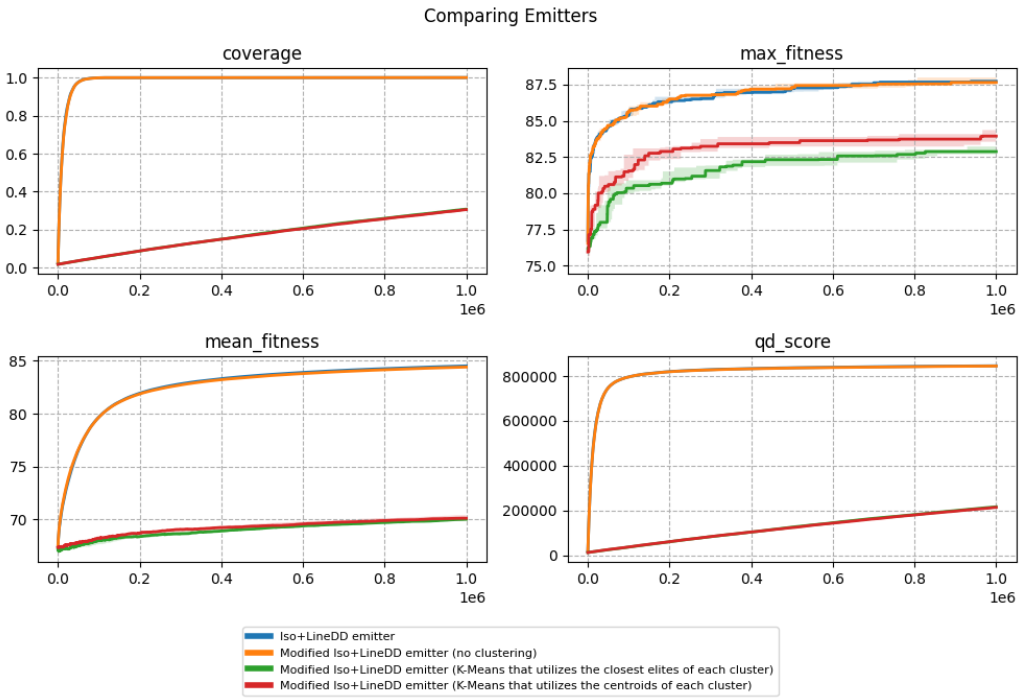
Number of clusters: 2^4



Number of clusters: 2^6



Number of clusters: 2^8



4.4. Main Findings

The diverse range of experimental scenarios across the various problem domains and dimensions, utilizing different grid sizes and running multiple repetitions for each experiment, enabled us to gain an in-depth understanding of the proposed emitters. Their comparison to the Iso+LineDD emitter was important, as the two new strategies followed the emitters' underlying principles while employing different selection strategies for the parent solutions. In this section, we provide a summary of everything we have learned from this research and emphasize the most important findings.

The following are the most important findings:

1. Small numbers of clusters:

   In cases where the number of clusters is small, 4 in our experiments, the performance of the proposed strategies (Strategy 1 and Strategy 2) is similar to that of the Iso+LineDD emitter in the majority of instances. This can be attributed to the limited impact of K-Means clustering in the selection process of the proposed emitters in these cases. However, as the number of clusters increases, the effectiveness of the suggested methodologies tends to decline. This is likely due to the fact that as the number of clusters gets bigger and the effect of K-Means is more pronounced, the proposed strategies focus more on exploiting their respective sub regions or getting "stuck" in local optima, which may result in decreased coverage and a longer convergence time for the mean fitness score.

2. Higher mean fitness score in 2-D Rastrigin function:

   In Scenario 4, the proposed emitters exhibited a higher mean fitness score when compared to the Iso+LineDD emitter on a 2-D Rastrigin Function with a 400x400 grid size. This observation suggests that the proposed strategies may exhibit better performance in specific problem domains and grid sizes, particularly those characterized by numerous local optima and large search spaces. The larger grid size in particular provides the proposed strategies with a more complex search space to navigate. This might enable them to outperform the Iso+LineDD emitter in terms of mean fitness in this specific case.

3. Performance variation depending on batch size and number of clusters:

   We observed that neither of the two proposed strategies consistently demonstrated superior performance on a particular metric or across all metrics when compared to the other. We noticed occasions when Strategy 1 outperformed Strategy 2, but we also saw the opposite. This demonstrates that neither of the two proposed strategies is universally

applicable, and that their selection should be based on the specific problem domain and its characteristics.

4. Challenges in higher-dimensional problem domains:

   For higher-dimensional (10-D and 100-D) problem domains, the proposed strategies struggled to match the success of the Iso+LineDD emitter, particularly as the number of clusters increased. This can be due to the more complex nature of these problem domains and the 100x100 grid size which creates more challenging search spaces for the emitters. It is possible that the proposed emitters' selection process and the K-Means algorithm are also more sensitive to the initial placement of the centroids in the search space, leading to a broader range of performance outcomes.

5. K-Means clustering impact in complex problem domains:

   Overall, we saw that the impact of K-Means generally increased with higher numbers of clusters. This was not the case however for more complex problem domains, like the 100-D Robotic Arm problem, where the influence of clustering algorithm was seen across all the cluster numbers. Regarding the question of whether the introduction of K-Means had a positive or a negative impact in the performance of the emitters, the answer is that generally the algorithms did not improve emitter's performance with the exception of a few instances, like in Scenario 4. Even if in scenarios with higher dimensions such as Scenario 7C, where Strategy 2 outperformed the Iso+LineDD emitter in terms of QD score, K-Means clustering had a negative impact overall on the selection process of the proposed emitters in these more complex search spaces, as they were unable to outperform or even match the Iso+LineDD emitter in terms of convergence rate or overall performance.

Chapter 5
Conclusions and Future Work

5.1. CONCLUSIONS 107
5.2. FUTURE WORK 108

5.1.Conclusions

The proposed strategies have demonstrated potential for enhancing both the exploitation and exploration of the search space in QD optimization problems. However, their performance varies and depends highly on factors such as the number of clusters, problem domain complexity, and the batch sizes in the selection process, which may require additional adjustments for consistent effectiveness.

The impact of the K-Means clustering algorithm became more pronounced with an increased number of clusters in 2-D problem domains, yet it did not consistently improve the performance of the proposed strategies. In fact, introducing K-Means often resulted in the proposed strategies falling short of the Iso+LineDD emitter's performance in terms of score value and convergence time. An exception was observed in the 2-D Rastrigin function with a larger grid size, where the proposed strategies outperformed the Iso+LineDD emitter in mean fitness, suggesting potential advantages in complex search spaces with numerous local optima. However, this advantage was not observed everywhere, highlighting the need for additional research.

In terms of their robustness and scalability, the proposed strategies performed inconsistently in higher-dimensional problem domains, such as the 10-D and 100-D problem domains. Interestingly, the impact of K-Means clustering was observed even with a smaller number of clusters in these higher-dimensional domains. Despite this, the proposed strategies generally struggled to match the performance of the Iso+LineDD emitter, highlighting the difficulty of successfully applying these strategies to search spaces with an increased level of complexity.

The varying performance of the strategies across different metrics and scenarios highlights the need for changes in the selection process of the proposed strategies in order to consistently improve their performance across a broader range of problem domains. By addressing all of the shortcomings brought up in this study, the proposed strategies have the potential to significantly improve the selection process of Iso+LineDD and other emitters, thereby leading to the advancement of QD optimization algorithms that are more adaptable and efficient, potentially benefiting a wide range of applications.

5.2.Future Work

This study opens two major opportunities for future research, one concentrating on computational efficiency and the other on improving the performance of the presented approaches.

1. Enhancing computational efficiency: Due to its adaptability and flexibility for specific experimental needs, both of the proposed emitters were implemented on the QDJAX library. However, the necessary setup for the use of GPU acceleration was not completed for these experiments, preventing its use and resulting in longer experiment execution durations. Future work could involve the implementation of the proposed strategies in additional libraries, such as QDAX. Because QDAX also makes use of hardware accelerators such as GPUs/TPUs for massive parallelism and drastically reducing the runtime of QD algorithms [38], using such libraries could enable faster experimentation and more extensive testing across a wider range of problem domains, cluster numbers, and parameter settings.
2. Improving strategies' performance: The K-Means clustering algorithm did not consistently enhance the performance of the proposed strategies in this research. A promising direction for future work could involve adapting the selection process of emitters, particularly with regard to the initialization of centroids in the K-Means algorithm. Currently, the initial centroids are randomly selected; however, modifying this to a selection process that is more biased towards diverse and high-performing elites could potentially result in significant improvements. The initialization of centroids is critical to the algorithm's performance; therefore, this change could result in different and potentially more successful outcomes. Furthermore, in our research, K-Means clustering was applied for the selection of the second batch B2 of elites. A novel direction for future research could involve applying the same clustering technique for the selection of the first batch B1 of elites instead. This adjustment could provide a different perspective on how K-Means clustering impacts the overall performance of our proposed strategies. Additionally, the integration of alternative clustering techniques, such as Hierarchical clustering or DBSCAN, could also be explored.

In conclusion, the future work outlined above represents promising avenues for continuing this research. As we continue improving these strategies, it is hoped that their full potential will be realized, leading to substantial improvements in the optimization process across diverse application domains.

References

[1] J. Lehman and K. O. Stanley, "Exploiting open-endedness to solve problems through the search for novelty," in Alife, 2008.

[2] J. Mouret and J. Clune, "Illuminating search spaces by mapping elites," *arXiv Preprint arXiv:1504.04909*, 2015.

[3] J. K. Pugh, L. B. Soros and K. O. Stanley, "Quality diversity: A new frontier for evolutionary computation," *Frontiers in Robotics and AI*, vol.3, Article No. 40, 2016.

[4] V. Vassiliades and J. Mouret, "Discovering the Elite Hypervolume by Leveraging Interspecies Correlation," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2018, pp. 149-156.

[5] M. C. Fontaine and S. Nikolaidis, "Differentiable Quality Diversity," arXiv. [Online]. Available: https://arxiv.org/abs/2106.03894.

[6] V. Vassiliades, K. Chatzilygeroudis and J. Mouret, "Using centroidal voronoi tessellations to scale up the multidimensional archive of phenotypic elites algorithm," IEEE Transactions on Evolutionary Computation, vol. 22, no. 4, pp. 623-630, 2017.

[7] V. Vassiliades, K. Chatzilygeroudis and J. Mouret, "A comparison of illumination algorithms in unbounded spaces," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 2017, pp. 1578–1581.

[8] S. Edition and K. H. Rosen, "Discrete Mathematics and Its Applications," 7th ed., vol. 1. New York, NY: McGraw-Hill, 2012.

[9] S. P. Boyd and L. Vandenberghe, Convex Optimization. Cambridge, UK: Cambridge University Press, 2004.

[10] L. A. Rastrigin, "Systems of extremal control," Nauka, Moscow, 1974.

[11] "Rastrigin function," Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Rastrigin_function. [Accessed: April 09, 2023].

[12] "Pythagorean theorem," Encyclopedia of Mathematics. [Online]. Available: http://encyclopediaofmath.org/index.php?title=Pythagorean_theorem&oldid=40035. [Accessed: April 09, 2023].

[13] "Euclidean Distance," Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Euclidean_distance. [Accessed: April 09, 2023].

[14] A. A. Bennett, "Theory of probability," Electrical Engineering, vol. 52, (11), pp. 752-757, 1933.

[15] "Random Variable," Encyclopedia of Mathematics. [Online]. Available: https://encyclopediaofmath.org/index.php?title=Random_variable&oldid=43639. [Accessed: Mar. 30, 2023].

[16] "Uniform Distribution," Encyclopedia of Mathematics. [Online]. Available: https://encyclopediaofmath.org/wiki/Uniform_distribution. [Accessed: March 30, 2023].

[17] "Continuous Uniform Distribution," Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Continuous_uniform_distribution. [Accessed: April 09, 2023].

[18] "Normal Distribution," Encyclopedia of Mathematics. [Online]. Available: https://encyclopediaofmath.org/wiki/Normal_distribution. [Accessed: March 30, 2023].

[19] X. Yu and M. Gen, *Introduction to Evolutionary Algorithms.* Springer: London, 2010.

[20] "Introduction to Evolutionary Algorithms: Genetic Algorithm," Brainyloop. [Online]. Available: https://brainyloop.com/introduction-to-evolutionary-algorithms-genetic-algorithm-neuro-evolution/. [Accessed: March 31, 2023].

[21] M. Preuss, "Introduction: Towards Multimodal Optimization," in *Multimodal Optimization by Means of Evolutionary Algorithms*, M. Preuss, Ed., Cham, Switzerland: Springer, 2015, pp. 1-25.

[22] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence.* MIT Press, 1992.

[23] D. Soni, "Introduction to Evolutionary Algorithms," Towards Data Science. [Online]. Available: https://towardsdatascience.com/introduction-to-evolutionary-algorithms-a8594b484ac. [Accessed: April 10, 2023].

[24] R. de Silva, "Introduction to Clustering Methods," Medium. [Online]. Available: https://medium.com/@ramindu2797/introduction-to-clustering-methods-9ecc7041aba1. [Accessed: April 08, 2023].

[25] P. Patel, "K-Means Clustering Algorithm," Medium. [Online]. Available: https://medium.com/@imparth/k-means-clustering-algorithm-34807a7cec71. [Accessed: April 10, 2023].

[26] M. Dogra, "K-means Clustering Algorithm: Explained and Implemented," Medium. [Online]. Available: https://immohann.medium.com/k-means-clustering-algorithm-explained-and-implemented-307161adcc61. [Accessed: April 08, 2023].

[27] S. Zhao, "K-means Clustering Clearly Explained," Medium. [Online]. Available: https://medium.com/@luo9137/k-means-clustering-clearly-explained-44746ccc3621. [Accessed: April 08, 2023].

[28] A. Cully and Y. Demiris, "Quality and diversity optimization: A unifying modular framework," *IEEE Transactions on Evolutionary Computation*, vol. 22, no. 2, pp. 245-259, 2017.

[29] K. Chatzilygeroudis *et al*, "Quality-diversity optimization: A novel branch of stochastic optimization," in *Black Box Optimization, Machine Learning, and no-Free Lunch Theorems*, Springer, P. M. Pardalos, V. Rasskazova, and M. N. Vrahatis, Eds., Springer: Cham, 2021, pp. 109–135.

[30] V. Pariza, "Fast Learning of Diverse Robotic Skills," B.S. Thesis, Dept. Computer Science, University of Cyprus, Cyprus, 2022.

[31] J. Mouret, "Novelty-based multiobjectivization," in *New Horizons in Evolutionary Robotics: Extended Contributions from the 2009 EvoDeRob Workshop*, 2011.

[32] J. Lehman and K. O. Stanley, "Evolving a diversity of virtual creatures through novelty search and local competition," in *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation* (GECCO '11), N. Krasnogor and P. L. Lanzi, Eds., Dublin, Ireland, July 12-16, 2011, pp. 211–218. ACM, 2011.

[33] J. H. Friedman, J. L. Bentley and R. A. Finkel, "An algorithm for finding best matches in logarithmic expected time," *ACM Transactions on Mathematical Software (TOMS)*, vol. 3, *(3)*, pp. 209-226, 1977.

[34] J. Clune, J. Mouret and H. Lipson, "The evolutionary origins of modularity," *Proceedings of the Royal Society B: Biological Sciences*, vol. 280, no. 1755, pp. 20122863, 2013.

[35] A. Gaier, A. Asteroth, and J.-B. Mouret, "Aerodynamic Design Exploration through Surrogate-Assisted Illumination," in *18th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Denver, CO, USA, June 5-9, 2017. doi: 10.2514/6.2017-3330.

[36] N. Justesen, S. Risi, and J.-B. Mouret, "MAP-Elites for noisy domains by adaptive sampling," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, Prague, Czech Republic, July 13-17, 2019, pp. 121-122.

[37] M. Flageat *et al*, "Benchmarking Quality-Diversity Algorithms on Neuroevolution for Reinforcement Learning," *arXiv Preprint arXiv:2211.02193*, 2022.

[38] B. Lim et al., "Accelerated Quality-Diversity through Massive Parallelism," Transactions on Machine Learning Research, 01/2023, 2023.

Apppendix A
Implementations

A.1 Implementation of K-Means Clustering on JAX

This appendix describes JAX library K-Means clustering implementation. As described in section 3.4, the implementation has been modified from https://github.com/creinders/ClusteringAlgorithmsFromScratch. This implementation integrated K-Means clustering with the MAP-Elites Grid Archive in Strategy 1 and Strategy 2. The K-Means clustering algorithm was implemented across three files: **base_algorithm.py**, **kmeans_base.py**, and **kmeans_jax.py**. The following sections outline the contents of each file.

BaseAlgorithm

```
class BaseAlgorithm:

    def __init__(self, callback=None, verbose=False) -> None:
        self.verbose = verbose
        self.callback = callback

    def get_hook(self, name):
        return getattr(self.callback, name, None)

    def call_hook(self, name, *args, **kwargs):
        callback_op = self.get_hook(name)
        if callable(callback_op):
            callback_op(*args, **kwargs)
```

KMeansBase

```
import os
import sys

import numpy as np
from scipy.spatial.distance import cdist

from clustering.kmeans_jax.base_algorithm import BaseAlgorithm


class KMeansBase(BaseAlgorithm):

    def __init__(self,
                 n_clusters,
                 max_iter=100,
                 early_stop_threshold=0.01,
                 seed=None,
                 callback=None,
                 verbose=False
                 ) -> None:
        super().__init__(callback=callback, verbose=verbose)
        self.n_clusters = n_clusters
        self.max_iter = max_iter
        self.early_stop_threshold = early_stop_threshold
        self.rng = np.random.RandomState(seed)
```

```
def init_clusters(self, X, n_clusters):
        n = X.shape[0]

        i = self.rng.permutation(n)[:n_clusters]
        centers = X[i]
        return centers

    def prepare(self, X):
        centers = self.init_clusters(X, self.n_clusters)
        return X, centers

    def _main_loop(self, X, centers):
        pass

    def tensor_to_numpy(self, t):
        return np.array(t)

    def finalize(self, centers, assignments):
        return centers, assignments

    def fit(self, X, method="centroids"):
        X, initial_centers = self.prepare(X)

        self.call_hook('on_main_loop_start')
        centers, assignments = self._main_loop(X, initial_centers)
        self.call_hook('on_main_loop_end')

        centers, assignments = self.finalize(centers, assignments)

        self.call_hook('on_epoch_end', self, X, centers, assignments)

        return centers, assignments
```

KMeansJax

```
import os
import sys

import chex
import jax
import jax.numpy as jnp
from jax import jit
from jax import random

from clustering.kmeans_jax.kmeans_base import KMeansBase


@jit
def cluster_update(cluster_index, old_cluster, assignments, X):
    q = assignments == cluster_index
    mask = q.astype(jnp.int32)
    c = jnp.sum(mask)
    s = jnp.sum(X * mask[:, None], axis=0)
    m = s / c

    return m
```

```
@jit
def step(X, centers):
    cluster_update_vmap = jax.vmap(cluster_update, in_axes=(0, 0, None,
None))

    distance = jnp.sum(jnp.square((X[:, :, None] - jnp.transpose(centers,
(1, 0))[None, ...])), axis=1)
    assignments = jnp.argmin(distance, axis=1)

    a = jnp.arange(centers.shape[0])
    new_centers = cluster_update_vmap(a, centers, assignments, X)

    diff = jnp.sum(jnp.square((new_centers - centers)))
    return new_centers, diff, assignments


class KMeansJax(KMeansBase):

    def __init__(self,
                 n_clusters: int,
                 max_iter: int = 100,
                 early_stop_threshold: float = 0.01,
                 seed: chex.PRNGKey = None,
                 init_centers=None):
```

```
        super().__init__(n_clusters, max_iter, early_stop_threshold)
        self.rand_key = random.PRNGKey(seed) if seed is not None else
random.PRNGKey(0)
        self.init_centers = init_centers

    def init_clusters(self, X, n_clusters):
        if self.init_centers is not None:
            assert n_clusters == len(self.init_centers), "Number of initial
centers provided must match n_clusters"

            return self.init_centers

        assert n_clusters <= len(X)
        self.rand_key, subkey = random.split(self.rand_key)
        i = random.choice(subkey, jnp.arange(len(X)), shape=(n_clusters,),
replace=False)
        centers = X[i]

        return centers



    def prepare(self, X):
        centers = self.init_clusters(X, self.n_clusters)
        centers = jnp.asarray(centers)
        X = jnp.asarray(X)

        return X, centers

    def fit(self, X, method="centroids", initial_centers=None):
        if initial_centers is None:
            X, initial_centers = self.prepare(X)

        self.call_hook('on_main_loop_start')
        centers, assignments = self._main_loop(X, initial_centers)
        self.call_hook('on_main_loop_end')
```

```
        centers, assignments = self.finalize(centers, assignments)

        self.call_hook('on_epoch_end', self, X, centers, assignments)

        return centers, assignments

    def _main_loop(self, X, centers):

        @jit
        def while_step(arg):
            iteration, centers, assignments, diff = arg
            new_centers, diff, assignments = step(X, centers)
            new_centers = new_centers.astype(centers.dtype)

            return (iteration + 1, new_centers, assignments, diff)

        @jit
        def cond(arg):
            iteration, centers, assignments, diff = arg
            return (iteration < self.max_iter) & (diff >
self.early_stop_threshold)

        assignments = jnp.zeros(X.shape[0], dtype=jnp.int32)

        initial_diff = jnp.array(1000, dtype=jnp.float32)

        iteration, centers, assignments, diff = jax.lax.while_loop(cond,
while_step,(0,centers, assignments, initial_diff))

        return centers, assignments

    def closest_points_to_center(self, input, assignments, centers,
cluster_index, x):

        mask = (assignments == cluster_index)

        large_value_mask = jnp.expand_dims(large_value_mask, -1)

        cluster_points = input + large_value_mask

        distances = jnp.sum(jnp.square(cluster_points -
centers[cluster_index]), axis=1)

        sorted_indices = jnp.argsort(distances)

        return cluster_points[sorted_indices][:x]
```

A.2 Implementation of Iso+LineDD Emitter on QDJAX

This appendix describes the QDJAX library's Iso+LineDD emitter implementation. As explained in section 3.5.1, the Iso+LineDD emitter is an advanced evolutionary strategy designed to exploit the structure in the search space by introducing small mutations in the direction of differences between solutions in the archive. In the module, **EmitterParams**, **EmitterState**, and **IsoLineEmitter** implement the emitter. These classes define the emitter's arguments, states, and functionality. The Iso+LineDD emitter was previously implemented in the QDJAX library, so this part is for reference and to help comprehend the proposed adjustments.

Iso+LineDD Emitter in QDJAX

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


from typing import Tuple, List
import jax.numpy as jnp
from functools import partial
from flax import struct
from flax.struct import PyTreeNode
import jax
from qdjax.core.emitters.emitter_base import EmitterBase
from qdjax.types import *
import numpy as np

# @struct.dataclass
class EmitterParams(PyTreeNode):
    """
    `Iso+LineDD Emitter Params`:

    Emitter Params define the parameters the algorithm uses
    and which can be changed manually outside of the
    of the Emitter.
    """
    x0: Params
    iso_sigma: float
    line_sigma: float
    lower_bounds: Array
    upper_bounds: Array

# @struct.dataclass
class EmitterState(PyTreeNode):
    """
    `Iso+LineDD Emitter State`:

    Emitter State defines the state of the Emitter that is defined
    and used only by the Emitter (should not be changed by user).
    """
    None # Empty DataClass
```

```
class IsoLineEmitter(EmitterBase):
    """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>`_.
    """
    def __init__(self,
                 archive,
                 x0,
                 iso_sigma,
                 line_sigma,
                 batch_size,
                 bounds=None,
                 **kwargs):
        # Static Parameters Describing Emitter
        self._emitter_name = "iso_line_emitter"
        self._uses_dqd = False
        self._evals_per_iter = self.get_evals_per_iterations(batch_size)
        # Static Parameters used in the Emitter that never change
        self._archive = archive
        self._solution_dim = len(x0)
        self._batch_size = batch_size
        # Default Parameters for EmitterParams dataclass
        self._x0 = np.array(x0, dtype=float)
        self._lower_bounds, self._upper_bounds = self.process_bounds(bounds,
self._solution_dim)
        self._iso_sigma = float(iso_sigma)
        self._line_sigma = float(line_sigma)

    @partial(jax.jit, static_argnames=("self",))
    def init(self, rand_key: RNGKey):
        """
        Initializes the instance with the defaule EmitterParams and
EmitterState
        """
        return (
                EmitterState(),
                EmitterParams(x0=self._x0,
                              iso_sigma=self._iso_sigma,
                              line_sigma=self._line_sigma,
                              lower_bounds=self._lower_bounds,
                              upper_bounds=self._upper_bounds)
        )

    @staticmethod
    def get_evals_per_iterations(batch_size):
        return batch_size

    @partial(jax.jit, static_argnames=("self",))
    def _ask_clip(self, parents, lower_bounds, upper_bounds):
        return jnp.minimum(jnp.maximum(parents, lower_bounds), upper_bounds)
```

```
    @partial(jax.jit, static_argnames=("self",))
    def ask(self,
            emitter_state: EmitterState,
            emitter_params: EmitterParams,
            archive_state: Any,
            rand_key: RNGKey):
        """
        Generates ``batch_size`` solutions.
        """

        batch_size = self._batch_size
        solution_dim = self._solution_dim
        key_selection, key_variation = jax.random.split(rand_key, 2)

        # SELECTION #
        key_select_p1, key_select_p2 = jax.random.split(key_selection, 2)
        sols_1, _ = self._archive.get_rand_elites(archive_state, batch_size,
key_select_p1)
        sols_2, _ = self._archive.get_rand_elites(archive_state, batch_size,
key_select_p2)

        # VARIATION #
        key_iso, key_line = jax.random.split(key_variation, 2)
        iso_gaussian = jax.random.normal(key_iso,
                            shape=(batch_size, solution_dim),
                            dtype=float) * emitter_params.iso_sigma

        # expanded last dimension used for multiplication later
        line_gaussian = jax.random.normal(key_line,
                            shape=(batch_size, 1),
                            dtype=float) * emitter_params.line_sigma

        new_sols = sols_1 + iso_gaussian + (sols_2 - sols_1) * line_gaussian

        return emitter_state, self._ask_clip(new_sols,
                        emitter_params.lower_bounds,
                        emitter_params.upper_bounds)

    @partial(jax.jit, static_argnames=("self",))
    def tell(self,
            emitter_state: EmitterState,
            emitter_params: EmitterParams,
            solutions: Params,
            objective_values,
            behavior_values,
            skip_sols_mask: Array,
            archive_state: Any,
            rand_key: RNGKey):
        """
        Inserts entries into the archive.
        """
        archive_state, _ =
self._archive.add_to_archive(archive_state=archive_state,
                             solutions=solutions,
                             behavior_values=behavior_values,
                             objective_values=objective_values,
                             skip_sols_mask=skip_sols_mask)
        return emitter_state, archive_state
```

A.3 Implementation of Iso+LineDD Emitter with Reduced Batch Sampling

This appendix discusses the QDJAX library's modified Iso+LineDD emitter with reduced batch sampling. As explained in section 3.5.2, this modification aims to alter the selection process of elites during offspring generation to facilitate the integration of clustering techniques in future work. The QDJAX library has been extended to support this modified Iso+LineDD emitter, and the implementation includes a new module, **iso_line_variation_emitter.py**, and an additional method, get_rand_elites_for_reduced_batch, in the MAP-Elites Grid Archive implementation (**grid_archive_base.py**). By providing the implementation details for the modified Iso+LineDD emitter with reduced batch sampling, this appendix serves as a reference for understanding the adaptation and its potential for enabling the integration of clustering techniques.

Modified Iso+LineDD Emitter with Reduced Batch Sampling

```
from functools import partial

import numpy as np
from flax.struct import PyTreeNode

from qdjax.core.emitters.emitter_base import EmitterBase
from qdjax.types import *


class EmitterParams(PyTreeNode):

    x0: Params
    iso_sigma: float
    line_sigma: float
    lower_bounds: Array
    upper_bounds: Array


class EmitterState(PyTreeNode):

    None  # Empty DataClass


class IsoLineVariationEmitter(EmitterBase):

    def __init__(self,
                 archive,
                 x0,
                 iso_sigma,
                 line_sigma,
                 batch_size,
                 num_elites_to_repeat,
                 bounds=None,
                 **kwargs):
```

```
        # Static Parameters Describing Emitter
        self._emitter_name = "iso_line_variation_emitter"
        self._uses_dqd = False
        self._evals_per_iter = self.get_evals_per_iterations(batch_size)
        # Static Parameters used in the Emitter that never change
        self._archive = archive
        self._solution_dim = len(x0)
        self._batch_size = batch_size
        self._num_of_elites_to_repeat = num_elites_to_repeat
        # Default Parameters for EmitterParams dataclass
        self._x0 = np.array(x0, dtype=float)
        self._lower_bounds, self._upper_bounds = self.process_bounds(bounds,
self._solution_dim)
        self._iso_sigma = float(iso_sigma)
        self._line_sigma = float(line_sigma)

    @partial(jax.jit, static_argnames=("self",))
    def init(self, rand_key: RNGKey):

        return (
            EmitterState(),
            EmitterParams(x0=self._x0,
                          iso_sigma=self._iso_sigma,
                          line_sigma=self._line_sigma,
                          lower_bounds=self._lower_bounds,
                          upper_bounds=self._upper_bounds)
        )

    @staticmethod
    def get_evals_per_iterations(batch_size):
        return batch_size

    @partial(jax.jit, static_argnames=("self",))
    def _ask_clip(self, parents, lower_bounds, upper_bounds):
        return jnp.minimum(jnp.maximum(parents, lower_bounds), upper_bounds)

    @partial(jax.jit, static_argnames=("self",))
    def ask(self,
            emitter_state: EmitterState,
            emitter_params: EmitterParams,
            archive_state: Any,
            rand_key: RNGKey):
        """
        Generates ``batch_size`` solutions.
        """
        batch_size = self._batch_size
        num_of_elites_to_repeat = self._num_of_elites_to_repeat
        solution_dim = self._solution_dim
        key_selection, key_variation = jax.random.split(rand_key, 2)

        # SELECTION #
        key_select_p1, key_select_p2 = jax.random.split(key_selection, 2)
        b1, _ =
self._archive.get_rand_elites_for_reduced_batch(archive_state, batch_size,
num_of_elites_to_repeat, key_select_p1)

        b2, _ = self._archive.get_rand_elites(archive_state, batch_size,
key_select_p2)
```

```
        # VARIATION #
        key_iso, key_line = jax.random.split(key_variation, 2)
        iso_gaussian = jax.random.normal(key_iso,
                                         shape=(batch_size,
                                                solution_dim),
                                    dtype=float) * emitter_params.iso_sigma

        # expanded last dimension used for multiplication later
        line_gaussian = jax.random.normal(key_line,
                                          shape=(batch_size, 1),
                                   dtype=float) * emitter_params.line_sigma

        new_sols = b1 + iso_gaussian + (b2 - b1) * line_gaussian

        return emitter_state, self._ask_clip(new_sols,
                                             emitter_params.lower_bounds,
                                             emitter_params.upper_bounds)



    @partial(jax.jit, static_argnames=("self",))
    def tell(self,
             emitter_state: EmitterState,
             emitter_params: EmitterParams,
             solutions: Params,
             objective_values,
             behavior_values,
             skip_sols_mask: Array,
             archive_state: Any,
             rand_key: RNGKey):
        """
        Inserts entries into the archive.
        """
        archive_state, _ =
self._archive.add_to_archive(archive_state=archive_state,
                             solutions=solutions,
                             behavior_values=behavior_values,
                             objective_values=objective_values,
                             skip_sols_mask=skip_sols_mask)
        return emitter_state, archive_state
```

Extension of the Archive class to support the modified emitter

```
@partial(jax.jit, static_argnames=("self",
                                   "batch_size",
                                   "num_of_rand_elites"))
def get_rand_elites_for_reduced_batch(self,
                                      archive_state: ArchiveState,
                                      batch_size: int,
                                      num_of_rand_elites: int,
                                      rand_key: chex.PRNGKey,
                                      metadata: Dict[str, Any] = None) ->
Tuple[chex.Array, Dict[str, Any]]:
    """
    Generates a batch of elite solutions from the archive by randomly
    selecting 'num_of_rand_elites' solutions and repeating each solution to
    fill the 'batch_size'. Returns the repeated elite solutions along with
    their indices in the archive.
    """
```

```
    res = GridArchiveBase._get_rand_elites(self._grid_shape,
                                           archive_state,
                                           num_of_rand_elites,
                                           rand_key,
                                           metadata)
    elites, elite_metadata = res[0], res[1]

    repeats = batch_size // num_of_rand_elites
    remaining = batch_size % num_of_rand_elites

    # Repeat each elite solution 'repeats' times
    repeated_elites = jnp.repeat(elites, repeats, axis=0)

    if remaining > 0:
        extra_elites = elites[:remaining]
        repeated_elites = jnp.concatenate([repeated_elites, extra_elites],
                                          axis=0)

    # Repeat the indices accordingly
    repeated_indices = jnp.repeat(elite_metadata['elites_indices'], repeats)

    # If there are remaining solutions needed to fill the batch,
    # repeat the first 'remaining' elite indices
    if remaining > 0:
        extra_indices = elite_metadata['elites_indices'][:remaining]
        repeated_indices = jnp.concatenate([repeated_indices,extra_indices],
axis=0)

    return repeated_elites, {'elites_indices': repeated_indices}
```

A.4 Implementation of Strategy 1

Strategy 1 - Perturbation Towards Elites Closest to Cluster Centers is implemented in the QDJAX library in this appendix. As mentioned in section 3.5.3.1, this technique used K-Means clustering to improve the Iso+LineDD emitter selection process by perturbing baseline elites towards the direction of correlation of the elites closest to the cluster centers. The implementation includes a new module, **iso_line_kmeans_variation.py**, and an additional method, get_closest_elites_kmeans, extended in the MAP-Elites Grid Archive implementation (**grid_archive_base.py**).

Modified Iso+LineDD Emitter (Strategy 1)

```
"""
Provides the IsoLineKmeansVariationEmitter.
Adapted Iso+LineDD Emitter that introduces K-Means Clustering in the
selection step of the ask method. It selects the elites that are closer to
the cluster centers and samples from them. The implementation of K-Means
Clustering on JAX is used for this purpose.
"""

from functools import partial
from typing import Optional

import numpy as np
from flax.struct import PyTreeNode

from qdjax.core.emitters.emitter_base import EmitterBase
from qdjax.types import *


class EmitterParams(PyTreeNode):
    """
    `Modified Iso+LineDD Emitter Params`:

    Emitter Params define the parameters the algorithm uses
    and which can be changed manually outside of the
    of the Emitter.
    """
    x0: Params
    iso_sigma: float
    line_sigma: float
    lower_bounds: Array
    upper_bounds: Array


class EmitterState(PyTreeNode):
    """
    `Modified Iso+LineDD Emitter State`:

    Emitter State defines the state of the Emitter that is defined
    and used only by the Emitter (should not be changed by user).
    """
    centers: Optional[Array] = None


class IsoLineKmeansVariationEmitter(EmitterBase):
```

```
    """
    Adapted Iso+LineDD Emitter that introduces K-Means Clustering in the
    selection step of the ask method.
    """

    def __init__(self,
                 archive,
                 x0,
                 iso_sigma,
                 line_sigma,
                 batch_size,
                 num_of_clusters,
                 num_elites_to_repeat,
                 bounds=None,
                 **kwargs):
        # Static Parameters Describing Emitter
        self._emitter_name = "iso_line_kmeans_variation_emitter"
        self._uses_dqd = False
        self._evals_per_iter = self.get_evals_per_iterations(batch_size)
        # Static Parameters used in the Emitter that never change
        self._archive = archive
        self._solution_dim = len(x0)
        self._batch_size = batch_size
        self._num_of_clusters = num_of_clusters
        self._num_elites_to_repeat = num_elites_to_repeat
        # Default Parameters for EmitterParams dataclass
        self._x0 = np.array(x0, dtype=float)
        self._lower_bounds, self._upper_bounds = self.process_bounds(bounds,
self._solution_dim)
        self._iso_sigma = float(iso_sigma)
        self._line_sigma = float(line_sigma)

    @partial(jax.jit, static_argnames=("self",))
    def init(self, rand_key: RNGKey):
        """
        Initializes the instance with the default EmitterParams and
        EmitterState
        """
        # Initialize the centers with zeros, for example.
        initial_centers = jnp.zeros((self._num_of_clusters,
                                     self._solution_dim))

        return (
            EmitterState(centers=initial_centers),
            EmitterParams(x0=self._x0,
                          iso_sigma=self._iso_sigma,
                          line_sigma=self._line_sigma,
                          lower_bounds=self._lower_bounds,
                          upper_bounds=self._upper_bounds)
        )

    @staticmethod
    def get_evals_per_iterations(batch_size):
        return batch_size



    @partial(jax.jit, static_argnames=("self",))
    def _ask_clip(self, parents, lower_bounds, upper_bounds):
        return jnp.minimum(jnp.maximum(parents, lower_bounds),
                           upper_bounds)
```

```python
@partial(jax.jit, static_argnames=("self",))
def ask(self,
        emitter_state: EmitterState,
        emitter_params: EmitterParams,
        archive_state: Any,
        rand_key: RNGKey):
    """
    Generates ``batch_size`` solutions.
    """
    batch_size = self._batch_size
    num_of_clusters = self._num_of_clusters
    num_elites_to_repeat = self._num_elites_to_repeat
    solution_dim = self._solution_dim
    centers = emitter_state.centers

    # Check if the centers array contains all zeros
    is_centers_zeros = jnp.all(jnp.equal(centers, 0))

    key_selection, key_variation = jax.random.split(rand_key, 2)

    # SELECTION #
    key_select_p1, key_select_p2 = jax.random.split(key_selection, 2)
    b1, _ = self._archive.get_rand_elites_for_reduced_batch(
                                                        archive_state,
                                                        batch_size,
                                                num_elites_to_repeat,
                                                        key_select_p1)

    def centers_zeros_true_fn(_):
        return self._archive. get_closest_elites_kmeans(archive_state,
                                                        batch_size,
                                                        key_select_p2,
                                                        num_of_clusters,
                                                        None)

    def centers_zeros_false_fn(_):
        return self._archive. get_closest_elites_kmeans(archive_state,
                                                        batch_size,
                                                        key_select_p2,
                                                        num_of_clusters,
                                                    emitter_state.centers)

    b2, centers, _ = jax.lax.cond(is_centers_zeros,
                                  None,
                                  centers_zeros_true_fn,
                                  None,
                                  centers_zeros_false_fn)

    # Update the EmitterState with the new centers
    emitter_state = emitter_state.replace(centers=centers)
```

```
        # VARIATION #
        key_iso, key_line = jax.random.split(key_variation, 2)
        iso_gaussian = jax.random.normal(key_iso,
                                         shape=(batch_size,
                                                solution_dim),
                                         dtype=float) *
emitter_params.iso_sigma

        # expanded last dimension used for multiplication later
        line_gaussian = jax.random.normal(key_line,
                                          shape=(batch_size,
                                                 1),
                                          dtype=float) *
emitter_params.line_sigma

        new_sols = b1 + iso_gaussian + (b2 - b1) * line_gaussian

        return emitter_state, self._ask_clip(new_sols,
                                             emitter_params.lower_bounds,
                                             emitter_params.upper_bounds)

    @partial(jax.jit, static_argnames=("self",))
    def tell(self,
             emitter_state: EmitterState,
             emitter_params: EmitterParams,
             solutions: Params,
             objective_values,
             behavior_values,
             skip_sols_mask: Array,
             archive_state: Any,
             rand_key: RNGKey):
        """
        Inserts entries into the archive.
        """
        archive_state, _ =
self._archive.add_to_archive(archive_state=archive_state,
                             solutions=solutions,
                             behavior_values=behavior_values,
                             objective_values=objective_values,
                             skip_sols_mask=skip_sols_mask)
        return emitter_state, archive_state
```

Extension of the Archive class to support the modified emitter

```
@partial(jax.jit, static_argnames=("self",
                                   "batch_size",
                                   "num_of_clusters"))
def get_closest_elites_kmeans(self,
                              archive_state: ArchiveState,
                              batch_size: int,
                              rand_key: chex.PRNGKey,
                              num_of_clusters: int,
                              init_centers: chex.Array = None,
                              metadata: Dict[str, Any] = None) ->
Tuple[chex.Array, chex.Array, Dict[str, Any]]:
```

```
return GridArchiveBase._get_closest_elites_kmeans(self._grid_shape,
                                                  archive_state,
                                                  batch_size,
                                                  rand_key,
                                                  metadata,
                                                  num_of_clusters,
                                                  init_centers)
```

```
@staticmethod
def _get_closest_elites_kmeans(grid_shape: chex.Array,
                               archive_state: ArchiveState,
                               batch_size: int,
                               rand_key: chex.PRNGKey,
                               metadata=None,
                               num_of_clusters: int = 3,
                               init_centers: chex.Array = None):
    """
    Selects the closest elites from cluster centers using K-means
clustering.
    """
    # Build the Grid Archive's index elites map
    total_cells = archive_state.objective_values.shape[0]
    fitness_2d = archive_state.objective_values.reshape(grid_shape)
    # Find indices of non-empty cells
    indices = jnp.argwhere(jnp.logical_not(jnp.isnan(fitness_2d)),
                           size=total_cells)
    indices = jnp.transpose(indices, axes=(1, 0))
    indices_1d = jnp.array(jnp.ravel_multi_index(indices,
                                                 fitness_2d.shape,
                                                 mode='clip')).astype(int)

    elites_order_indices = jax.random.randint(rand_key,
                                              shape=(indices_1d.shape),
                                              minval=0,

maxval=archive_state.num_of_sols)
    # Select the elites
    elites_indices = indices_1d.at[elites_order_indices].get()
    elites = archive_state.sols_archive.at[elites_indices].get()

    if init_centers is None:
        kmeans = KMeansJax(n_clusters=num_of_clusters,
                           max_iter=10,
                           early_stop_threshold=0.01,
                           seed=rand_key[0])
    else:
        kmeans = KMeansJax(n_clusters=num_of_clusters,
                           max_iter=4,
                           early_stop_threshold=0.01,
                           seed=rand_key[0],
                           init_centers=init_centers)
    centers, assignments = kmeans.fit(elites)

    # Calculate the number of elites that should be taken per cluster and
the remainder
    if batch_size < num_of_clusters:
        # Calculate the number of elites that should be taken per cluster
        num_per_cluster = 1

        selected_elites = []
```

```
        # Randomly select batch_size clusters
        rand_key, subkey = jax.random.split(rand_key)
        selected_cluster_indices = jax.random.choice(subkey,
                                                     jnp.arange(num_of_clusters),
                                                           shape=(batch_size,),
                                                           replace=False)

        # Get the closest num_per_cluster elite from each selected cluster
        for cluster_index in selected_cluster_indices:
            closest_points = kmeans.closest_points_to_center(elites,
                                                             assignments,
                                                             centers,
                                                             cluster_index,
                                                          x=num_per_cluster)
            selected_elites.append(closest_points)

        # Concatenate selected elites
        selected_elites = jnp.vstack(selected_elites)
    else:
        num_per_cluster = batch_size // num_of_clusters
        remainder = batch_size % num_of_clusters

        selected_elites = []

        # Get the closest num_per_cluster elites from each cluster
        for cluster_index in range(num_of_clusters):
            closest_points = kmeans.closest_points_to_center(elites,
assignments, centers, cluster_index,

x=num_per_cluster)
            selected_elites.append(closest_points)

        # Get the next closest elite from the first remainder clusters
        for cluster_index in range(remainder):
            next_closest_point = \
            kmeans.closest_points_to_center(elites,
                                            assignments,
                                            centers,
                                            cluster_index,
                                            x=num_per_cluster + 1)[-1]
            selected_elites[cluster_index] =
jnp.vstack((selected_elites[cluster_index],next_closest_point)

        # Concatenate selected elites
        selected_elites = jnp.vstack(selected_elites)

    return selected_elites, centers, {'elites_indices': None}
```

A.5 Implementation of Strategy 2

Strategy 2 - Perturbation Towards Cluster Centroids is implemented in QDJAX library in this appendix. As explained in section 3.5.3.2, this strategy perturbs solutions towards the direction of correlation of the centroids of clusters instead of the closest elites from each cluster center, as in Strategy 1. The implementation includes a new method, get_centroid_elites_kmeans, extended in the MAP-Elites Grid Archive implementation (**grid_archive_base.py**) and the new module for the proposed emitter, **iso_line_kmeans_centroids_variation.py.**

Modified Iso+LineDD Emitter (Strategy 2)

```
"""
Provides the IsoLineKmeansCentroidsVariationEmitter.
Adapted Iso+LineDD Emitter that introduces K-Means Clustering in the
selection step of the ask method. Ít selects the centroids of the cluster
centers and samples from them, utilizing them as elites. The implementation
of K-Means Clustering on JAX is used for this purpose.
"""

from functools import partial
from typing import Optional

import numpy as np
from flax.struct import PyTreeNode

from qdjax.core.emitters.emitter_base import EmitterBase
from qdjax.types import *


class EmitterParams(PyTreeNode):
    """
    `Mofified Iso+LineDD Emitter Params`:

    Emitter Params define the parameters the algorithm uses
    and which can be changed manually outside of the
    of the Emitter.
    """
    x0: Params
    iso_sigma: float
    line_sigma: float
    lower_bounds: Array
    upper_bounds: Array

class EmitterState(PyTreeNode):
    """
    `Modified Iso+LineDD Emitter State`:

    Emitter State defines the state of the Emitter that is defined
    and used only by the Emitter (should not be changed by user).
    """
    centers: Optional[Array] = None

class IsoLineKmeansCentroidsVariationEmitter(EmitterBase):
    """
    Adapted Iso+LineDD Emitter that introduces K-Means Clustering in the
    selection step of the ask method.
    """
```

```
    def __init__(self,
                 archive,
                 x0,
                 iso_sigma,
                 line_sigma,
                 batch_size,
                 num_of_clusters,
                 num_elites_to_repeat,
                 bounds=None,
                 **kwargs):
        # Static Parameters Describing Emitter
        self._emitter_name = "iso_line_kmeans_centroids_variation_emitter"
        self._uses_dqd = False
        self._evals_per_iter = self.get_evals_per_iterations(batch_size)
        # Static Parameters used in the Emitter that never change
        self._archive = archive
        self._solution_dim = len(x0)
        self._batch_size = batch_size
        self._num_of_clusters = num_of_clusters
        self._num_elites_to_repeat = num_elites_to_repeat
        # Default Parameters for EmitterParams dataclass
        self._x0 = np.array(x0, dtype=float)
        self._lower_bounds, self._upper_bounds = self.process_bounds(bounds,
self._solution_dim)
        self._iso_sigma = float(iso_sigma)
        self._line_sigma = float(line_sigma)

    @partial(jax.jit, static_argnames=("self",))
    def init(self, rand_key: RNGKey):
        """
        Initializes the instance with the default EmitterParams and
        EmitterState
        """
        # Initialize the centers with zeros, for example.
        initial_centers = jnp.zeros((self._num_of_clusters,
self._solution_dim))

        return (
                EmitterState(centers=initial_centers),
                EmitterParams(x0=self._x0,
                              iso_sigma=self._iso_sigma,
                              line_sigma=self._line_sigma,
                              lower_bounds=self._lower_bounds,
                              upper_bounds=self._upper_bounds)
        )

    @staticmethod
    def get_evals_per_iterations(batch_size):
        return batch_size

    @partial(jax.jit, static_argnames=("self",))
    def _ask_clip(self, parents, lower_bounds, upper_bounds):
        return jnp.minimum(jnp.maximum(parents,
                                       lower_bounds),
                           upper_bounds)
```

```
    @partial(jax.jit, static_argnames=("self",))
    def ask(self,
            emitter_state: EmitterState,
            emitter_params: EmitterParams,
            archive_state: Any,
            rand_key: RNGKey):
        """
        Generates ``batch_size`` solutions.
        """
        batch_size = self._batch_size
        num_of_clusters = self._num_of_clusters
        num_elites_to_repeat = self._num_elites_to_repeat
        solution_dim = self._solution_dim
        centers = emitter_state.centers

        # Check if the centers array contains all zeros
        is_centers_zeros = jnp.all(jnp.equal(centers, 0))

        key_selection, key_variation = jax.random.split(rand_key, 2)

                # SELECTION #
        key_select_p1, key_select_p2 = jax.random.split(key_selection, 2)
        b1, _ = self._archive.get_rand_elites_for_reduced_batch(
                                                                archive_state,
                                                                batch_size,
                                                        num_elites_to_repeat,
                                                                key_select_p1)

        def centers_zeros_true_fn(_):
            return self._archive.get_centroid_elites_kmeans(archive_state,
                                                            batch_size,
                                                            key_select_p2,
                                                            num_of_clusters,
                                                            None)

        def centers_zeros_false_fn(_):
            return self._archive.get_centroid_elites_kmeans(archive_state,
                                                            batch_size,
                                                            key_select_p2,
                                                            num_of_clusters,
                                                        emitter_state.centers)

        b2, centers, _ = jax.lax.cond(is_centers_zeros, None,
centers_zeros_true_fn, None, centers_zeros_false_fn)

        # Update the EmitterState with the new centers
        emitter_state = emitter_state.replace(centers=centers)

        # VARIATION #
        key_iso, key_line = jax.random.split(key_variation, 2)
        iso_gaussian = jax.random.normal(key_iso,
                             shape=(batch_size, solution_dim),
                             dtype=float) * emitter_params.iso_sigma

        # expanded last dimension used for multiplication later
        line_gaussian = jax.random.normal(key_line,
                             shape=(batch_size, 1),
                             dtype=float) * emitter_params.line_sigma

        new_sols = b1 + iso_gaussian + (b2 - b1) * line_gaussian
```

```
        return emitter_state, self._ask_clip(new_sols,
                        emitter_params.lower_bounds,
                        emitter_params.upper_bounds)

    @partial(jax.jit, static_argnames=("self",))
    def tell(self,
             emitter_state: EmitterState,
             emitter_params: EmitterParams,
             solutions: Params,
             objective_values,
             behavior_values,
             skip_sols_mask: Array,
             archive_state: Any,
             rand_key: RNGKey):
        """
        Inserts entries into the archive.
        """
        archive_state, _ =
self._archive.add_to_archive(archive_state=archive_state,
                             solutions=solutions,
                             behavior_values=behavior_values,
                             objective_values=objective_values,
                             skip_sols_mask=skip_sols_mask)
        return emitter_state, archive_state
```

Method:

```
def get_centroid_elites_kmeans(self,
                           archive_state: ArchiveState,
                           batch_size: int,
                           rand_key: chex.PRNGKey,
                           num_of_clusters: int,
                               init_centers: chex.Array = None,
                           metadata: Dict[str, Any] = None) ->
Tuple[chex.Array, chex.Array, Dict[str, Any]]:

    return GridArchiveBase._get_centroid_elites_kmeans(self._grid_shape,
                                                       archive_state,
                                                       batch_size,
                                                       rand_key,
                                                       metadata,
                                                       num_of_clusters,
                                                       init_centers)
```

Extension of the Archive class to support the modified emitter

```
@staticmethod
def _get_centroid_elites_kmeans(grid_shape: chex.Array,
                                archive_state: ArchiveState,
                                batch_size: int,
                                rand_key: chex.PRNGKey,
                                metadata=None,
                                num_of_clusters: int = 3,
                                init_centers: chex.Array = None):
    """
    Selects elites using K-means clustering. The method that returns a batch
    of batch_size elites where the elites are the centers (centroids) of the
    clusters created from kmeans. If the clusters that are created are less
    than the value of batch_size, it repeats each in the batch a specific
    amount of times in order to return batch_size elites and not less. On
    the other, if there are more clusters than the value of batch_size, it
    returns random batch_size centers as the batch:
    """
    # Build the Grid Archive's index elites map
    total_cells = archive_state.objective_values.shape[0]
    fitness_2d = archive_state.objective_values.reshape(grid_shape)
    # Find indices of non-empty cells
    indices = jnp.argwhere(jnp.logical_not(jnp.isnan(fitness_2d)),
size=total_cells)
    indices = jnp.transpose(indices, axes=(1, 0))
    indices_1d = jnp.array(jnp.ravel_multi_index(indices, fitness_2d.shape,
mode='clip')).astype(int)

    elites_order_indices = jax.random.randint(rand_key,
                                              shape=(indices_1d.shape),
                                              minval=0,
                                          maxval=archive_state.num_of_sols)

    elites_indices = indices_1d.at[elites_order_indices].get()
    elites = archive_state.sols_archive.at[elites_indices].get()

    if init_centers is None:
        kmeans = KMeansJax(n_clusters=num_of_clusters,
                           max_iter=10,
                           early_stop_threshold=0.01,
                           seed=rand_key[0])

    else:
        kmeans = KMeansJax(n_clusters=num_of_clusters,
                           max_iter=4,
                           early_stop_threshold=0.01,
                           seed=rand_key[0],
                           init_centers=init_centers)
    centers, assignments = kmeans.fit(elites)

    if batch_size < num_of_clusters:
        # Randomly select batch_size clusters
        rand_key, subkey = jax.random.split(rand_key)
        selected_cluster_indices = jax.random.choice(subkey,
                                                     jnp.arange(num_of_clusters),
                                                          shape=(batch_size,),
                                                          replace=False)

        # Get the closest num_per_cluster elite from each selected cluster
        selected_elites = centers[selected_cluster_indices]
```

```
    else:
        # Repeat the centroids if there are less than batch_size centers
        num_of_repeats = batch_size // num_of_clusters
        remainder = batch_size % num_of_clusters

        selected_elites = jnp.repeat(centers, num_of_repeats, axis=0)

        # Add the centers from the first remainder clusters to the batch
        if remainder > 0:
            selected_elites = jnp.vstack((selected_elites,
                                          centers[:remainder]))

    return selected_elites, centers, {'elites_indices': None}
```