Diploma Thesis

DYNAMIC PROGRAMMING: AN EXPERIMENTAL ANALYSIS AND COMPARISON OF THE TOP-DOWN AND BOTTOM-UP APPROACHES

Nicolas Zachariou

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

January 2021

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

DYNAMIC PROGRAMMING: AN EXPERIMENTAL ANALYSIS AND COMPARISON OF THE TOP-DOWN AND BOTTOM-UP APPROACHES

Nicolas Zachariou

Supervisor

Dr. Chryssis Georgiou

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

January 2021

Acknowledgements

I would like to thank my professor Dr. Chryssis Georgiou. His ideas, guidance and supervision helped me get throughout the whole process. I would also like to thank my family for the support, but most importantly, my significant other MD. Eleni Shimitra for standing by me.

Abstract

Curiosity is key in evolution; may that be as a species or personal growth. But how good would curiosity be if one couldn't answer any of his questions? Us, humans generally love knowing the answers to everything, and we continuously develop tools that help us answer our questions and finding the best solutions to our problems.

Dynamic Programming is an Algorithmic Approach that builds optimal solutions to problems by dynamically dividing a problem into smaller parts, and then combining these smaller so called "sub-problems" to find the optimal solution for bigger and bigger parts.

In this study, we investigate in depth the two basic implementation approaches of dynamic programming: bottom up and top down. The first is iterative and the second is recursive, which uses memoisation so that no subproblem is computed more than once. By implementing dynamic programming solutions of basic problems using both approaches, we aim in analysing their performance with respect to execution time, resource usage and ease of use (development time). The goal is to classify or rather, group dynamic programming solutions based on their performance and if possible, to find a trend or even discover a rule of thumb, that may be able to provide an insight, as to which implementation approach is more efficient for each problem.

Table of Contents

Chapter 1

Introduction

1.1 What is Dynamic Programming?

Dynamic Programming [1] is an algorithmic approach that helps solve problems with an optimal description quickly and efficiently. For an algorithm to be considered as a Dynamic Programming algorithm, it must fulfil two requirements. Firstly, it must have an explicit recursive description and / or it must be described by an optimal function. Additionally, it must store the value of each of its calculated sub-problems and use this information to solve others of its sub-problems.

A sub-problem is a problem described by the same function as the original problem that is (or may have) derived from the original problem. This is the essence of dynamic programming. When the problem is presented, smaller sub-problems can be derived from its optimal function, these sub-problems can also be divided in the same manner. By storing the values of each and every one of these sub-problems we can ensure that each sub-problem will be calculated once at most, preventing excess calculations.

This technique is very useful when dealing with recursive functions, however it can also be used to calculate optimal solutions quickly in an iterative manner. This is where the two implementations make their entrance.

1.2 Approaches to Dynamic Programming

The two implementation approaches of Dynamic Programming are the **Iterative** (also known as **Bottom-Up**) and the **Recursive** (also known as **Top-down**) approaches.

When dealing with recursive descriptions, we can use then to divide a problem into smaller sub-problems, with each sub-problem being an optimal solution to a part of the original problem. Using this knowledge, we can compute the value of each required sub-problem only once and find the solution to the original problem without unneeded computations. This approach is generally thought to have the worst performance of the two because of its recursive nature [4].

The other approach involves an iterative method. We start from smaller sub-problems that can easily be computed to find the optimal solution for bigger sub-problems. We continue this process for all possible sub-problems, and with every iteration we move into bigger and more complex sub-problems. This continues until all of the sub-problems have been calculated, and thus the final answer is found.

In both cases we use a table to store the states of any computed sub-problems and we use this table to retrieve any information that has already been calculated thus speeding up the process.

The iterative technique is also known as **tabulation**, as we 'build' or 'fill' an array / a table in a bottom – up manner. This is because we start from small, easy problems and we build into bigger ones. The recursive technique is also known as **memoization** (or **memoisation**). This technique is what speeds up the recursion since it prevents the program from calculating a sub-problem more than once from different recursion branches (by storing already computed subproblems in a table).

Implementing a recursive problem is generally a fast process. On the other hand, the recursive approach is generally thought of as the slower one regarding execution times and memory usage, since its recursive nature slows it down and makes use of multiple stacks. *However, is this always the case?*

We failed to find in the literature any experimental study demonstrating that the iterative approach is more efficient in practice.

1.3 Objective of the study

The objective of this work is to study a variety of dynamic algorithms implemented with both approaches in order to find possible patterns, reach some conclusions and classify these algorithms in terms of their performance, both regarding execution time, memory usage and general efficiency. What affects an implementations performance is its input complexity, and the form of its input data. Another factor is the complexity of the solution. This study attempts to provide evidence that in practice, depending on the problem, the iterative approach is not always the best option.

1.4 Methodology

To ensure we had a better understanding of Dynamic Programming we made sure to study old courses to remind ourselves about the techniques used in Dynamic Programming [5], as well as the corresponding chapters of well-known algorithmic books[1][2]. We experimented with some problems to make sure we had a strong grip of the concept. Then we began exploring new data structures, and ways that could increase the efficiency of either implementation. We later investigated different resource measuring tools to help us record the usage of resources like CPU utilization [9], memory usage [7] and execution time [10]. We began testing on data collection by developing scripts that would automate the process. After this we were ready to start implementing problems. We will view more details about the tools we use and the process behind the data collection in Chapter 2. After this, we were ready to start implementing dynamic programming solutions of carefully selected problems.

We tried to cover a variety of problems, based on the input structure; we investigated problems concerning sequences, problems on tables (arrays), graphs and trees. In particular we considered the following problems:

Sequences:

Most Common Sub Sequence [1][5] - Finding the longest common sequence contained in two other sequences.

Longest increasing Sub Sequence [2] – Finding the longest increasing sub sequence of a list / sequence of numbers. We will solve it in two ways. For the first solution we use a 1-Dimension array and a 2-Dimension array for the other solution.

Tables:

Chain Matrix Multiplication [1][2] – The problem of finding the best order in which matrix multiplications should be done. Matrix multiplications have a cost. We aim to find the order in which the cost of multiplying a set of matrices is the smallest possible.

0-1 Knapsack [2][3] - A standard problem of Dynamic Programming. In a few words, it is the problem of finding the best combination of items to maximize the total value of a sack that can contain a certain weight.

Graphs:

Dijkstra's Shortest Path [2] - It is a very useful problem on Graphs, which finds the shortest possible path between two given nodes.

Trees:

Independent Set [2] – A problem which requires a set of nodes and returns its biggest independent set.

K-Trees – The problem of finding the amount of sub-tress of size *K*, in a given Tree.

Tree Diameter – We find the diameter of the tree which is the maximum distance between any two nodes in a tree.

For each problem we provide its definition, its recursive expression, the pseudocode of each approach (Bottom-Up and Top-Down) and the experimental evaluation of their implementation.

1.5 Document Structure

In **Chapter 2** we discuss the background research that lead us to record data usage as we do. We also discuss the tools we use, how we use them and what their use may imply regarding the data we collect.

In **Chapter 3** we investigate the Most Common Sub Sequence problem. In **Chapter 4** we continue with sequences and we work on the Longest Increasing Sub Sequence problem in 1D and 2D. In **Chapter 5** we move to problems in tables by exploring the Chain Matrix Multiplication problem. In **Chapter 6** we investigate the 0-1 Knapsack problem. In **Chapter 7** we investigate our problem on graphs which is Dijkstra's shortest path. Our final category of problems starts in **Chapter 8** with the Independent Sets problem. Later, in **Chapter 9** we investigate the K-Trees problem and finally in **Chapter 10** we explore the Tree diameter problem.

In **Chapter 11** we give our last thoughts and last conclusions after giving a small recap of all previous chapters. In this final chapter we also expand on the problems we have encountered during the development of this project and some possible future research directions based on our experience.

Chapter 2

Background

2.1 Preliminary Study

We can have an introduction to Dynamic Programming from the courses of University of Cyprus that study algorithms [5]. To understand the behaviour of each implementation we followed execution examples as well as pseudo code. We experimented with our own implementations and tried to follow through these implementations to better understand how each approach works.

2.2 Experimenting with other Data Structures

Before beginning with problem implementations and data collection we have experimented with different types of data structures. We questioned whether there was another data structure that allowed for better efficiency than the 2D array. We used lists, hash maps and even trees. But we realised that the selection complexity of the array $O(1)$ was its biggest strength, something that could not be rivalled by any other structure. We also noticed that the pointer used to implement a linked list or any other structure that uses a "node" like a tree, made it very inefficient in storing data. To be more precise, lets calculate the required amount of memory in Bytes to store a single number (Integer). In a *64Bit* computer, an integer most likely uses *4* Bytes to be stored. However, any pointer uses *8* (thus, *64Bits*). Therefore, to store a

single integer in a data structure with a "node" we would need *8+4 Bytes = 12 Bytes*, which is *3* times more memory. Concluding to the following: For any "node" based data structure to be more efficient than an array, more than $\frac{2}{3}$ of the array must be unused. Therefore, the number of unique branches a recursion must make should be less than $\frac{1}{3}$ of *N* (if we assume than an iterative approach uses an *NxN* array).

2.3 Measuring Execution Time

After exploring other data structures and concluding to the table as the best option, we began researching ways to measure memory usage and execution time. We realised that some Dynamic Programming problems require some pre-processing, others require some data conversions. We concluded that we could not use any external process to capture the execution time because by doing that we would also measure the input generation or input reading as part of the execution. However, if any type of data conversion is required for either of the approaches to work, that would be measured since it is part of the calculation. The execution time between two identical runs may not be as consistent, to remove the effect of outliers in our data we will be taking the data of 12 runs for each execution. We will then remove the lowest and highest values before taking the average of the 10 remaining runs. For every measurement we generate the input randomly to capture the average of realistic scenarios.

2.4 Memory Measurement

To measure the memory usage, we seeked the help of external tools. We used the **Valgrind** memory checking tool [7] available for Linux distributions. We realised that this tool allowed for a thorough inspection of all memory allocation calls by the system, giving a very precise answer as to how much **Heap** a program uses, as well as the **Extra Heap** allocated. This tool however does not measure the **Stack** usage of a program. To measure the Stack, we use a profiler of the Valgrind tool called **Massif** [8]. This tool however has a downfall which was not discovered at first. To closely inspect the stack usage of a program each allocation or call to the stack that occurs is measured, therefore each call requires even more memory, this means that recursive programs that usually require plenty of stack could not be measured since the demand of the stack was bigger than the available system memory.

To get through this problem we began studying other memory measuring tools and methods. We realised that the system monitoring tool was our best bet. To capture the memory usage of a program, we start the program in a new process to capture its Process ID (PID). Then we inspect the memory usage of that process using the "**top**" tool [9] available for Linux. We capture the memory usage % in intervals of 0.1 seconds, and store it in a temporary file, later finding the maximum amount of memory usage from that file. If possible we use both the Valgrind and Top tools for the same problem comparing the results, and we found out that both techniques yielded very similar results. Because the memory measurements are much more consistent and run slower than time measurements, we will be taking the average of 3 runs for every memory measurement.

It should be noted that the Valgrind tool [7] retrieves the memory usage of a program in snapshots. Therefore, in rare cases the memory usage may be different, this is why the memory usage was recorded as an average of 3 runs. On the contrary, the stack usage is inspected more carefully, this is also why it requires much more memory to be measured and is also much slower.

2.5 System Used

The System we used to run the implementations and took our measurements operated over an AMD Ryzen 7 processor, namely the R7 4700U mobile APU. The total system memory was 16GB of ram.

The Operating System used during the development was Ubuntu 20.04LTS. All programs are developed in C++ (C std 11) in visual studio code. We believe that C++ is a very suitable language since it enables precise memory management and avoids unnecessary performance overheads; hence it allows for a more precise comparison between the two approaches, both regarding memory usage and execution time. To make sure we tested our implementations to the limits allowed by our system, we chose input sizes such as the memory usage was eventually maxed out.

2.6 CPU Utilization Measurements

The CPU utilization was closely monitored via Ubuntu built in tools. The Ubuntu system monitor tool provided a general resource usage image while the “top” program and the resource monitoring tool was used to monitor CPU utilization specifically. We found that our implementations did not use any sort of multithreading optimizations and the CPU utilization ranged at the maximum of 12% (100% single core utilization, 12% of 8 cores). Therefore, no multithreading advantages were given to any of our implementations.

Chapter 3

Most Common Sub Sequence

3.1 Problem Description and Dynamic Programming Solution

The Most Common Sub-Sequence problem [1] regards two letter sequences *A* and *B* and seeks the biggest common sub-sequence of letters found in both *A* and *B* (not necessarily consecutive). A variation of this problem is used by biologists with strands of DNA to better understand it. All letters of the MCSS must be included in both Sequences.

.

For example, the MCSS of the sequence “APPLES” and “PINEAPPLE” has a length of 5. This example is shown in **Table 1** and **Table 2**.

Definitions

- Sequence *A*, *B*
- Alphabet size *S*
- *A(i)* is the *i-th* character of *A*
- *B(j)* is the *j-th* character of *B*
- *length(A) = N*, *length(B) = M*
- Array of *(N+1)x(M+1)*

Sub-Problem

Let $P(i,j)$ be the optimal solution for the first *i-th* letters of sequence *A* and *j-th* letters of sequence *B*. Since $P(I,j)$ is the optimal solution we can assume that:

- ***if*** $A(i) = B(j)$ ***then*** $P(i+1, j+1) = P(I,j) + 1$
- ***else*** $P(I,j) = max\,(P(I,j-1), P(i-1,j))$

Therefore. we derive the following recursive equation:

- $P(i,-1) = P(-1,j) = 0$ *(terminal case)*
- $P(i,j) = \{$

$$\boldsymbol{if}\ \big(A(i) = B(i)\big)\ \boldsymbol{then}\ P(i-1,j-1) + 1,$$
$$\boldsymbol{else}\ max\big(P(i-1,j), P(i-1,j), P(i-1,j-1)\big)$$

$\}$

Bottom-Up Approach (Iterative)

We iterate through array elements 1-by-1 and calculate the value of the MCSS for every point of the sequences. An extra column and row are used to represent the terminal case. This approach calculates every possible combination of *i* and *j* (*N+1 x M+1* combinations).

```
//int[][] array = {0}
int iterate(int[][] array, string A, string B){
	for every i index of A: //(1..N)
		for every j index of B: //(1..M)
			if (A(i) == B(i)):
				array[i][j] = array[i-1][j-1] +1;
			else if (array[i][j-1] <= array[i-1][j]):
				array[i][j] = array[i-1][j];
			else:
				array[i][j] = array[i][j-1];
	return array[N-1][M-1];
}
```

Example

Following (**Table 1 and Table 2**) is the result of the iterative implementation for the given example.

Sequences		A	P	P	L	E	S
	0	0	0	0	0	0	0
P	0	0	1	1	1	1	1
I	0	0	1	1	1	1	1
N	0	0	1	1	1	1	1
E	0	0	1	1	1	2	2
A	0	1	1	1	1	2	2
P	0	1	2	2	2	2	2
P	0	1	2	3	3	3	3
L	0	1	2	3	4	4	4
E	0	1	2	3	4	5	5

Table 1 presents the solution of the Bottom-Up approach for the given example of the Most Common Sub Sequence problem

Top-Down Approach (Recursive)

Starting from the end of each sequence try to reverse engineer the MCSS by creating solution paths. Every time the corresponding letters of sequence *A* or *B* are matching, 1 possible solution path is created but when the letters differ 2 possible solution paths are created. There is an extra column and row to represent the terminal case. The array is initialized with -1 for the sake of **Memoization**. Recursion stops at the terminal cases, or when a value that is not '-1' is reached. Which means the sub-problem has already been solved by another recursion instance.

```
//int[][] array = {-1}
//recurse(array, A, B, N, M)
int recurse(int** array, string A, string B, int i, int j){
	if(i<0 || j<0):
		return 0;

	if(array[i+1][j+1] >= 0):
		return array[i+1][j+1];

	if(A(i) == B(j)):
		array[a+1][b+1] = recurse(array, A, B, i-1, j-1) +1;
	else:
		array[a+1][b+1] = max(recurse(array, A, B, i, j-1),recurse(array, A, B,
	i-1, j));

	return array[i+1][j+1];
}
```

Example

Following (**Table 2**) is the result of the recursive implementation for the given example

Sequence		A	P	P	L	E	S
	-1	-1	-1	-1	-1	-1	-1
P	-1	0	1	1	1	1	1
I	-1	0	1	1	1	1	1
N	-1	0	1	1	1	1	1
E	-1	0	1	1	1	2	2
A	-1	1	1	1	1	2	2
P	-1	-1	2	2	2	2	2
P	-1	-1	-1	3	3	3	3
L	-1	-1	-1	-1	4	4	4
E	-1	-1	-1	-1	-1	5	5

Table 2 presents the solution of the Top-Down approach for the example given fot the Most Common Sub Sequence problem

3.2 Experimental Comparison

Scenarios and Preparation

To evaluate the performance of the two implementations we will measure their execution time, and their memory usage with sequences of different lengths. The sequences will be randomly generated via a random sequence generator using the alphabet size variable *S*. A sequence can have up to *S* different characters.

We measure all input sizes with the following alphabet sizes: *1, 2, 6, 11, 16, 21, 26*. We want to observe whether the alphabet size has an impact on the performance of either implementation, and how each algorithm responds as the problem size (sequence length increases). This problem is a symmetrical problem. Therefore, by executing it with inputs A and B we can say we also executed inputs B and A. We could also experiment with sequences of different lengths. However, we test sequences of similar lengths (N=M).

We realise that this problem has a rather slow execution (a single run may take hours), therefore, to counter this we also consider smaller problem sizes that do not push the system to its limits.

Results and Discussion

Figure 1 depicts the effect of the input size to the execution time of both implementations. In this example we tested with an alphabet size of 26 *(A-Z)*. **Figure 2** presents the impact of the input size in total memory usage and % memory usage. These results were taken for the same alphabet size.

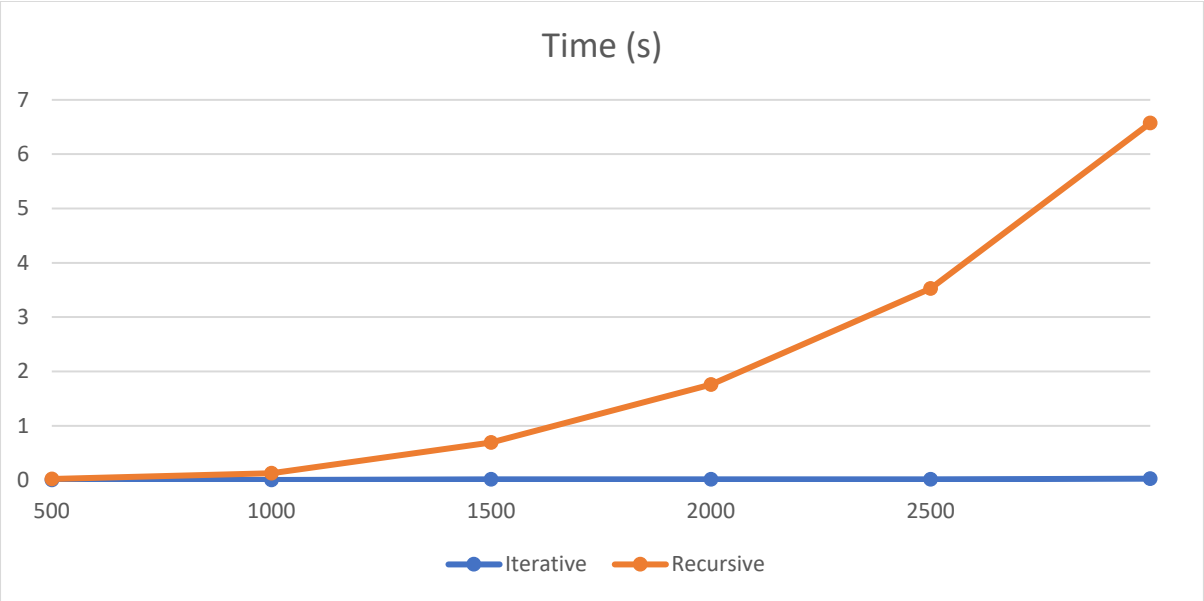

Figure 1 depicts the average execution time (in Seconds) in regard to problem size of the MCSS problem with alphabet size of 26 for both implementations

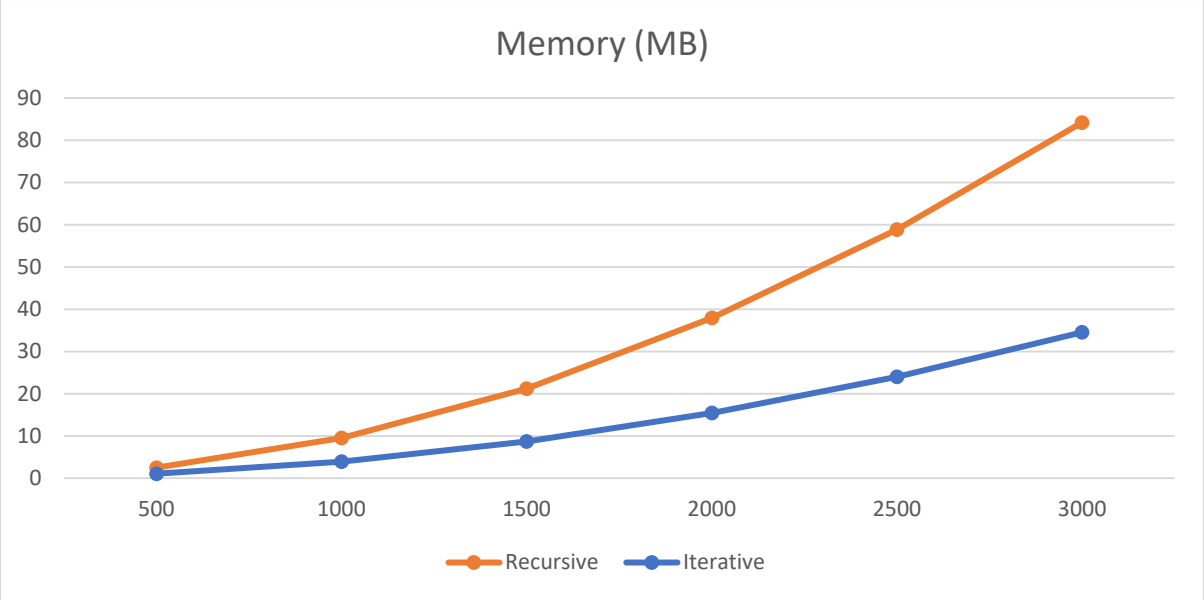

Figure 2 depicts the average memory usage in Megabytes and % Total memory of the MCSS problem with alphabet size of 26 for both implementations

As one can easily observe, our results for the Most Common Sub Sequence Problem (*S=26)* show that the execution time for the recursive approach yields much slower results while consuming more memory (**Figure 1** and **Figure 2**). As we mentioned earlier, running the implementation for bigger problem sizes increased the execution time exponentially as our results show, therefore we can't test for bigger input problems.

We will continue by investigating the effect of the Alphabet Size (*S)* on the performance of both implementations. **Figure 3** shows the effect of the alphabet size *S*, in the performance of the iterative approach. While **Figure 4** shows this effect for the recursive approach.

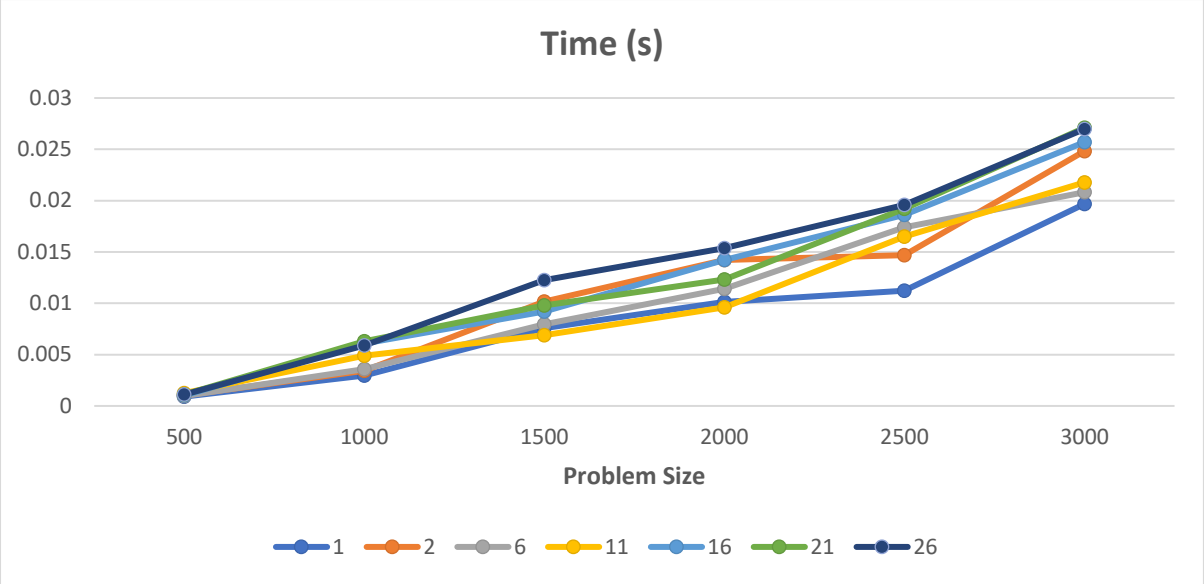

Figure 3 depicts the average execution time (in seconds) of the MCSS problem in regard to problem size for the Iterative approach for different Alphabet Sizes

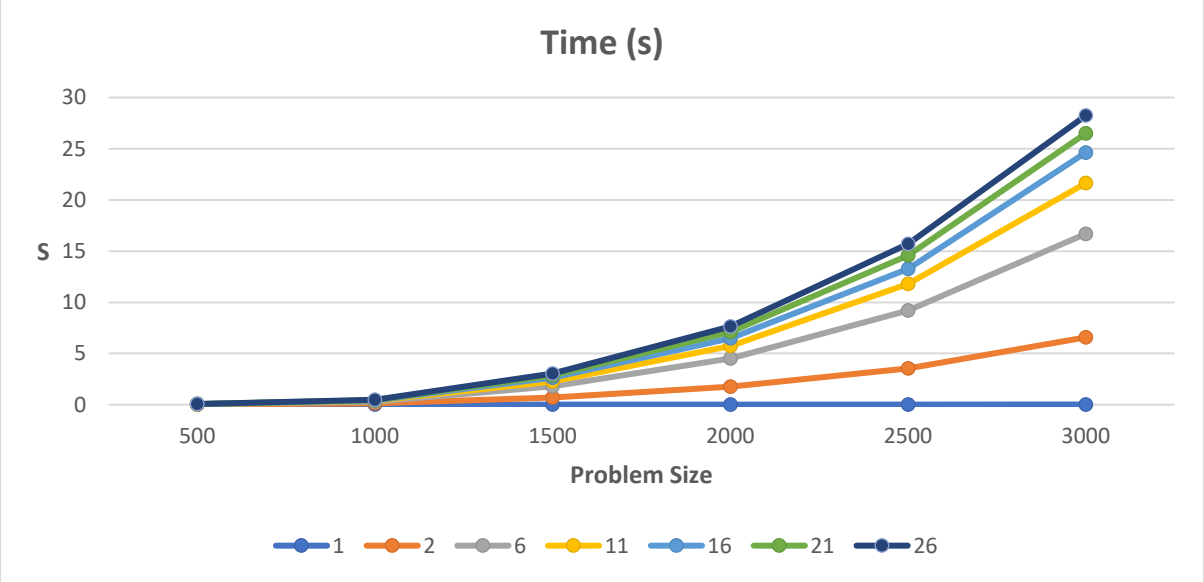

Figure 4 depicts the average execution time (in Seconds) of the MCSS problem in regard to problem size for the Recursive approach for different Alphabet Sizes

These results show that the alphabet size of the sequences affects the performance of the recursive approach while the discrepancies shown in the results of the iterative approach are within margin of error. We can conclude that the recursive approach benefits from small alphabet sizes. This is most likely because of the reduction of the branching factor of the recursions that occur when the alphabet size is lowered. By lowering the alphabet size we increase the similarity of the two sequences, therefore reducing the recursive calls. (identical characters result in 1 recursion call while differing characters result in 2).

Figure 5 and **Figure 6** present the effect of the alphabet size in memory usage for the iterative and recursive implementation approaches respectively.

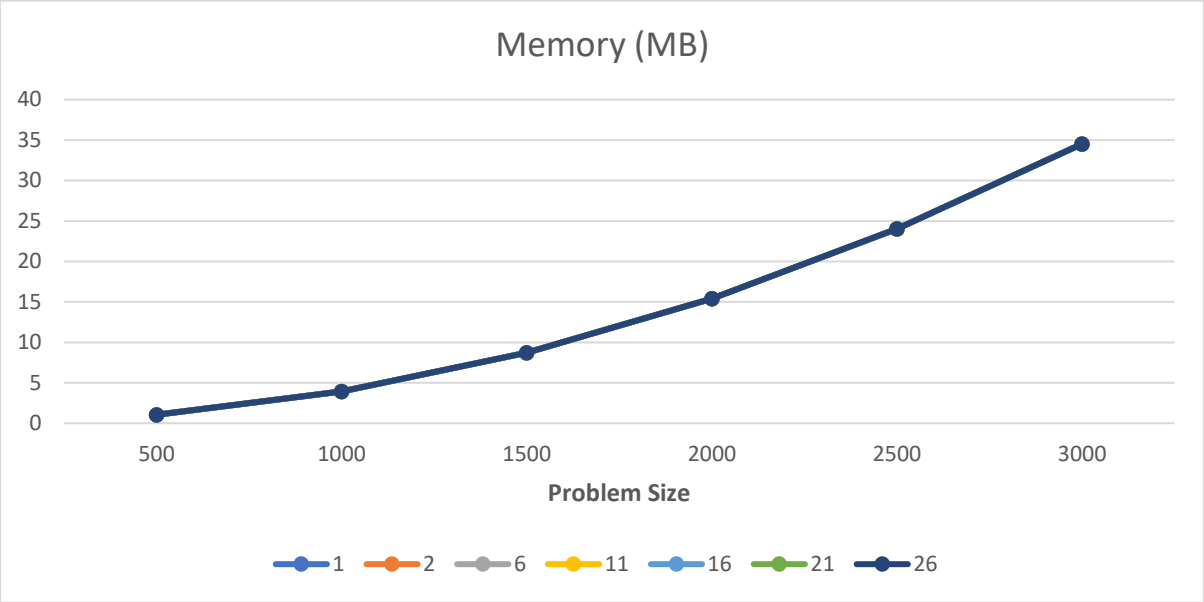

Figure 5 depicts the average memory usage (in Megabytes) of the MCSS problem in regard to Problem Size for the Iterative approach for different Alphabet Sizes

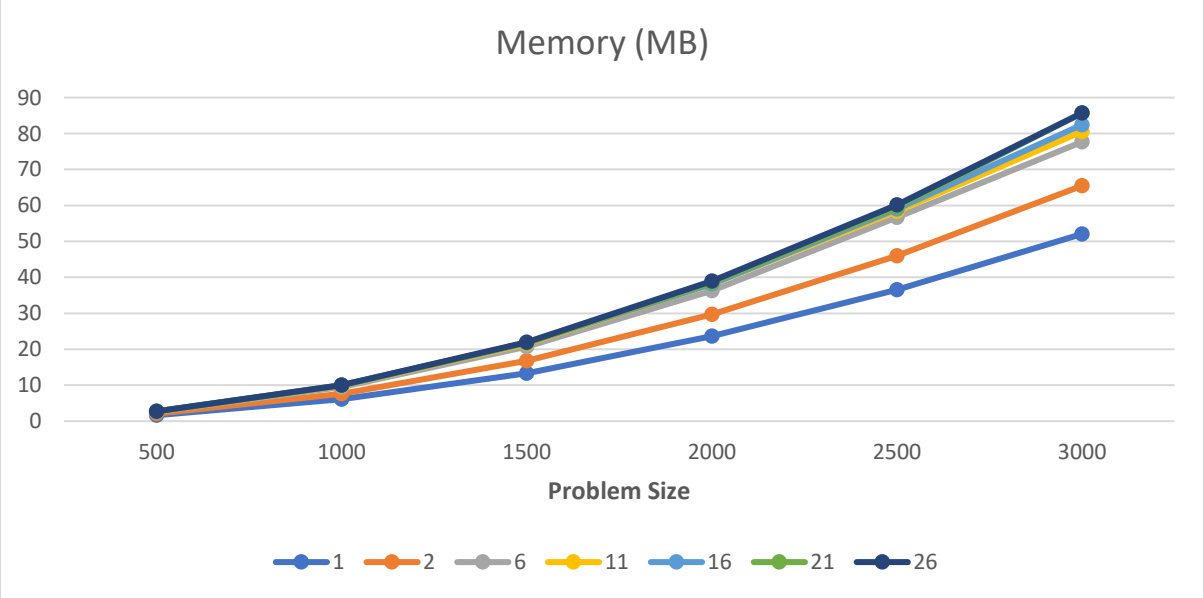

Figure 6 depicts the average memory usage (in Megabytes) of the MCSS problem in regard to Problem Size for the Recursive approach for different Alphabet Sizes

From our plots we observe that while the alphabet size has no effect in the memory usage of the Iterative approach, it greatly affects the memory usage of the Recursive approach. Bigger alphabet sizes increase the memory usage, lowering the implementations performance even further. This also supports our theory that the branching factor increases with the alphabet size, making the implementation slower and more memory demanding.

Chapter 4

Longest Increasing Sub Sequence

4.1 Problem Description and Dynamic Programming Solution

The problem of the Longest Increasing Sub-Sequence [1][2] is a problem that seeks the largest ascending (increasing) sub sequence from a given sequence of numbers. An increasing sub-sequence is a sequence where each of its values is followed by a greater value and is preceded by a smaller value. This problem is useful in the study of algorithms, mathematics and random matrix theory. This problem can be solved using a 2D array, however optimizations make it possible to solve it in a 1D array. We will examine both solutions.

Let's view an example. Given the sequence *(3,10,2,1,20)* the longest increasing subsequence is *(3,10,20)* with a length of 3.

4.2 2D Solution

Definitions

- Sequence *S* of length *N*
- Cache array *C* of size *N*
- Maximum Value *M*

Sub-Problem

Let $P(i,j)$ be the optimal solution for the *i-th* letter moving to the *j-th* letter, (letter j is the next number of the optimal sequence). If the value of i is bigger than the value of j then we can safely assume that *P(i,j)* is not part of the optimal subsequence, therefore we move to the next possibility *P(i,j+1).* Since *P(i,j)* is the optimal solution we can assume that:

- ***if*** $S(i) \geq S(j)$ ***then*** $P(i,j) = P(i,j+1) \quad (i < j)$
- ***otherwise*** $P(i,j) = max\ (\ P(i,j+1)\ ,\ P(j,j+1)\ )$

Bottom-Up Approach (Iterative)

```
//int array[] = {0]
int iterate(int[][] array, int[] S){
	//cache contains the previous best of each index
	int cache[N] = {0};
	for every index i of S: //(0..N-1)
		for every j from 0 to i+1: //(0..i+1)
			if(S[i] < S[j]):
				array[i][j] = cache[i]+1;
				if(cache[j]<array[i][j]) cache[j]=array[i][j];

	//find the value of the LISS in the cache
	int retVal = 0;
	for(int i=0; i<N; i++):
		retVal = max(retVal, cache[i]);
	return retVal+1;
```

Top-Down Approach (Recursive)

```
//int array[] = {-1]
int recurse(int[][] array, int[] S, int i, int j){
	if(curr >= N) return 0;
	//memoization
	if(array[i][j] >= 0) return array[i][j];
	//value of j is bigger, so we skip
	if(S[i] >= S[j]):
		array[i][j] = return recurse(array, S, i, j+1);
	else:
		array[i][j] = max(recurse(array, S, i, j+1), 1 + recurse(array, S, j,
		j+1));

	return array[i][j];
}
```

4.2.1 Experimental Comparison

Scenarios and Preparation

The problem requires a single number sequence. To test the performance of the approaches with different inputs we will first generate random number sequences. We use the *M* variable to determine the maximum value of the sequence, values start from 0 and the biggest possible number is equal to *M-1*. During our testing we found out that changing the variable *M* does not affect the performance of either approach.

Results and Discussion

Figure 7 shows the impact of the input size of the problem in execution time (seconds). While **Figure 8** depicts the impact of the input size in memory usage (Gigabytes) and % total memory usage. To acquire these results, we used *M* of *1000*.

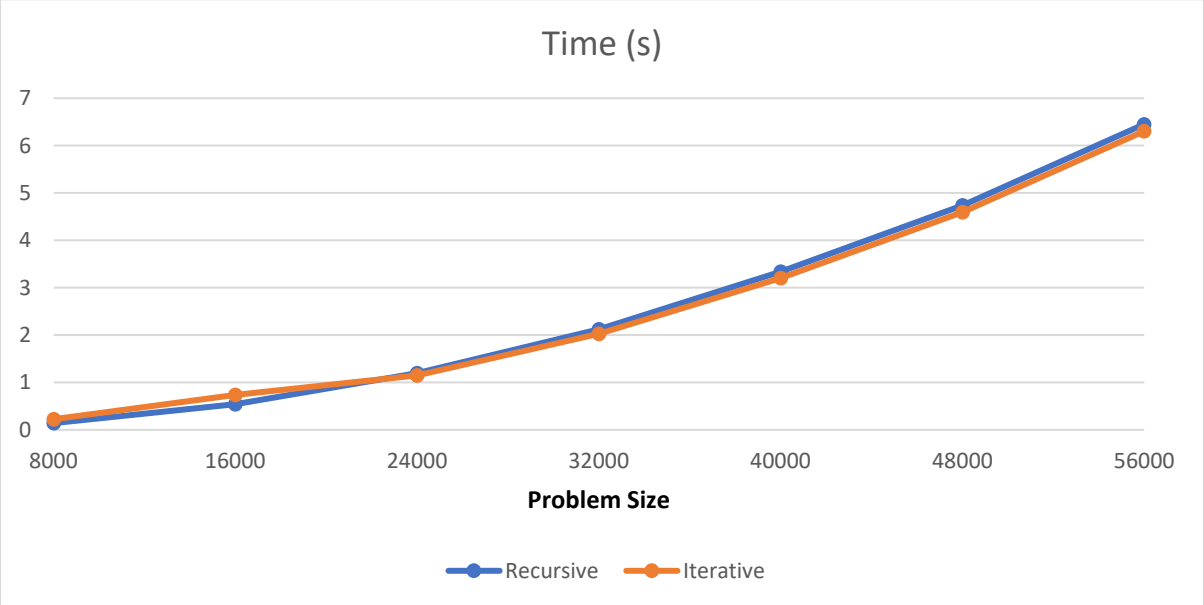

Figure 7 depicts the average execution time (in Seconds) of the LISS 2D problem for both implementations

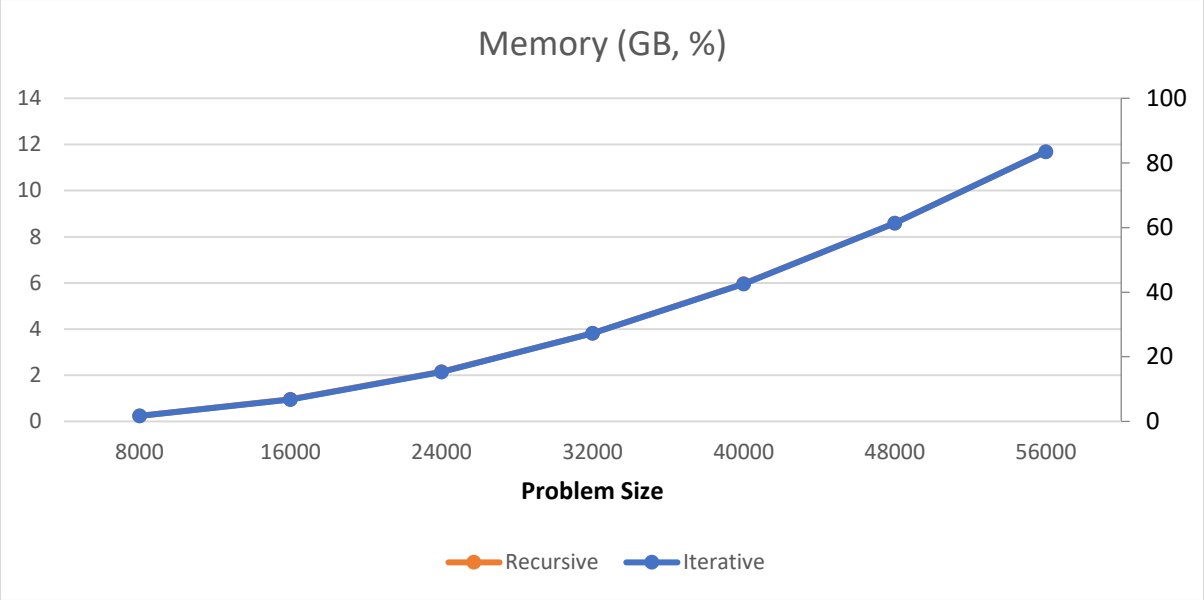

Figure 8 depicts the average memory usage in Gigabytes and % Total memory of the LISS 2D problem for both implementations

The extra consumption of the stack by the recursive implementation is minimal compared to the total heap consumption. The small discrepancies of the execution time of the two implementations are within error margin. Therefore, we can conclude that both approaches behave in a similar manner, and both measurements (time and memory) seem to perform equally well.

4.3 1D Solution

This problem is a continuation of the LISS problem solved using a 2D array, however optimized to be solved using a 1D array. This means that the problem can be solved for much greater sizes while consuming much less memory. In more detail, the memory usage is reduced by an exponential factor of 1 (reduction of dimensionality by 1).

Bottom-Up Approach (Iterative)

```
//int[] array = {1}
int iterate(int[] S, int[] array)
	for every index i in S: //0..N
		for every index j from 0 to i: //0..i
			if ( S[i] > S[j] && array[i] < array[j] + 1)
				array[i] = array[j] + 1;

	int max =INT32_MIN, index=-1;
		for every number in S
			if(array[i] > max):
				max = array[i];
				index = S[i];

	return array[index];
```

Top-Down Approach (Recursive)

```
//int[] array = {-1}
int recurse(int[] S, int[] array, int curr){
	//end case
	if(curr >= N) return 0;
	//memoization
	if(array[curr] > 0) return array[curr];

	int result = 1;
	for every node i to curr: //0..curr
		if(S[curr] > S[i])
			result = max(result, 1 + recurse(S, array, i));
	array[curr] = result;
	return result;
```

4.3.1 Experimental Comparison

Scenarios and Preparation

To compare the results**,** we acquired by using the 2D solution with the 1D solution, we use the same input parameters. These parameters are N (sequence length) and *M* (amount of unique values). Similarly**,** to the 2D solution, our results show that the variable *M* does not affect performance.

Results and Discussion

Figure 9 depicts the effect of the problem input size in execution time (seconds), and **Figure 10** depicts the effect it has on memory usage (Kilobytes) and % total memory usage. The variable *M* used was *1000*.

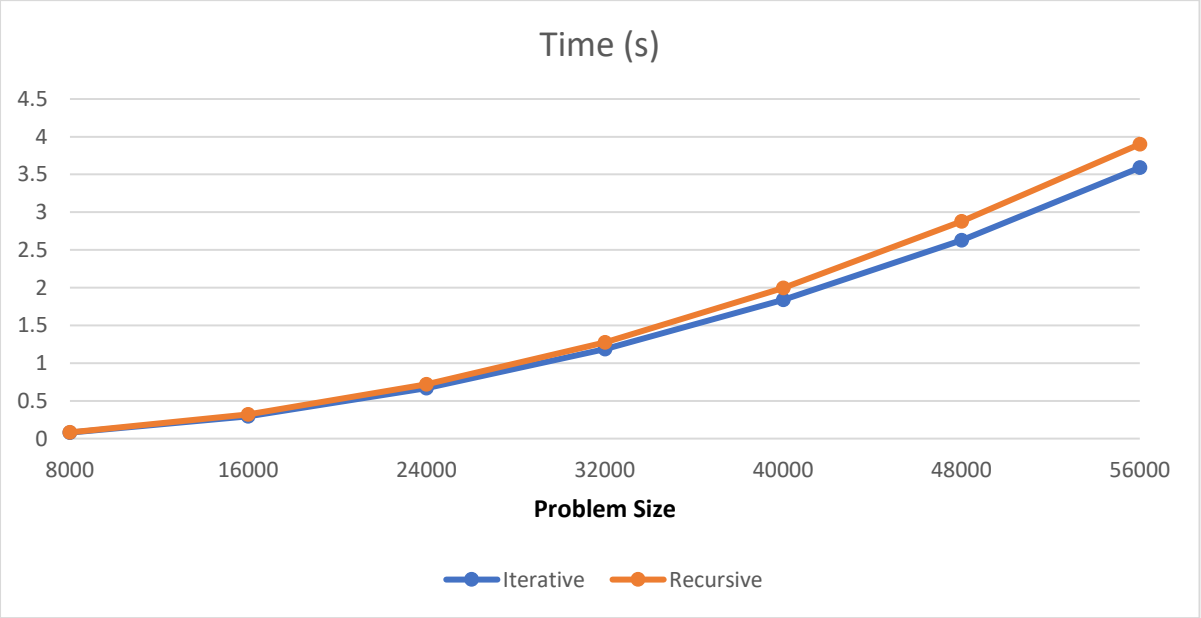

Figure 9 depicts the average execution time (in Seconds) of the LISS 1D problem for both implementations

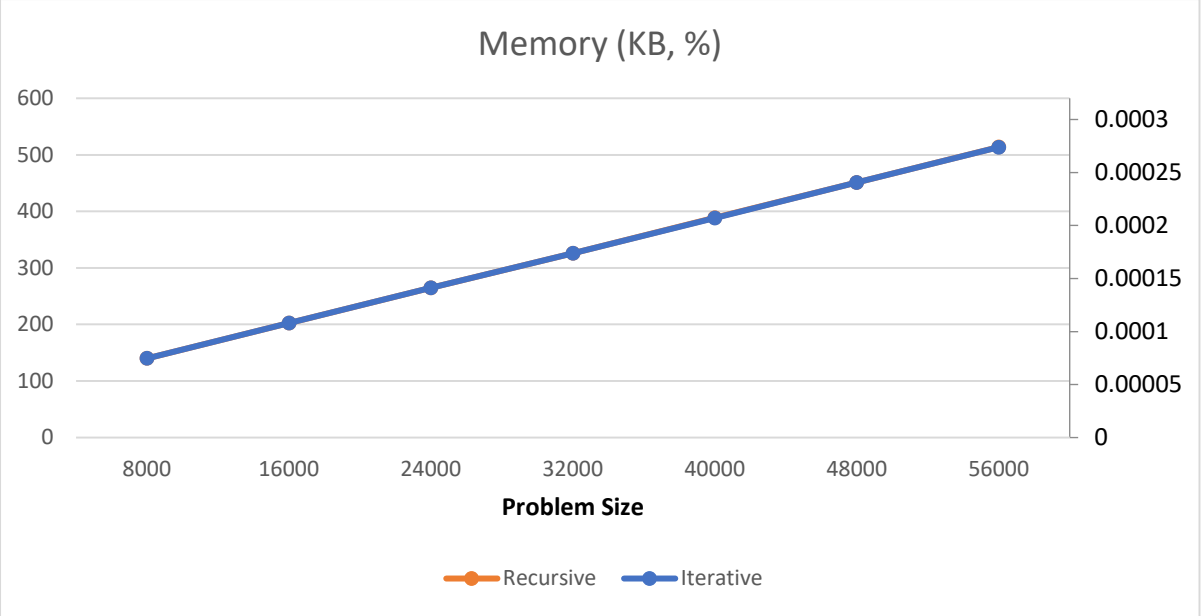

Figure 10 depicts the average memory usage in Kilobytes and % Total memory of the LISS 1D problem for both implementations

After viewing these results, we can see that this solution consumes very little memory ($< 0.001\%$), making it very efficient, with both approaches providing similar results. However, comparing the execution times of the two implementations we can also see that the iterative approach is pulls ahead with a slight lead.

4.4 Comparison between 1D and 2D Solutions

The results of the two solutions show that the 1D solution is the clear winner in both aspects. It is very efficient when regarding the memory usage and it is faster than the 2D solution. The iterative approach of the 1D solution provides the best results overall with a small lead in execution time over the recursive implementation.

Chapter 5

Chain Matrix Multiplication

5.1 Problem Description and Dynamic Programming Solution

Let two matrices *A* and *B* of size *NxI* and *JxM* respectively. The multiplication (dot-product) of these 2 matrices requires that *I* is equal to *J* and results in a new matrix of size *NxM* [1][2]. This process requires *NxIxM* operations, this is also set to be the cost of the multiplication. When multiple matrices have to be multiplied, the order in which these matrices are multiplied has an effect in the resulting cost. The aim is to minimize this cost by choosing the optimal order in which these matrices should be multiplied.

Definitions

- Set *S* of *N* Matrices *(WixHi)*
- *(S(i).w, S(i).h)* are the dimensions of the *i-th* matrix in the list, where '*w'* is its width and '*h'* is its height.
- *S* contains the matrices in order; therefore, we can assume, *S(i).w = S(i+1).h* since the matrix multiplication operation requires that the width of the preceding array is equal to the height of the following array. Now assume list *L*. We can assume *L* is a list of *N+1* values, and *N(i)* is equal to *S(i)*.w and *N(i)* is also equal to *S(i+1).h*.
- Maximum value of matrix width and height *M*

Example

Table 3 contains the result of both implementation approaches with the given matrices and generated list L.

Matrices*: [5x10], [10x12], [12x8], [8x7], [7x11]* (5) ➔ L: *[5,10,12,8,7,11]* (6)

Matrix	0	1	2	3	4
0	0	600	1080	1360	1745
1	0	0	960	1512	2282
2	0	0	0	672	1596
3	0	0	0	0	616

Table 3 presents an example with each solution for the Chain Matrix Multiplication problem

Sub-Problem

- *Let P(i,j)* be the optimal solution for matrices *i* up to *j*
- *P(i,i) = 0* is the terminal case.
- The actual problem is solved when *P(0,N)* is solved.

We use the step variable to calculate using a pivot, therefore:

$$P(i,k) = \min_{i \le j < k} (P(i,j) + P(j+1,k) + S(i).w * S(j).h * S(k).h)$$

We use a 'step' variable, let 'step' be *s* (or *k)*. We start solving *P(i, i+s)* for every *i*, increasing *s* by 1 after each iteration until the problem is solved. The idea is that since *P(i,i+s)* is known, *P(i,i+s+1)* can be calculated in *O(1*).

Bottom-Up Approach (Iterative)

```
int iterate(int[][] array, int[] L){
	int indexStart, indexEnd, val;
	for(int step=1; step<N; step++):
		for(indexStart=0; indexStart<N - step -1; indexStart++):
			indexEnd = indexStart + step;
			int min = infinity;
			for(int k=indexStart; k<indexEnd; k++):
				val = array[indexStart][k] + array[k+1][indexEnd] +
				L[indexStart] * L[k+1] * L[indexEnd+1];
				if(val < min):
					min = val;
			array[indexStart][indexEnd] = min;

	return array[0][N-1];
}
```

Top-Down Approach (Recursive)

```
int recurse(int[][] array, int[] L, int indexStart, int indexEnd){
	if (indexStart == indexEnd)
		return 0;
	if(array[indexStart][indexEnd] != 0)
		return array[indexStart][indexEnd];

	int min = infinity;
	for(int k=indexStart; k<indexEnd; k++){
	int val = recurse(array, L, indexStart, k) + recurse(array, L, k+1, indexEnd) +
L[indexStart] * L[k+1] * L[indexEnd+1];
	if(val < min)
		min = val;

	int retVal = min;
	array[indexStart][indexEnd] = retVal;
	return retVal;
}
```

5.2 Experimental Comparison

Scenarios and Preparation

To test the efficiency and performance of each approach we will generate a random list '*L'* of size *N+1* values and the specified maximum value of *M*. There can be up to *M* unique values within the list *L* (from *0 to M-1*). The Chain Matrix Multiplication is the problem tested with the most required computation needed since it has a complexity *of* $O(N^3)$, therefore we will take measurements of smaller problem sizes *N*. The variable *M* does not affect the performance of either implementation.

Results and Discussion

Figure 11 depicts the impact of the problem input size in execution time (seconds) while **Figure 12** shows the memory usage in Megabytes and % total memory usage in regards to the input size. The variable *M* was set to *20*.

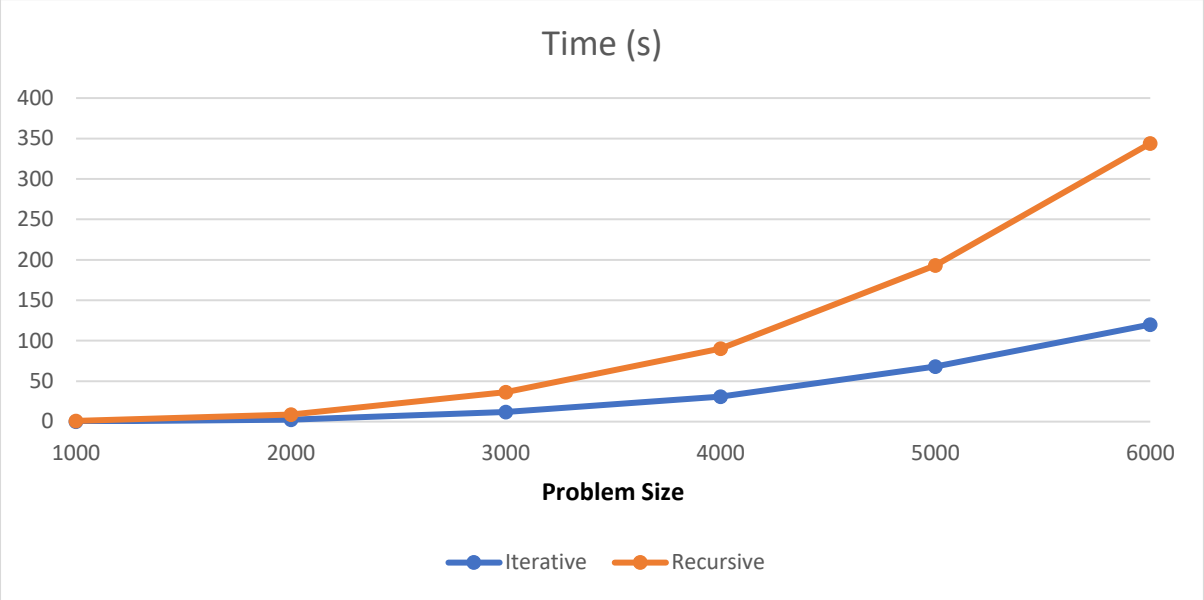

Figure 11 depicts the average execution time of the Chain Matrix Multiplication problem for both implementations

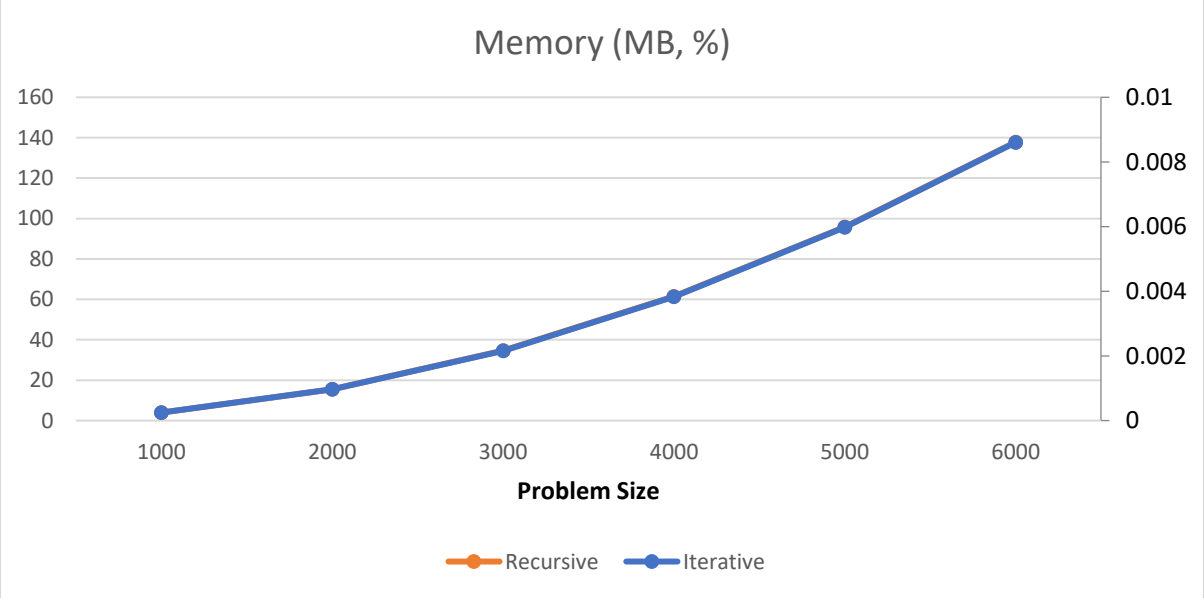

Figure 12 depicts the average memory usage in Megabytes and % Total memory of the Chain Matrix Multiplication problem for both implementations

From these results we can observe that while the memory usage is similar for both implementations, the recursive approach falls behind when it comes to execution time. The iterative approach is the much better choice. The amount of extra stack usage of the recursive approach is not significant compared to the total heap usage.

Chapter 6

0-1 Knapsack

6.1 Problem Description and Dynamic Programming Solution

The Knapsack problem [3] is to find the selection of items within a certain weight limit *C*, with the most value. Each item has its own cost (weight) and worth values. This problem is very useful, it is used to capture the customer values and the discrete characteristics of loads. The objective is to maximize the customer values within a given supply.

For the '0-1' version of the problem we assume an item can either be completely inside the sack or completely out. We also assume that weights are natural numbers. The idea is that we create *C* sacks, and then we choose an item arbitrarily and attempt to place the item in these sacks to maximize the sacks value. The choice is to either place the item in the sack or not. When all items have been processed for all *C* sacks the solution will be found.

Definitions

- Weight limit *C* (capacity of the knapsack)
- List of items *N*, each item has its own weight *Wi* and value *Vi*
- *C* sacks with capacity *Ci*, sacks begin with capacity *0* up to *C*.
- Maximum weight *Wmax* and value *Vmax*

Sub-Problem

- Let *P(i,j)* be the optimal value for all items up to the *i-th* item in the *j-th* sack (sack with capacity *j*).
- An item can either be placed in the *j-th* sack or not, it can only be placed if it can fit inside the sack.

Therefore, we derive the following equation:

$$P(i,j) = 0 \quad \textbf{if } i = 0 \ \textbf{ or } \ \mathrm{j} = 0 \ \textbf{ (terminal cases)}$$

$$P(i,j) = \max(\mathrm{Vi} + \mathrm{P(i-1,j)}, \mathrm{P(i-1,j)}) \quad \textbf{if } \ Wi \leq i \textbf{ (item can fit in sack)}$$

$$P(i,j) = \mathrm{P(i-1,j)} \quad \textbf{otherwise}$$

Example

The following table (**Table 4**) shows a randomly generated list of items that were generated with the following parameters:

Sack capacity C: 5 Items: 8 (weight: 1-3, value: 1-30)

Item	Weight	Value
1	3	4
2	3	13
3	2	15
4	1	26
5	3	2
6	2	5
7	3	12
8	1	12

Table 4 presents a randomly generated set of items that serves as our example

We will view the results of both implementations later.

Bottom-Up Approach (Iterative)

```
//int array[N+1][C+1] = {0}
int iterate(Item[] items, int[][] weights){
	int itemIndex, weightIndex;

	// Build table K[][] in bottom up manner
	for (every index of items) : //1..N
		for (every index of weights): //1..C
			Item* item = items[itemIndex];
			// End case
			if (itemIndex == 0 || weightIndex == 0)
				weights[itemIndex][weightIndex] = 0;
			else if (item->weight <= weightIndex)
				weights[itemIndex][weightIndex] = max(item->value +
			weights[itemIndex][weightIndex - item->weight],
			weights[itemIndex][weightIndex]);
			else
				weights[itemIndex][weightIndex] =
			weights[itemIndex][weightIndex];

	return weights[N][C];
}
```

We begin from item with index 1 and attempt to place the item in the sacks. Sack values are initialized to 0, and we strive to find the combination of items for each sack that maximizes its value.

Item \ Sack		1	2	3	4	5
	0	0	0	0	0	0
1	0	0	0	4	4	4
2	0	0	0	13	13	13
3	0	0	15	15	15	28
4	0	26	26	41	41	41
5	0	26	26	41	41	41
6	0	26	26	41	41	46
7	0	26	26	41	41	46
8	0	26	26	41	53	53

*Table **5** presents the solution of the Bottom-Up approach for the example given of the 0-1 Knapsack problem*

Top-Down Approach (Recursive)

```
//int array[N+1][C+1] = {-1}
int recurse(Item** items, int** weights, int itemIndex, int weightIndex){
	if(itemIndex == 0):
		return 0;

	if (weights[itemIndex][weightIndex] != -1):
		return weights[itemIndex][weightIndex];

	Item* item = items[itemIndex];
	if(weightIndex < item->weight):
		weights[itemIndex][weightIndex] = recurse(items, weights, itemIndex,
	weightIndex);
		return weights[itemIndex+1][weightIndex];
	else
		weights[itemIndex+1][weightIndex] = max(recurse(items, weights,
	itemIndex-1, weightIndex), recurse(items, weights, itemIndex-1, weightIndex -
	item->weight) + item->value);
	return  weights[itemIndex+1][weightIndex];
}
```

Item \ Sack		1	2	3	4	5
	-1	-1	-1	-1	-1	-1
1	0	0	0	4	4	4
2	0	0	0	13	13	13
3	0	0	15	15	15	28
4	0	26	26	41	41	41
5	0	26	26	41	41	41
6	-1	26	26	-1	41	46
7	-1	-1	-1	-1	41	46
8	-1	-1	-1	-1	-1	53

Table 6 presents the solution of the Top-Down approach for the example given of the 0-1 Knapsack problem

6.2 Experimental Comparison

Scenarios and Preparation

To test the efficiency of each approach we will generate a random list of *N* items, each with weight in the range of [*1*,*wMax*] and value within the range of [*1*,*vMax*]. We want to investigate if these parameters affect the performance of either approach. We realised that the *vMax* and *wMax* variables do not affect performance. However, the Sack size *C* does as expected.

Results and Discussion

Figure 13 shows the effect of the problem input size in execution time (seconds) and **Figure 14** shows the memory consumption in Gigabytes and % total memory for the same input sizes. For these measurements *C* was set to be 25000.

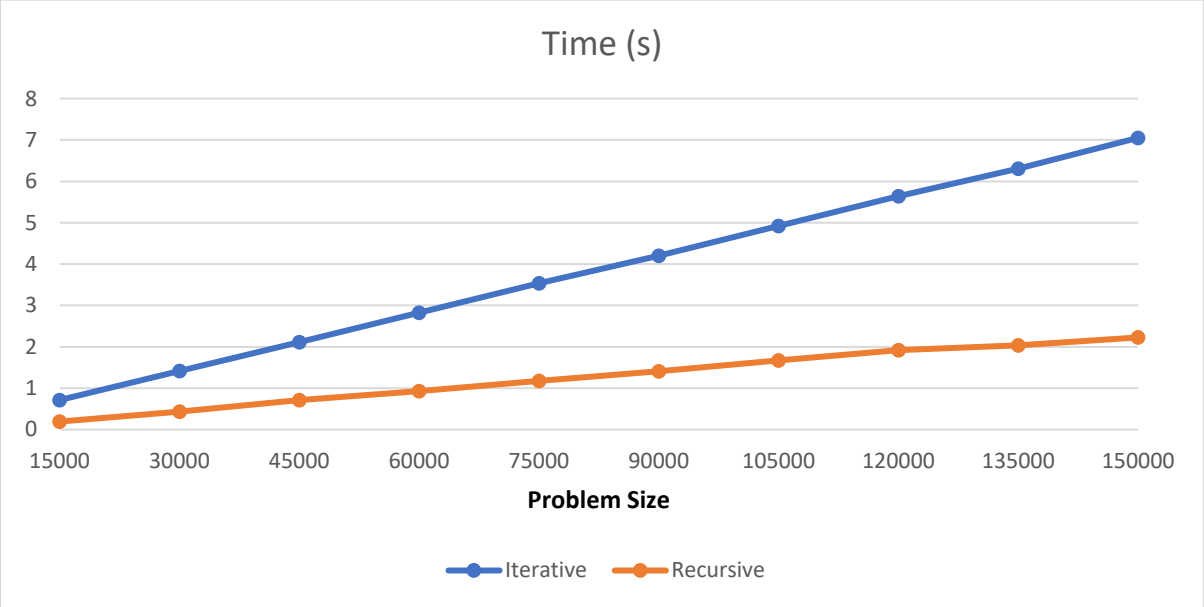

Figure 13 depicts the execution time (in Seconds) of the 0-1 Knapsack problem for both implementations

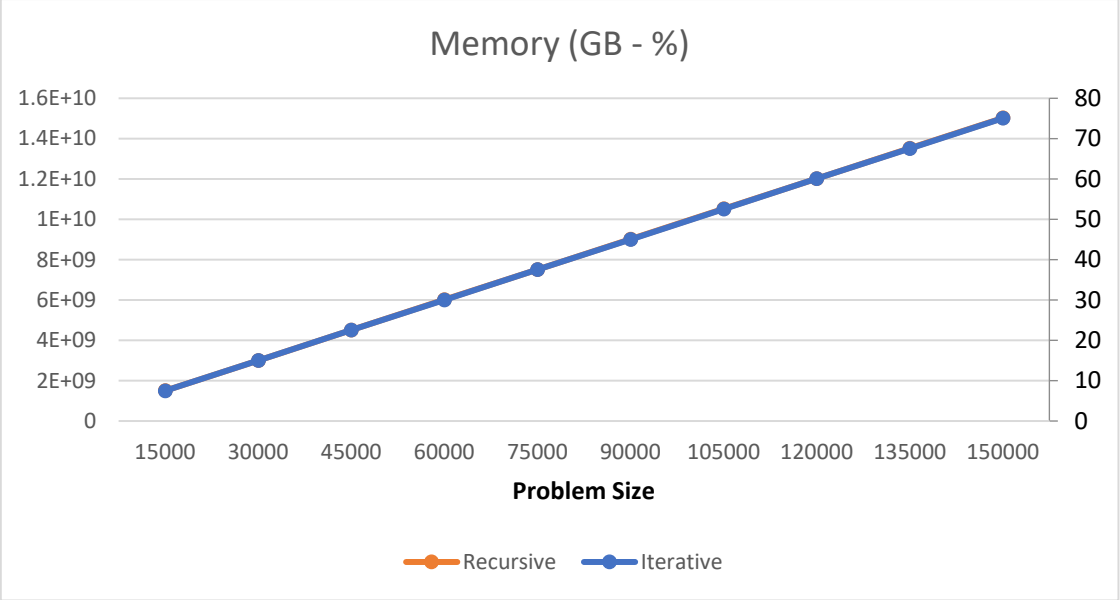

Figure 14 depicts the average memory usage in Megabytes and % Total memory of the 0-1 Knapsack problem for both implementations

Our results show that the memory usage seems to be very similar in both approaches while the execution is much slower in the recursive approach yielding bigger execution times. We can see that the memory usage follows a linear growth in regard to the problem size. This growth is also present in the execution time.

We expect the *C* variable to have an impact on the performance of both implementations. As a problem on arrays, 0-1 Knapsack creates a solution on a table of *NxC*. Therefore, we will record our measurements with different *C* variables to measure the performance of both implementations.

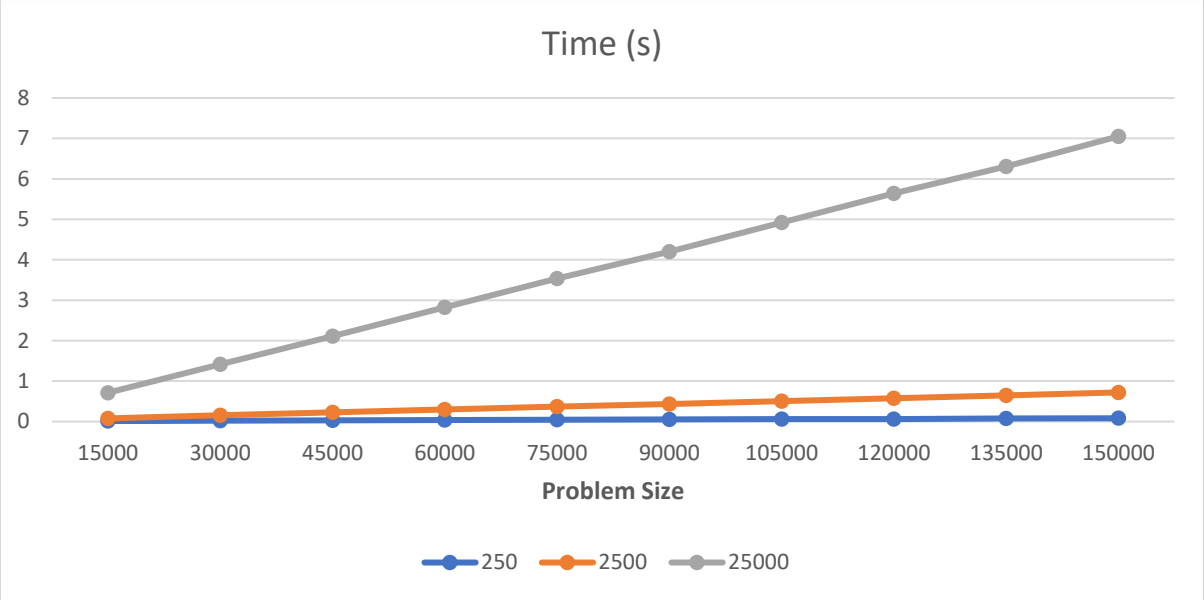

Figure 15 depicts the average in execution time (in Seconds) of the 0-1 Knapsack problem for the Iterative approach with measurements of multiple C sizes.

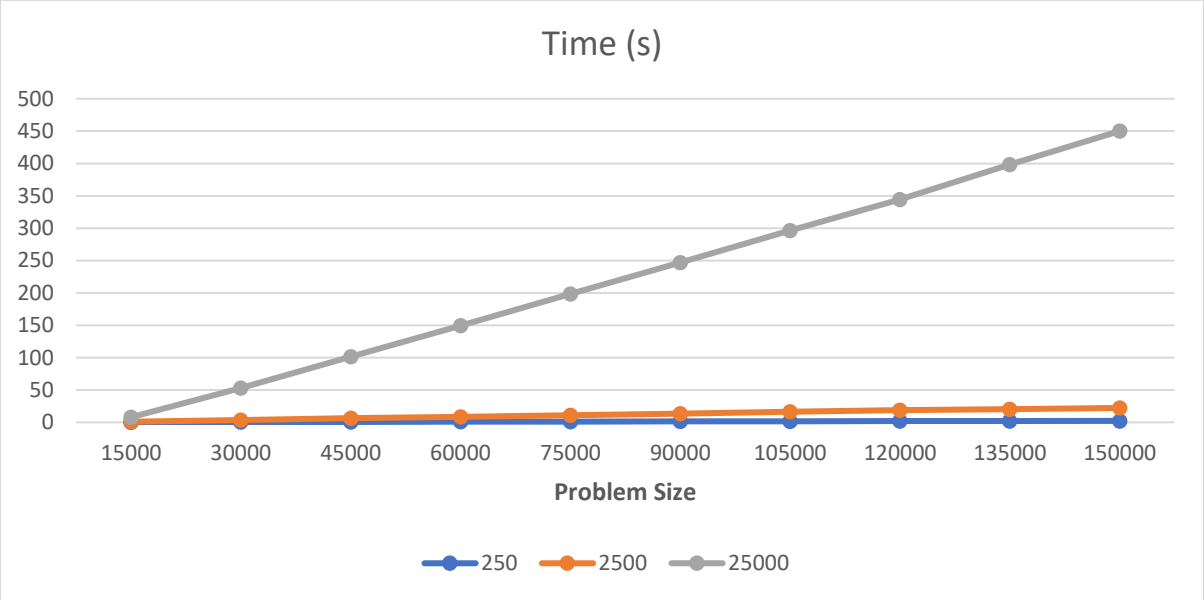

Figure 16 depicts the average in execution time (in Seconds) of the 0-1 Knapsack problem for the Recursive approach with measurements of multiple C sizes.

Figure 17 and **Figure 18** depict the memory usage of the iterative and recursive implementations respectively in regard to changes in the *C* variable. This is an expected result since the input of the problem depends both on the problem s*ize N* and the sack capacity *C*.

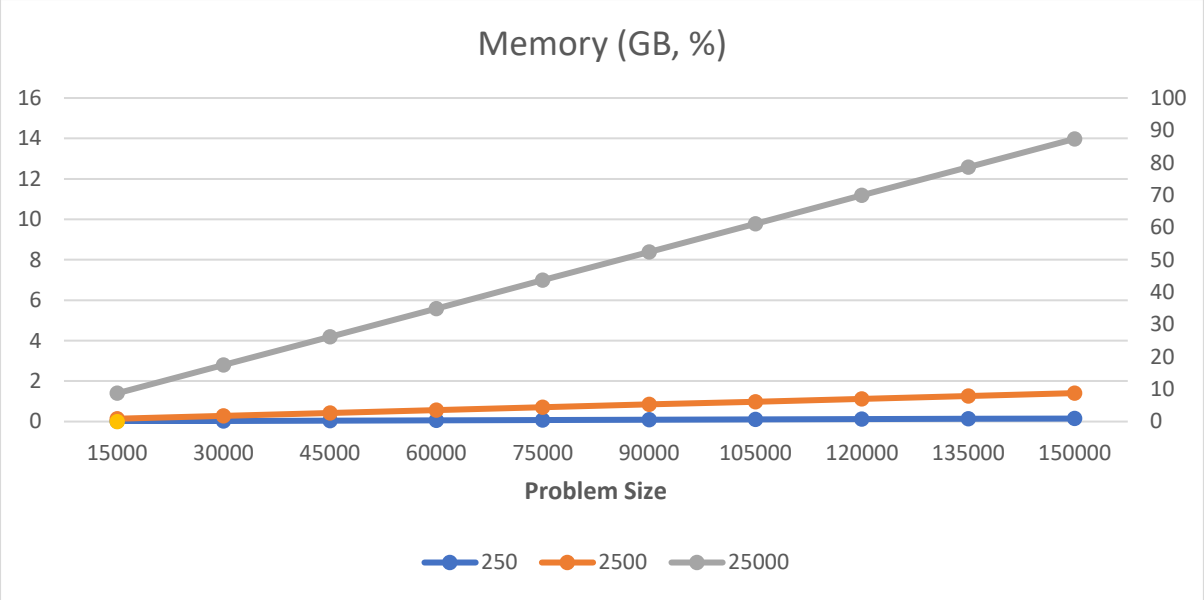

Figure 17 depicts the average memory usage in Gigabytes and % Total memory of the 0-1 Knapsack problem for the Iterative approach with measurements of multiple C sizes.

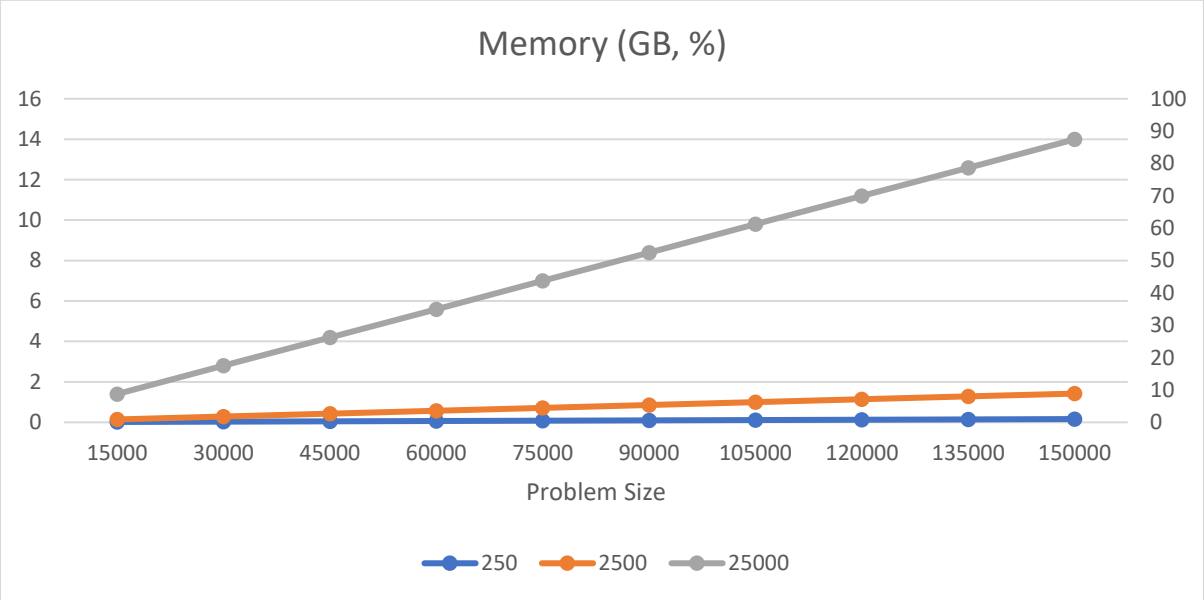

Figure 18 depicts the average memory usage in Gigabytes and % Total memory of the 0-1 Knapsack problem for the Recursive approach with measurements of multiple C sizes.

Our results show that increasing *C* we see an equal increase in memory usage for both implementations. However, an increase in *C* does not favour any implantation approach in particular since the demand in memory grows equally in both cases. Execution times tell a similar story, where an increase in *C* yields a proportionally equal increase in execution time for both implementation approaches.

Chapter 7

Dijkstra's Shortest Path

6.3 Problem Description and Dynamic Programming Solution

Finding the shortest path (and therefore the smallest distance) between two nodes in a graph is equally useful and important. It is useful in multiple fields, from general research to AI in game development. Dijkstra's algorithm does exactly that, given a Graph *G*, a pair of nodes, namely the starting point and the end point, it finds the shortest path between the two (2) nodes from inside the given graph. There's a debate as to where this algorithm should be considered a Dynamic Programming algorithm or a Greedy algorithm, but for the sake of this study we will consider it a DP algorithm. Variations of this algorithm exist that may yield better results, one of these variations is the A* (A-star) algorithm that introduces a cost variable and a heuristic variable. These variables are used to make more informed choices contrary to Dijkstra's approach, this is why A* is considered a greedier algorithm.

As aforementioned, Dijkstra's algorithm finds the closest path between two nodes inside a graph, to do this it starts from the given start node and it iteratively explores all its adjacent nodes until the end node is reached. To ensure that the optimal answer (or in other words, the shortest path between the two nodes) is found, the node explored in every iteration has to be the closest, unexplored node to the starting node. The algorithm can end before looping through all the nodes if the end point has been visited once because of this detail, as this detail provides the optimal answer of each state / sub-problem.

Definitions

- Graph *G* of *N* nodes, with density *D* and *wMax*
- Nodes *S* (‘Start node’) and node *T* (‘Finish’ / ‘End node’)
- ‘Visited’ array *V* of size *N*
- ‘Cache’ array *C* of size *N*

Sub-Problem

- $C(S,j)$ is the shortest distance between start node S and node j.
- Solution at: $C(S,T)$
- Recursive function for intermediate node i

Therefore:

- $C(S,j) = min\ (\ C(S-\{j\},i)\ ) + G(i,j)$

Algorithm

```
int Dijkstra(int[][] G, int[] cache, int s, int t)
	//visited array contains explored nodes
	bool visited[PROBLEM_SIZE] = false;
	visited[s] = true;

	// the ‘index’ variable represents the node being explored (closest unexplored
	// node to s), while the ‘min’ variable represents the distance of the closest
	// unexplored node (index)
	int index, min;
	for every node i in G:
	//find closest node index

		//start with node s
		if(i == s)
			index = s;
			min = cache[index];
		//for all other nodes
		else
			for every node j in G:
				if(visited[j]) continue;
				if(cache[j] < min) {
					index = j;
					min = cache[j];

	//explore nodes from index
		for every node j in G:
			if nodes i and j are connected:
				if cache[j] > (cache[index] + graph[index][j]):
					array[j] = array[index] + graph[index][j];
					path[j] = index;
		visited[index] = true;

	//return the cached distance of the end point
	return array[t];
```

Examples

In the following example we present a randomly generated graph *G* (**Figure 19** & **Table 7**) of 5 nodes, and the state array (**Table 8**) that occurs from our implementation.

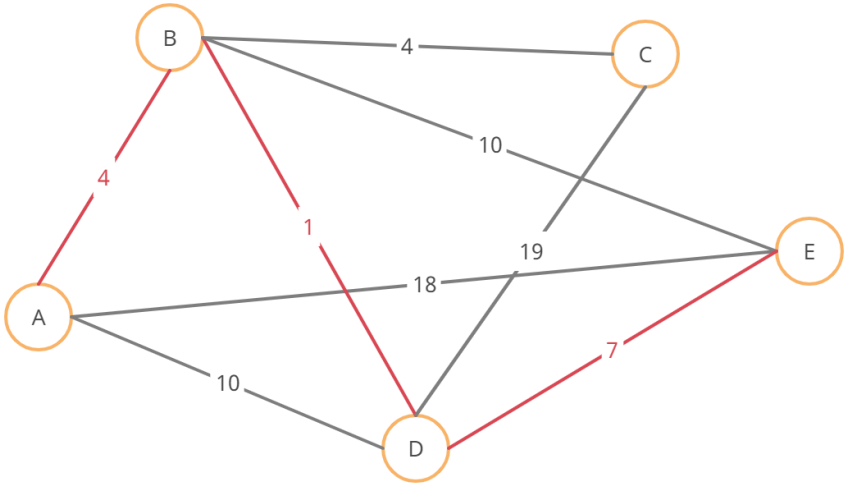

Figure 19 represents the example graph G

Node	A	B	C	D	E
A	-1	4	-1	10	18
B	4	-1	4	1	8
C	-1	4	-1	19	1
D	10	1	19	-1	7
E	18	8	1	7	-1

Table 7 represents the graph G as an array

Iteration \ Node	A	B	C	D	E
0 – A	- \ -	**4 \ A**	- \ -	**10 \ A**	**18 \ A**
1 – B	- \ -	4 \ A	**8 \ B**	**5 \ B**	**14 \ B**
2 – D	- \ -	4 \ A	8 \ B	**5 \ B**	**12 \ D**
3 – C	- \ -	4 \ A	8 \ B	5 \ B	12 \ D
4 – E	- \ -	4 \ A	8 \ B	5 \ B	12 \ D

Table 8 presents the solution for the example given of the Dijkstra's shortest path problem

6.4 Experimental Comparison

Scenarios and Preparations

Since the input of the problem is a graph, we can compare the performance of the 2 approaches with different graphs. Firstly, we create N nodes, then we iterate through all the nodes to connect them to each other. We use the density variable D to determine the maximum number of neighbours (adjacent nodes) a node can have. For each node we choose a random amount of connected nodes such Hi, such that $0 < Hi \leq D$, Then we connect every node to Hi other nodes. This way we ensure the graph is connected, and that the bigger the density, the more the average connections of each node. Every time we connect two nodes we generate a random weight value for their connection. During our experiments we realised that the weight of the connections of the graph does not affect the performance of any of the approaches.

Results and Discussion

Figure 20 and **Figure 21** show the effect of the problem input size in execution time (seconds) and memory usage in Gigabytes and % total memory, respectively. The graphs generated for this data had a density of 20 and a maximum weight size of 20 as well.

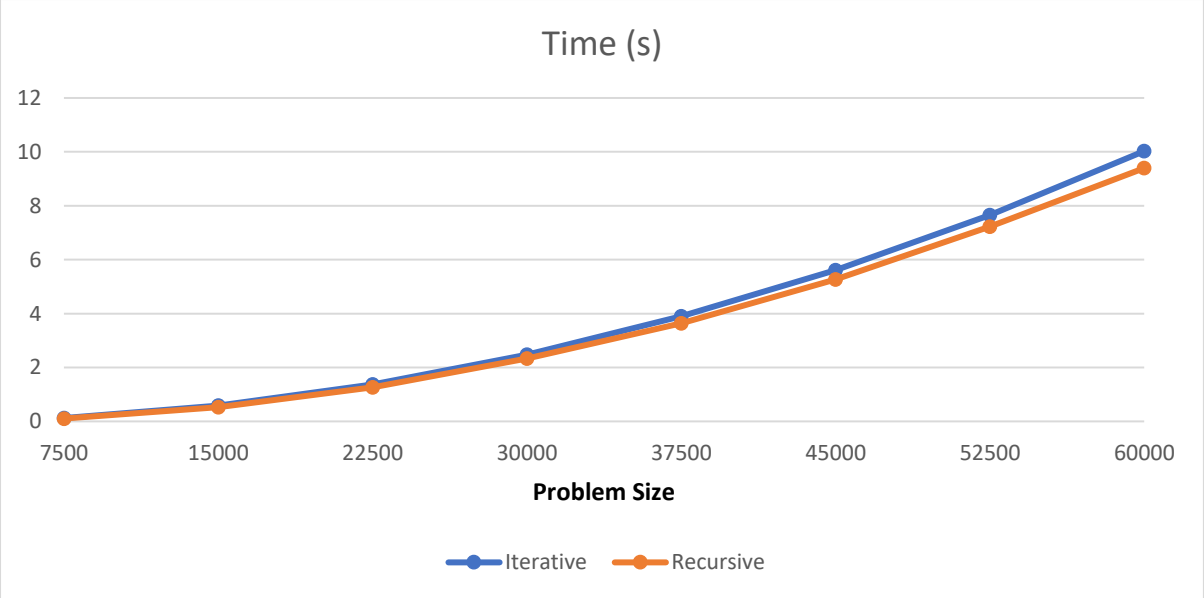

Figure 20 depicts the average execution time (in Seconds) of the Dijkstra's shortest path problem for both implementations

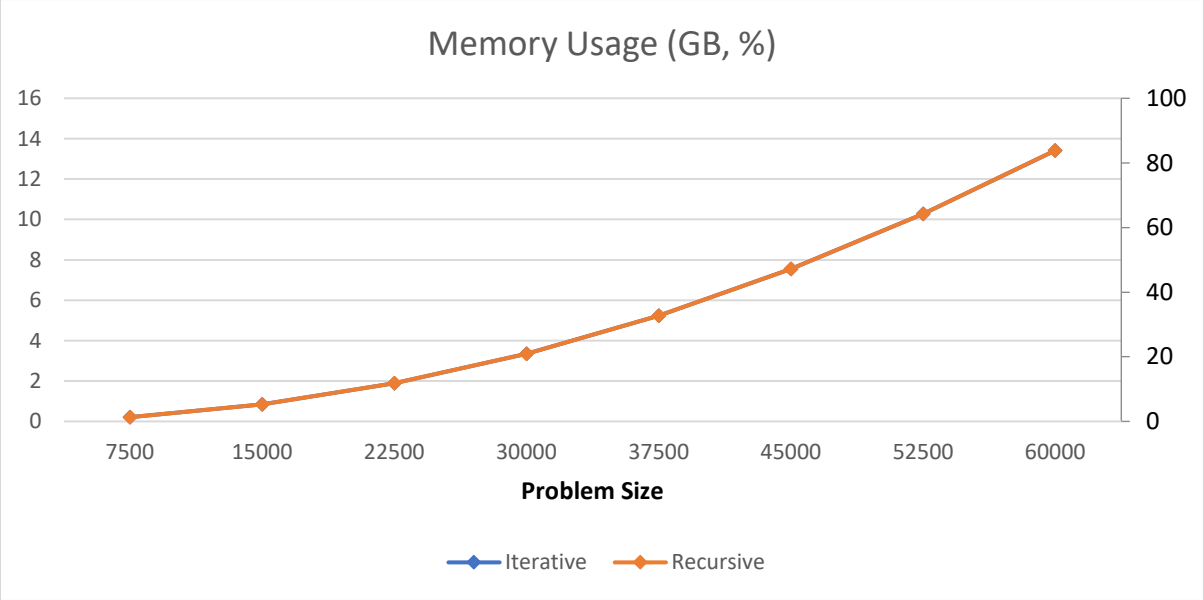

Figure 21 depicts the average memory usage in Gigabytes and % Total memory of the Dijkstra's shortest path problem for both implementations

Our results show a different story to all previous problems. While the memory usage is similar in both implementations the execution time is also quite similar. While this may seem unreasonable at first after a deeper look in our implementation we can see why this is. Dijkstra's optimal description requires that for every iteration the node we explore must be the one closest to the starting node. This means that for both implementations the same work is done, however we can see the iterative approach has a slight overhead, which is perhaps introduced because of the extra data structures required (pre-processing) to run a problem on graphs iteratively.

Figure 22 and **Figure 23** show how the density of a graph affects execution time for the iterative and recursive approach respectively. We see a similar response to the change of the density in both implementation approaches. In more detail, we see the execution time gradually increase as the density of the tree increases. We did not observe this when measuring memory, where we saw no meaningful change in memory usage when the graph density changes.

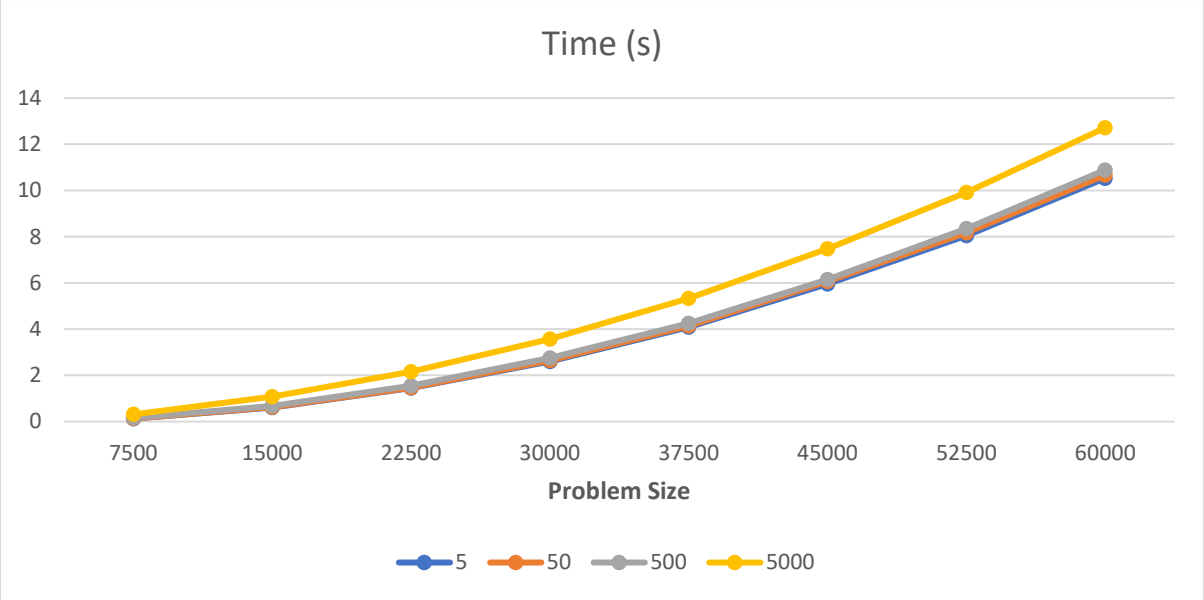

Figure 22 depicts the average execution time (in Seconds) of Dijkstra's problem for the Iterative approach with measurements of multiple graph Densities

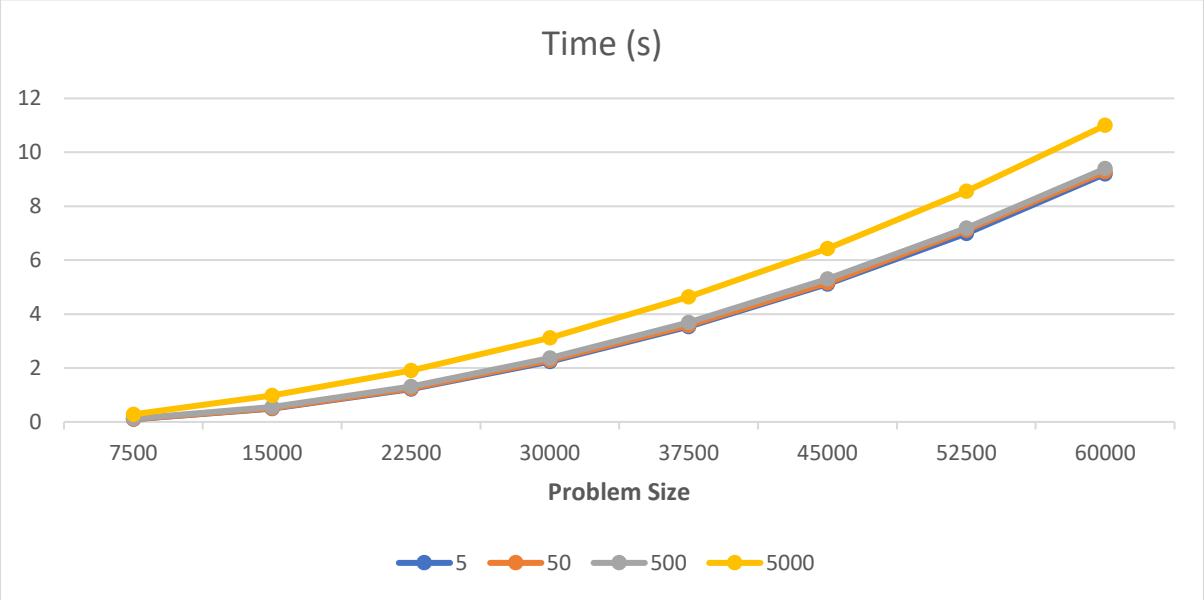

Figure 23 depicts the average execution time (in Seconds) of Dijkstra's problem for the Recursive approach with measurements of multiple graph Densities

Chapter 8

Independent Sets

8.1 Problem Description and Dynamic Programming Solution

The Independent Sets problem concerns a set of nodes *S* and the creation of a new set *S'* where for every node in *S'* one rule applies: no adjacent nodes are included in the set. In more detail, when a node is included in *S'*, all of its adjacent nodes are excluded. However, the adjacent nodes of all these excluded nodes can now be included in the new set (*S'*). The goal is to create the largest independent set *S'* possible. To do this we must include as many nodes as possible, however due to the aforementioned rule we must also exclude the least nodes possible. The exclusion of some nodes may have an overall positive effect, while the opposite is also possible.

To solve this problem, we assume a graph of nodes *G* as the set of nodes *S*. We proceed into rooting the graph *G* into a tree *T* at a random node *R* (root). To solve this problem, we assume that no cycles exist in the transformed tree.

Figure 24 and **Figure 25** show examples of the biggest independent sets in two different graphs. Nodes inside the solution are marked red. **Figure 25** can also be considered as a tree rooted at node 1. We can see that certain nodes can have a bigger negative impact than other when included in the solution. For example node 4 of **Figure 25** has 3 children leaves, which means by including it in the solution we would automatically lose 3 nodes. From this, it is now clear that the nodes must be computed in DFS order.

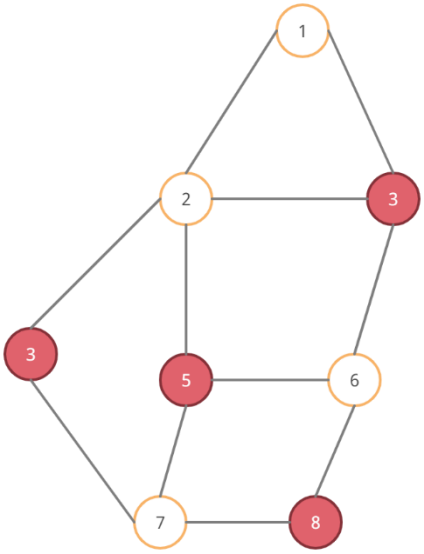

Figure 24 represents an example graph with a largest independent set coloured in red

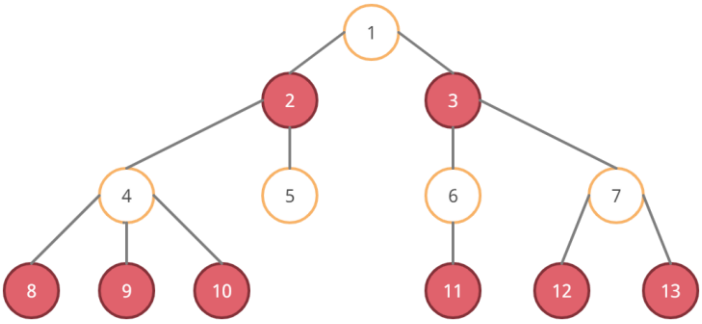

Figure 25 represents an example graph with a largest independent set coloured in red

Definitions

- Graph of *N* nodes and density *D*
- Rooted graph (tree) *T*
- A root node *R*

Sub-Problem

P(i) contains the optimal solution for node i.

$$\boldsymbol{val\_inc[i] = 1 + \sum [\![P(v) for\ every\ child\ v]\!]}$$

$$\boldsymbol{val\_exc[i] = 1 + \sum [\![P(v) for\ every\ grandchild\ v]\!]}$$

$$\boldsymbol{P(i) = \max (val\_inc[i], val\_exc[i])}$$

Bottom-Up Approach (Iterative)

```
//nodes are inserted into the stack in DFS order for optimization
int iterate(TreeNode* root, stack<TreeNode*> nodes){
	TreeNode* node;
	while(nodes.size() > 0):
		node = nodes.top();

		//if node has no children (is a leaf) set leaf value
		//and if so, remove it from the stack
		if(node->children.size() == 0):
			node->value = 1;
			nodes.pop();
			continue;

		//calculate value including the current node
		int val_including = 1;
		for(TreeNode* child: node->children):
			for(TreeNode* grandchild: child->children):
				val_including += grandchild->value;

		//calculate value excluding the current node
		int val_excluding = 0;
		for(TreeNode* child: node->children):
			val_excluding += child->value;

		//set the node value to the maximum of the 2 values
		node->value = max(val_including, val_excluding);
		nodes.pop();
	return root->value;
}
```

Top-Down Approach (Recursive)

```
int recurse(TreeNode* node){
	//node doesnt exist
	if(node == NULL)
		return 0;

	//if leaf value has already been calculated
	if(node->value)
		return node->value;

	//if its a leaf set value
	if(node->children.size() == 0)
		return (node->value = 1);

	//value when node is excluded
	int value_excluding = 0;
	for(TreeNode* child: node->children):
		value_excluding += recurse(child);

	//value when node is included
	int value_including = 1;
	for(TreeNode* child: node->children):
		if(child==NULL) continue;
		for(TreeNode* grandchild: child->children):
			value_including += recurse(grandchild);

	//return maximum of the 2
	return (node->value = max(value_including, value_excluding));
}
```

8.2 Experimental Comparison

Scenarios and Preparations

Our solution does not work on graphs containing cycles. Therefore, instead of generating a graph *G* which will later be rooted, we generate a tree to represent *S*.

This tree will be generated with a specific density *D* which represents the degree of the tree. Each node can have up to *D* children nodes (we choose the number of children of each node randomly between 0 and *D*). Every node is connected to at least 1 node, since every node is connected to their parent, hence, the tree is connected. The root is exempt from this rule because it has no parent node.

Results and Discussion

Figure 26 and **Figure 27** show the effect of input problem size in execution time (seconds) and total memory usage in Gigabytes and % memory usage respectively. To take these measurements we used a perfect tree with a degree of 2.

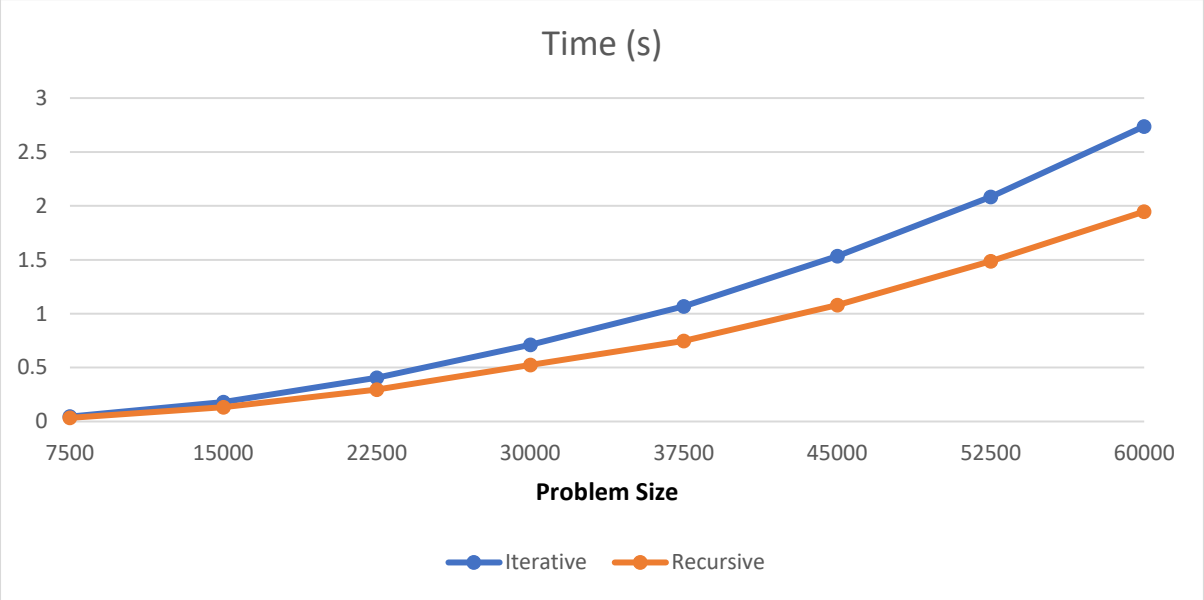

Figure 26 depicts the average execution time (in Seconds) of the Independent set problem for both implementations

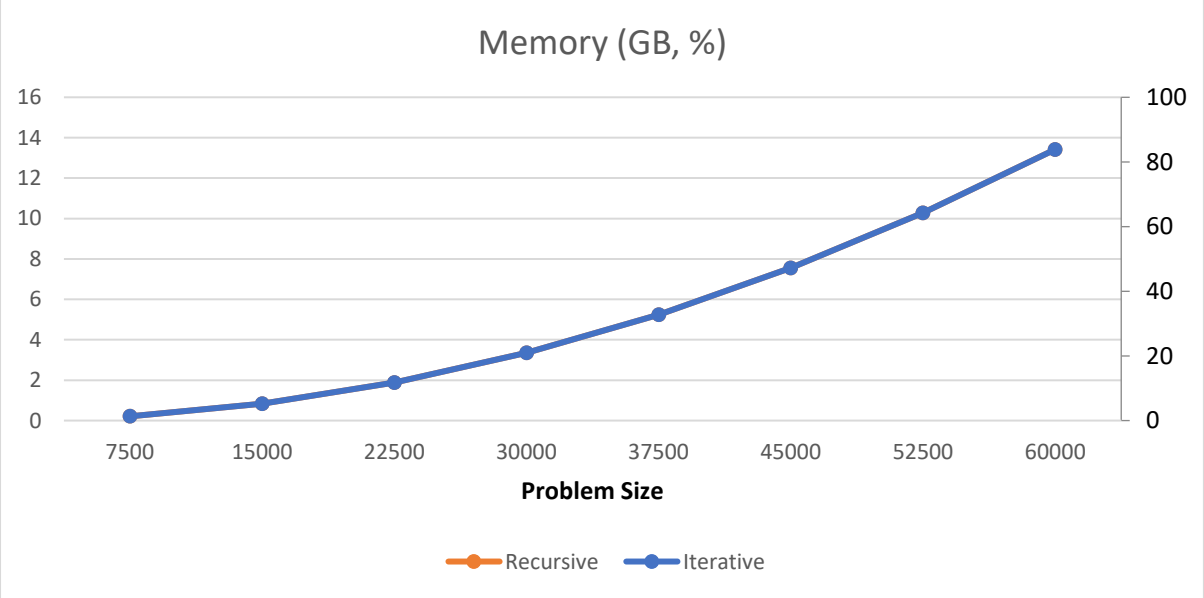

Figure 27 depicts the average memory usage in Gigabytes and % Total memory of the Independent sets problem for both implementations

This is where things get interesting. This problem is a problem solved on trees. It may be presented as a Graph (or a Set), but to solve it we have to root this graph to traverse it, and this is why cycles should not be included in the generated Tree.

The results show that contrary to all previous results, the execution of the recursive approach seems to be much faster while consuming the same amounts of memory as the iterative approach. The results at first seem unreasonable, however after some thorough investigation on the implementations things started to make sense.

The flaw of the iterative approach for this problem quickly became apparent. This problem requires the traversal of the tree in a Depth First manner (DPS). The leaves of the tree have to be computed, following them are their parent nodes, and this continuous up to the root. In other words, before computing a node we have to compute its children (if they exist).

Using recursion this can easily be done, however this is much harder and computationally intensive otherwise. This can easily be seen by comparing the two implementations. To come around this issue, we have to use new data structures, and creating these data structures is what results in the overhead of the iterative approach. Mainly, we have to put all the nodes of the tree in a list to be able to traverse them, on top of that, at every iteration we have to check which nodes can be computed. A node can be computed only if it has no children nodes or all of its children nodes have been already computed.

This results in what we observe in the plots, memory usage is similar in both approaches, since the recursive approach uses a tree to store its nodes and a 2D array to store the sub-problem data. The iterative approach however uses a queue to store the tree and the same 2D array for the problem data.

On the other hand, as previously mentioned, even if both approaches follow an exponential growth, the execution time of the iterative approach has a steeper growth, making the recursive approach much more efficient. The Top-Down approach is much faster in both execution and development / implementation time while being much easier to understand.

The following plots in **Figure 28** and **Figure 29** show how a change in tree density (*D)* affects the execution time of the Iterative and Recursive approach respectively. For the following measurements we used a complete tree. Every node of these trees can have up to *D* children nodes.

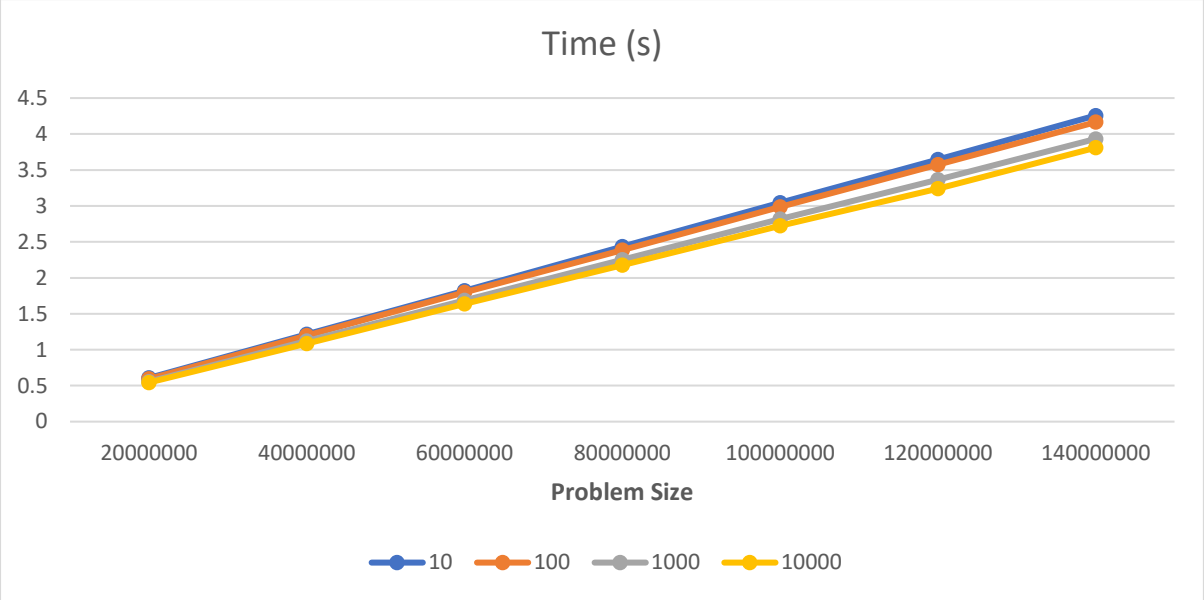

Figure 28 depicts the average execution time (in Seconds) of the Independent set problem for the Iterative approach with measurements of multiple tree Degrees

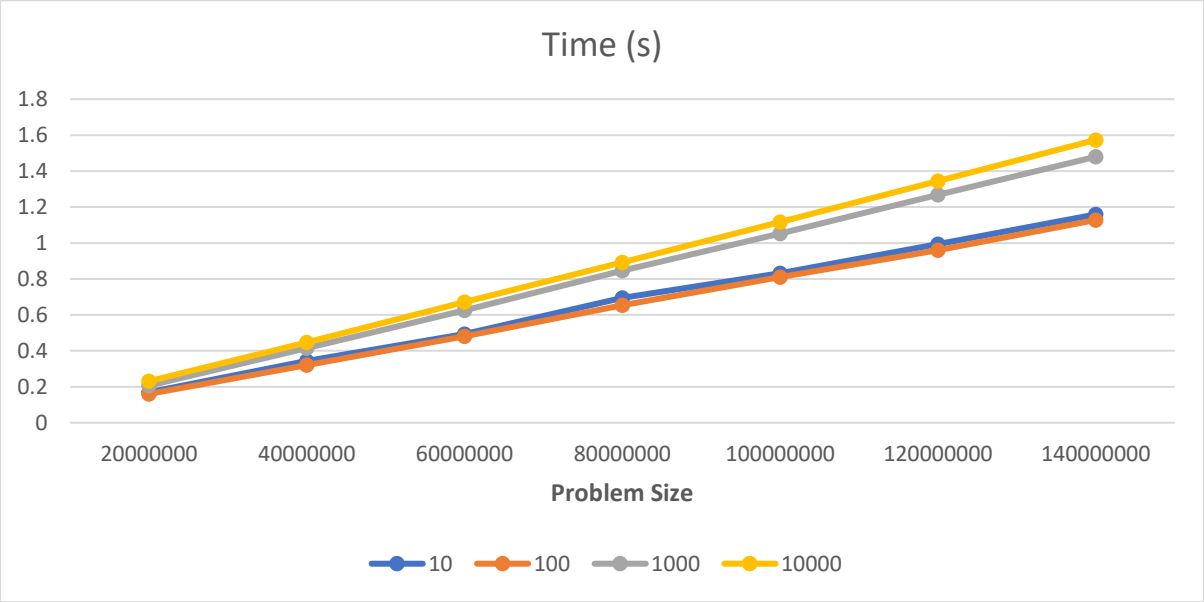

Figure 29 depicts the average execution time (in Seconds) of the Independent set problem for the Recursive approach with measurements of multiple tree Degrees

What our data shows is that as the tree density increases we see an increase in the performance of the iterative approach by speeding it up, the opposite is true for the recursive approach, where an increase in tree density slows it down. We did not observe a change in the memory usage of either implementation when measuring for different tree densities.

Chapter 9

K-Trees

9.1 Problem Description and Dynamic Programming Solution

This problem involves finding the number of subtrees of size *K*, from a tree rooted at *R*. The size of the tree is determined by the number of all its nodes including its root. Therefore, a sub-tree of size *K* is a tree with exactly *K* nodes. This problem is a very useful problem used in even distributions on trees.

Figure 30 shows an example tree of 7 nodes (numbered from 1 to 7) rooted at node 1. Coloured are its 2 k-trees of size 3 (green and cyan).

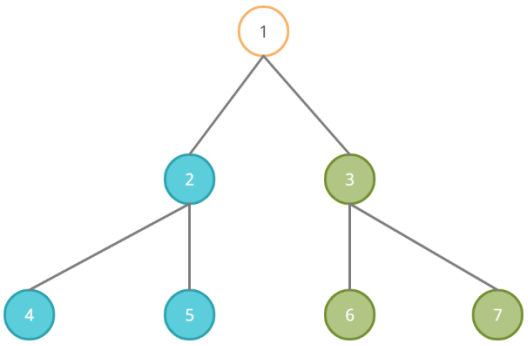

Figure 30 represents an example Graph G with 2 sub-trees of size 3 coloured green and cyan.

Definitions

- Tree of *N* nodes of degree *D*
- A root node *R*
- Cache array *C* of size *N*

Sub-Problem

P(i) contains the optimal solution for node *i*.
C[i] contains the size of the sub-tree rooted at *i*.

$$\boldsymbol{C[i]} = \mathbf{1} + \sum \llbracket \boldsymbol{C(v) for\ every\ child\ v} \rrbracket$$

$$\boldsymbol{P(i)} = (\boldsymbol{C[i]} == \boldsymbol{k}) + \sum \llbracket \boldsymbol{P(v) for\ every\ child\ v} \rrbracket$$

Bottom-Up Approach (Iterative)

```
//nodes are inserted into the stack in DFS order for optimization
int iterate(TreeNode* root, int* array, stack<TreeNode*> nodes, int k){
	TreeNode* node;
	while(nodes.size() > 0){
		node = nodes.top();

		//if node has no children (is a leaf) set leaf value
		//and if so, remove it from the queue
		if(node->children.size() == 0):
			node->value = (1==k);
			array[node->index] = 1;
			nodes.pop();
			continue;

		for(TreeNode* child: node->children):
			array[node->index] += array[child->index];
			node->value += child->value;

		array[node->index] += 1;
		node->value += (k == array[node->index]);

		nodes.pop();

	return root->value;
}
```

Top-Down Approach (Recursive)

```
int recurse(TreeNode* node, int[] array, int k){
	array[node->index] = 1;
	if(node->children.size() == 0 ):
		return 1 == k;

	int sum = 0;
	//DFS
	for(TreeNode* v: node->children):
		sum += recurse(v, array, k);
		array[node->index] += array[v->index];

	return sum + (array[node->index] == k);
}
```

9.2 Experimental Comparison

Scenarios and Preparation

To change the structure of the tree we introduce the degree variable *D*. This variable determines the maximum number of children any node can have (Degree). Firstly, we create *N* nodes and we iterate through all of them, assigning up to *D* children until all nodes receive a parent node.

Each node can have 0 to *D* children except the root which must have at least 1 (the root has no parent). We want to investigate whether the degree of the tree affects the performance of any of the implementations.

Results and Discussion

Figure 31 and **Figure 32** depict the effect of input problem size in execution time (seconds) and % total memory usage respectively. The tree used to take these measurements was a perfect tree with a degree of 2. Thus, every node had exactly 2 children (except leaves).

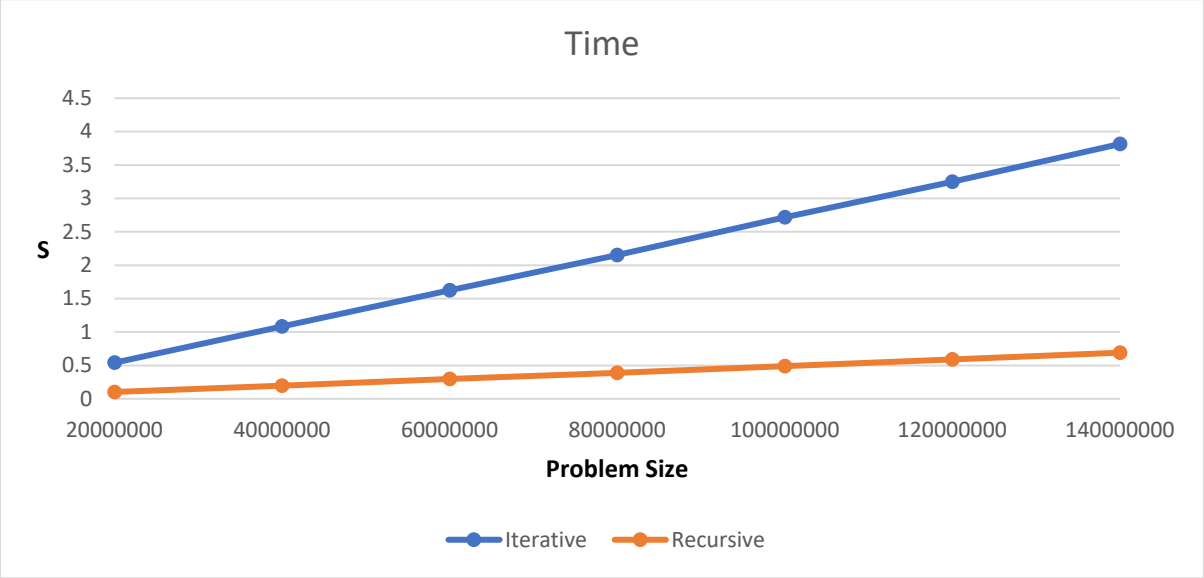

Figure 31 depicts the average execution time (in Seconds) of the K-Trees problem for both implementations

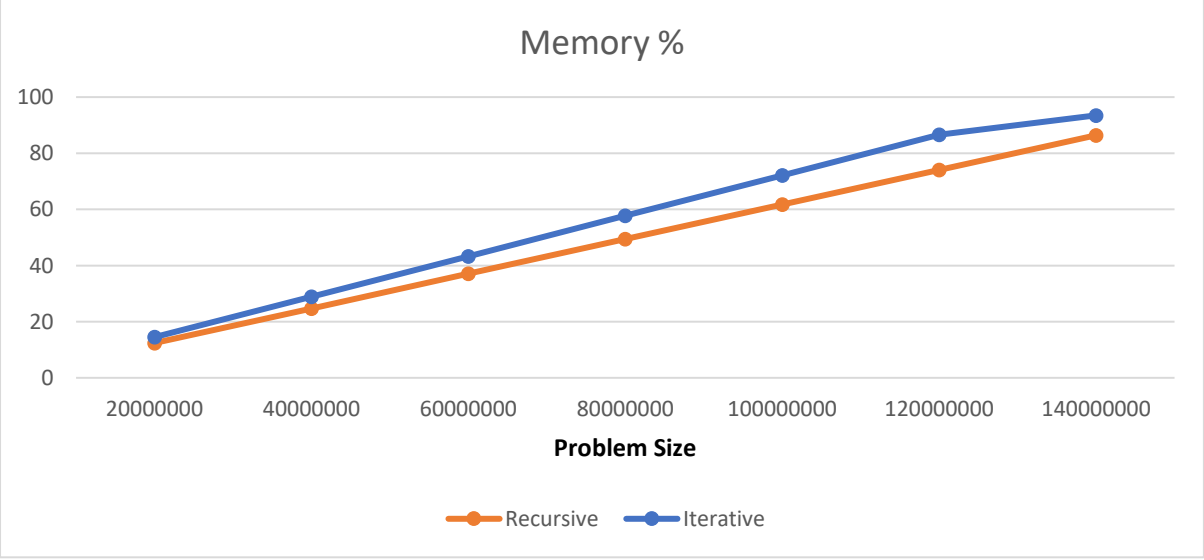

Figure 32 depicts the average memory usage in % Total memory of the K-Trees problem for both implementations

What we observe is that the memory usage growth is linear to the problem size. Likewise, the execution time is linear to the problem size. However, what we see is that the iterative approach consumes just a little more memory. What we also see is that even though both approaches run in linear time, the iterative approach is also slower.

The last point of the memory graph shows a drop of the memory usage in comparison to the general trend of the iterative approach. Our explanation is that this instance of the problem pushed the system to its outmost limits. The RAM of the computer was full, and the swap memory was used. This explains the drop on the memory usage, since the program didn't measure the swap memory used, this also explains the slower execution.

Figure 33 and **Figure 34** show the effect of the tree Degree (*D*) in execution time for the Iterative and Recursive implementation approaches respectively.

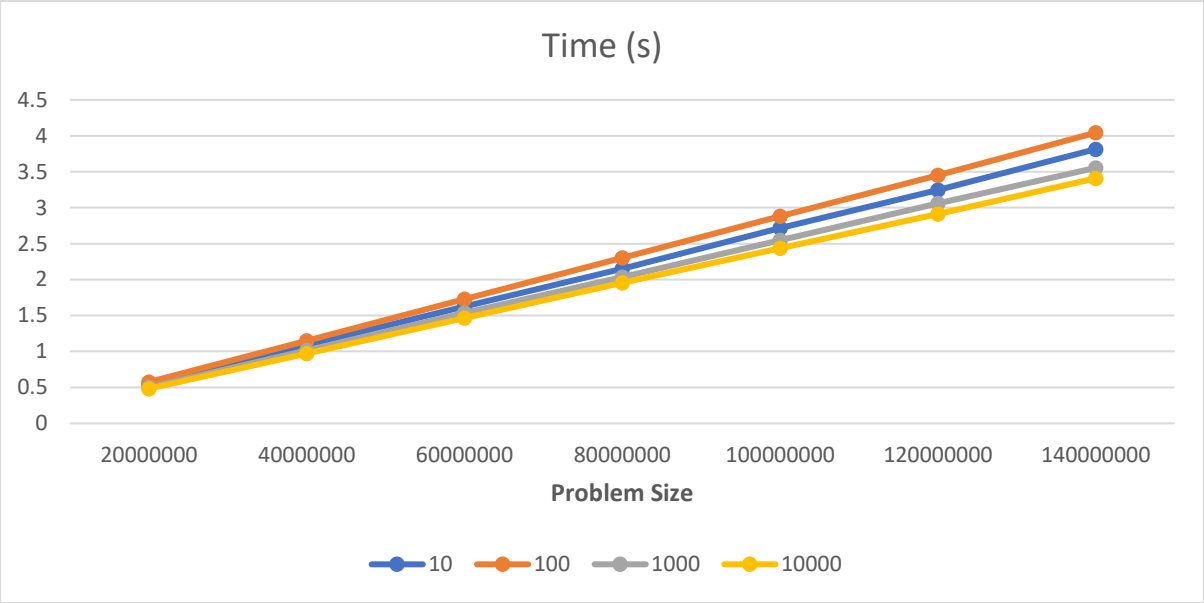

Figure 33 depicts the average execution time (in Seconds) of the K-Trees problem for the Iterative approach with measurements of multiple tree Degrees

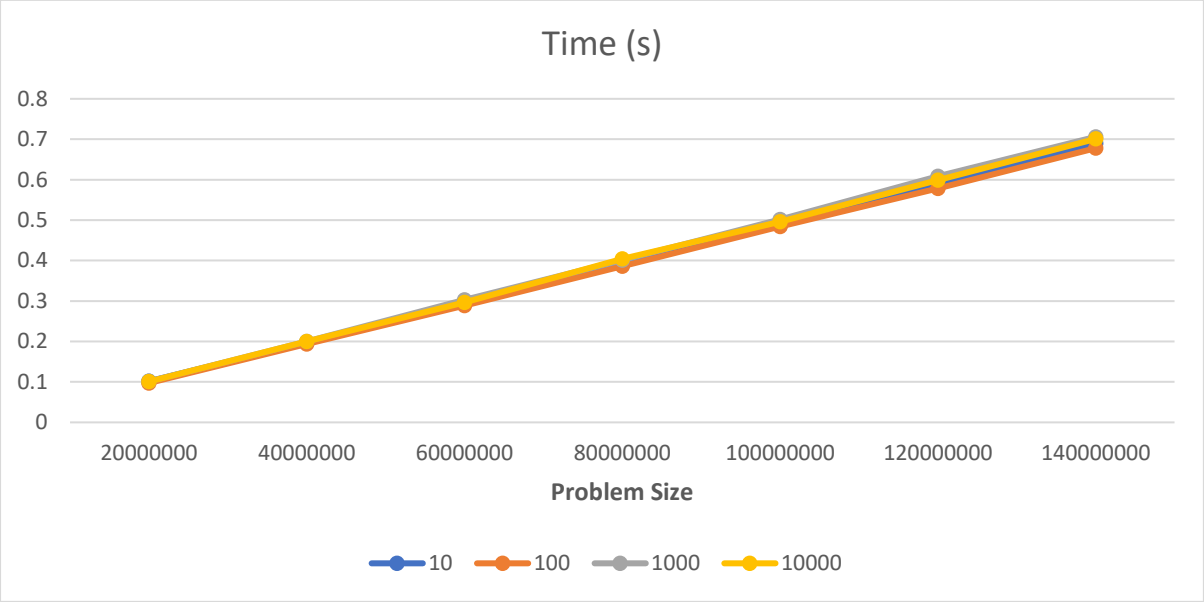

Figure 34 depicts the average execution time (in Seconds) of the K-Trees problem for the Recursive approach with measurements of multiple tree Degrees

These results show us that while the Recursive approach is gradually being slowed down by the increase of the tree Degree, the results of the Iterative approach show a different story. As the degree of the tree increases the execution time of the Iterative approach lowers, increasing its performance. In these plots we do not see any signs of the swap memory being used.

Chapter 10

Tree Diameter

9.3 Problem Description and Dynamic Programming Solution

This problem involves finding the diameter of a tree rooted at R. The diameter is the maximum distance between any two nodes inside the tree.

Figure 35 shows an example of this problem on a tree of 10 nodes. The red nodes are part of the diameter of the tree. The diameter shown is just one of the possible solutions. Another possible solution would occur if we excluded node 8 and included node 9.

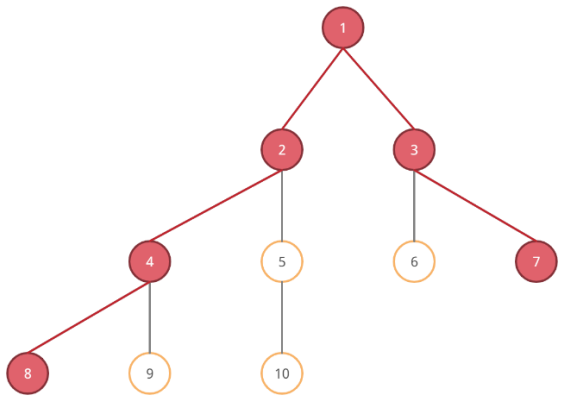

Figure 35 represents an example graph G with its diameter coloured in red.

Definitions

- Tree of *N* nodes of Degree *D*
- *A* root node *R*
- Caching arrays *max1_array* and *max2_array* of size *N*

Sub-Problem

From the example on **Figure 35** we make two observations. A node can either not be included in the solution or be included with at most 2 of its children. Only 1 node of the solution can have 2 of its children be included in the solution path. The rest of the nodes can either have 1 or none (leaves only). Therefore, we will keep track of the two longest paths that occur at every node. Then the parent of each node will compare these paths to find the global maximum.

- *max1_array[i]* contains the longest path of node *i* including itself with 1 of its children.
- *max2_array[i]* contains the longest path of node *i* including itself with 2 of its children.
- *P(i)* contains the optimal solution for node i (max of *max1_array[i]* and *max2_array[i])*.
- We use DFS (Depth First Search) to traverse the tree from its leaves first

From this we derive the following:

$\boldsymbol{for\ every\ node\ n\ in\ DFS\ (from\ root\ R)}$

- $\boldsymbol{tmp1 = max(max1\_array[v]\ for\ every\ child\ v)}$
- $\boldsymbol{tmp2 = 2nd\ \ max(max1\_array[v]\ for\ every\ child\ v)}$
- $\boldsymbol{max1\_array[n] = 1 + tmp1}$
- $\boldsymbol{max2\_array[n] = 1 + tmp1 + tmp2}$
- $\boldsymbol{P(i) = max\ (max1\_array[n], max2\_array[n])}$

Note: The root might not be included in the solution, so we have to keep track of the diameter of the tree at a global scope. Another solution would be to keep track of the node with the largest *max2_array* value since it will contain the value of the diameter (only 1 node contains two paths).

Bottom-Up Approach (Iterative)

```
//nodes are inserted into the stack in DFS order for optimization
int iterate(long[] array, TreeNode* root, stack<TreeNode*> nodes){
	int maxVal=0;
	TreeNode* node;
	while(nodes.size() > 0):
		node = nodes.top();

		//if node has no children (is a leaf) set leaf value
		//and if so, remove it from the queue
		if(node->children.size() == 0):
			node->value = 1;
			array[node->index]= 1;
			nodes.pop();
			continue;

		//calculate firstmax and secondmax of node
		int firstmax=0, secondmax=0;
		for(TreeNode* child: node->children):
			if(child->value >= firstmax):
				secondmax = firstmax;
				firstmax = child->value;
			:else if(child->value > secondmax)
				secondmax = child->value;

		//set the node value to the maximum of the 2 values
		node->value = firstmax + 1;
		array[node->index]= 1 + firstmax + secondmax;
		maxVal = max(max(node->value, array[node->index]), maxVal);
		nodes.pop();

	return maxVal;
}
```

Top-Down Approach (Recursive)

```
Int max1[N] ={0}, max2[N] ={0};
int recurse(TreeNode* root){
	vector<int> fValues;

	//DFS traversal
	for every child c of node:
		recurse(c);
		//push chldrens' values of inc array into list
		fValues.push_back(max1[c->index]);

	//find 2 max fvalues
	int tmp1=-1, tmp2=-1;
	for every index i of fValues:
		if(fValues[i] >= max1):
			tmp2 = tmp1;
			tmp1 = fValues[i];
		else if(fValues[i] > max2):
			tmp2 = fValues[i];

	//this is necessary for leaves (no children, therefore, no tmp1 value)
	inc_array[root->index] =1;
	if(tmp1 != -1):
		max1[root->index] += tmp1;

	if(tmp2 != -1):
		max2[root->index] = 1 + tmp1 + tmp2;

	return diameter = max(diameter, inc_array[root->index], exc_array[root-
>index]);
}
```

9.4 Experimental Comparison

Scenarios and Preparation

Similarly to previous tree problems, we use the degree variable *D* to change the structure of the tree. The tree is fully connected. Each node has a single parent (except the root) and all nodes have up to *D* children (the root has at least 1). We change the degree of the tree to investigate whether the structure of the tree affects the performance of any of the two implementations.

Results and Discussion

Figure 36 and **Figure 37** depict the effect of input problem size in execution time (seconds) and % total memory usage respectively. The tree used to take these measurements was a perfect tree with a degree of 2. Thus, every node had exactly 2 children (except leaves).

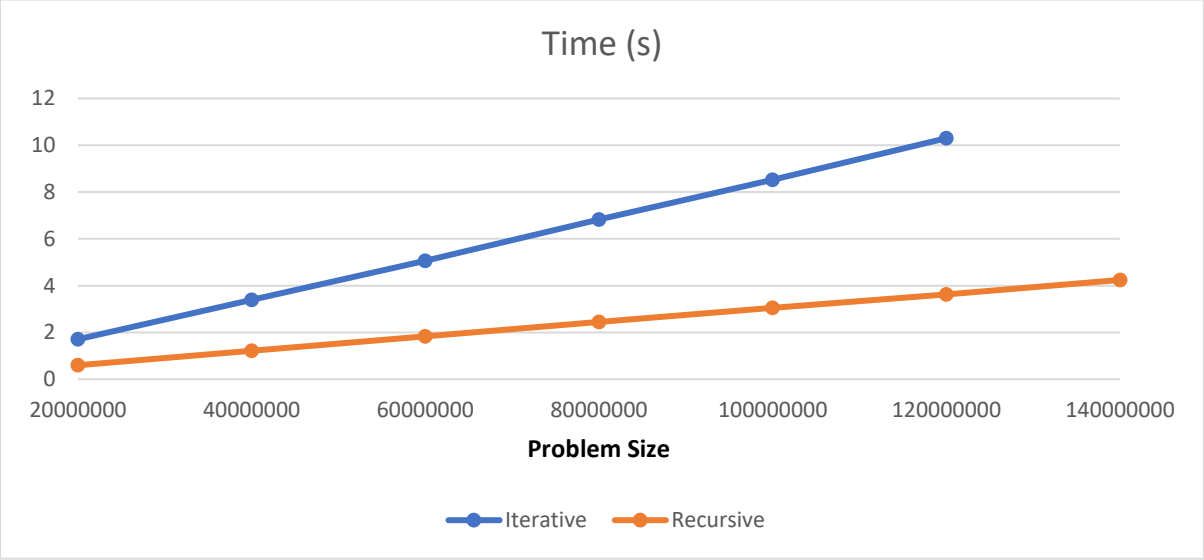

Figure 36 depicts the average execution time (in Seconds) of the Tree Diameter problem for both implementations

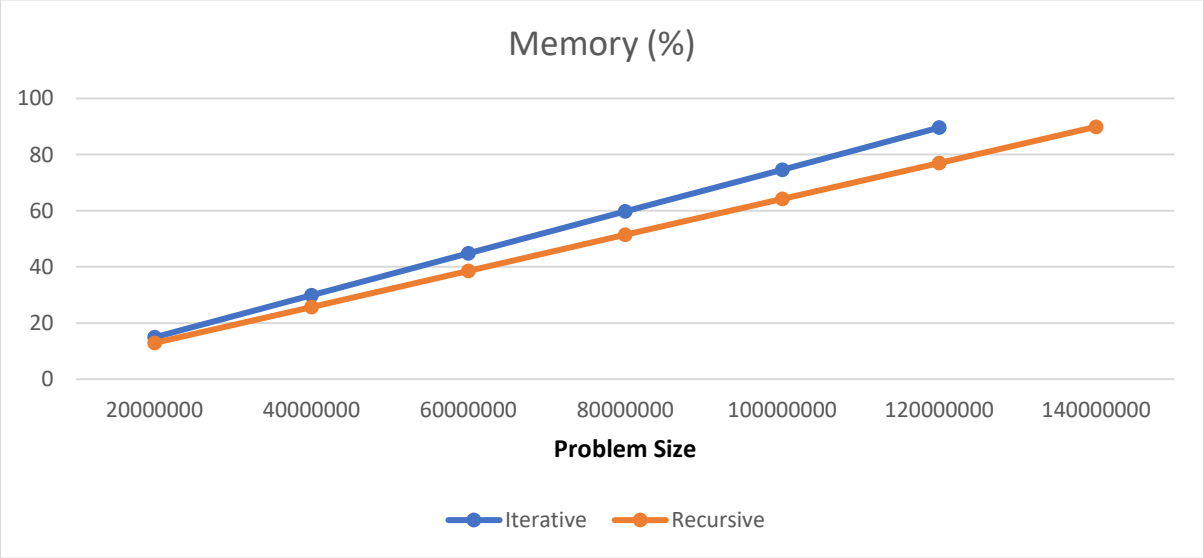

Figure 37 depicts the average memory usage in % Total memory of the Tree Diameter problem for both implementations

The first observation is that the efficiency of the recursive approach is greater in both aspects. We see a linear increase in both the memory usage and the execution time for both approaches. However, the rate of increase of the iterative approach is greater. In more detail, the execution time is much greater but the memory usage not so much, however it was enough to make this

approach unable to run the problem for the last input size. The recursive approach is a clear winner in both aspects, especially in execution time.

Now its time to test the effect of the tree Density to the performance of the two implementations.

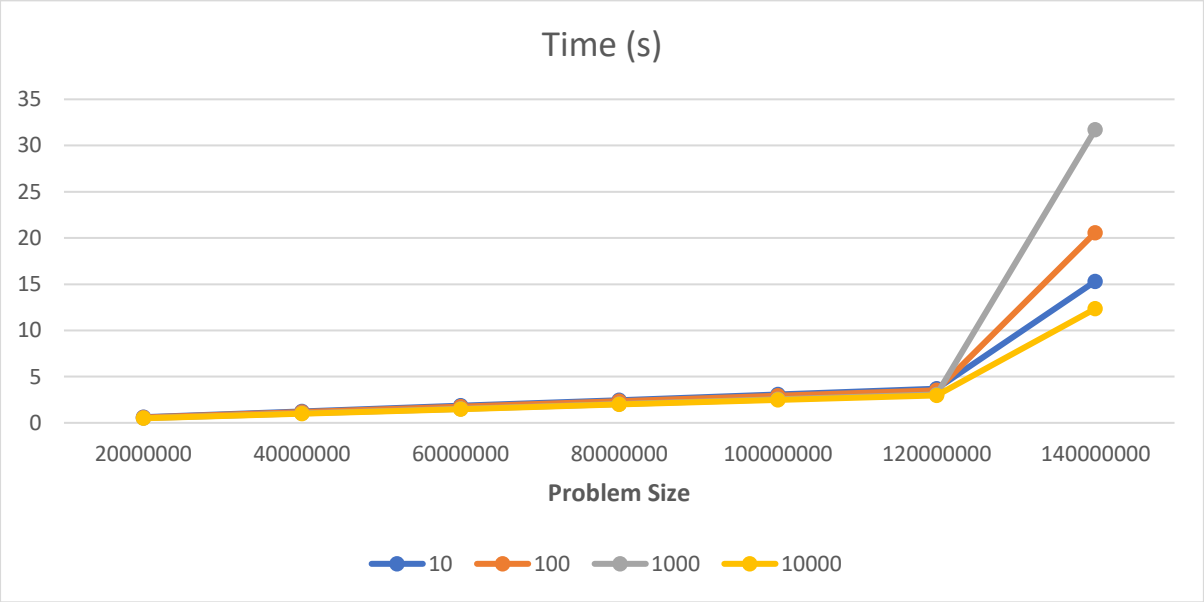

Figure 38 depicts the average execution time (in Seconds) of the Tree Diameter problem for the Iterative approach with measurements of multiple tree Degrees

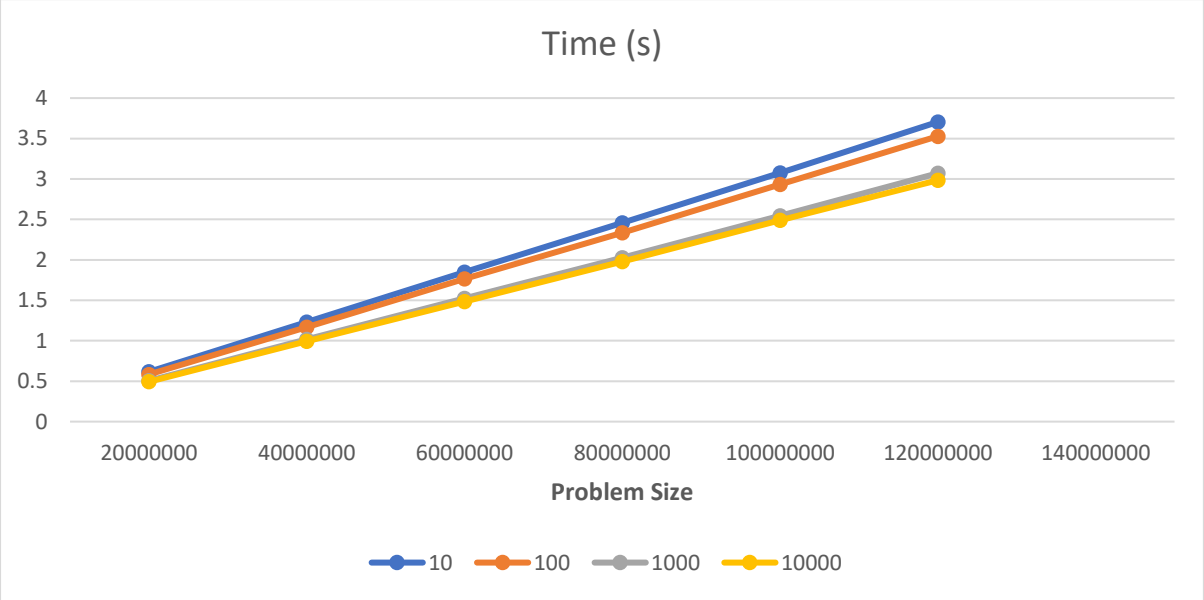

Figure 39 depicts the average execution time (in Seconds) of the Tree Diameter problem for the Iterative approach with measurements of multiple tree Degrees (plot cleaned)

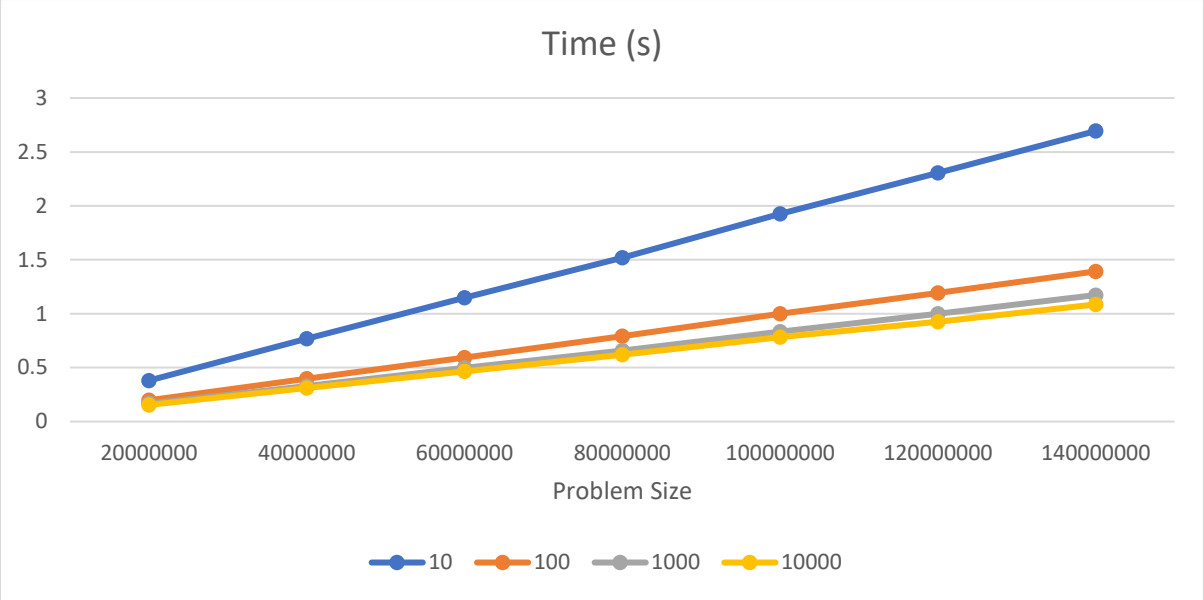

Figure 40 depicts the average execution time (in Seconds) of the Tree Diameter problem for the Recursive approach with measurements of multiple tree Degrees

In **Figure 38** we observe the change that occurs in the execution time of the iterative approach when the degree of the tree changes. In this plot we observe un unreasonably huge slowdown as the memory is being filled, this is where the swap memory is being used. Since the swap memory is much slower the execution slows down, hence the performance suffers. In **Figure 39** we remove the last measurements to make the changes of the previous measurements clearer. **Figure 40** depicts the effect of the tree degree in the execution time of the recursive implementation. Comparing these results we observe that both approaches seem to benefit in an increase of the tree degree. As the tree degree increases so does the performance of both implementations. We did not observe a measurable change in the memory usage of either implementation as the degree of the tree changes.

Chapter 11

Conclusion

10.1 Summary

After close inspection of the obtained results we came to some surprising conclusions. The type of the input of the problem seems to play an important role in its performance. We categorize the problems based on the Input Complexity.

Problem Categorization

	Input
Array (1D or 2D)	MCSS, LISS 1D, LISS 2D, Chain Matrix Multiplication, Knapsack
Graph	Dijkstra
Tree	Independent Sets, K-Trees, Tree Diameter

Through all the experiments, we conclude the following:

Input	Problems	Conclussion
2D Array	• MCSS • LISS 1D • LISS 2D • Chain Matrix Multiplication • Knapsack	These problems are heavily favoured by the iterative approach in regards to the Execution time. The memory usage is very similar for either approach, however we have reason to believe that some problems may even favour the iterative approach in memory usage as well.
Graphs	• Dijkstra's Shortest Path	This problem is a problem presented on a Graph. The limitation of the iterative approach becomes clear here. A problem that is declared in a "pointer style" data structure is hard to traverse in an iterative manner. Even when we use an array to represent the graph it is even harder to traverse the graph in DFS order. Dijkstra's algorithm however requires the traversal of the graph in a specific manner, therefore the results of the two implementations are very similar. The iterative approach had a slight overhead in execution time caused by the pre-processing of the data.
Trees	• Independent Sets • K-Trees • Tree Diameter	Here is where things got interesting. When solving problems on graphs the weakness of the iterative approach showed itself. However, problems on trees showed something more, a strength of the recursive approach. Traversing trees is very easy in a recursive manner, making it both faster in development (implementation) and in execution. Yielding better results both in memory usage and execution time. The difference was not as much in the memory used, however the execution was a lot faster, making the recursive approach the clear winner for problems on trees.

A general conclusion is that Tree and Graph traversal is both more intuitive, faster and more memory efficient in a recursive manner instead of an iterative manner.

10.2 Problems Encountered

There were several problems encountered during this study. Most of the problems had to do with data collection. As mentioned at the beginning measuring the memory usage of a program can be tedious, especially when dealing with problems that consume a lot of memory which was the case for the project. The tools we used were unable to function because of the high memory demands, so we had to resort in different methods.

The biggest problem faced however was the time it took to collect the data. This was the case with problems like the MCSS and the Chain Matrix Multiplication. For big problem sizes, that would use up most of the memory, the execution of the program would take hours. In the case of the Chain Matrix Multiplication program the execution would take weeks and even months. Therefore, we had to change our input sizes.

10.3 Future Work

Finding out about the behaviour of the recursive approach on different data structures was not something we suspected at first. However just when we began developing the programs we realised how useful recursion is on some data structures like trees. Discovering more of these behaviours would be interesting.

An interesting extension of this work is to further investigate how other parameters would affect the performance of each approach. A comparison between the same implementation in two different programming languages would be interesting. The effect of the Operating System may also be important, both in resource usage and execution time.

Bibliography

[1] Thomas H. Cormen, Charles E. Leiserson and Ronald L. Rivest, “Introduction to Algorithms”.

[2] Sanjoy Dasgupta, Christos Papadimitriou and Umesh Vazirani, “Algorithms”.

[3] Hans Kellerrer, Ulrich Pferschy, David Pisinger, “Knapsack Problems”, 2004.

[4] Yanhong A. Liu and Scott D. Stoller. Stony Brook University. “From recursion to iteration: what are the optimizations?”
https://www3.cs.stonybrook.edu/~stoller/papers/PEPM2000.pdf

[5] Chryssis Georgiou, Lectures notes of CS236: Algorithms and Complexity, Dept. of Computer Science, University of Cyprus, 2020 https://www.cs.ucy.ac.cy/~chryssis/EPL236/

[6] Ubuntu OS, Linux distribution: https://ubuntu.com/.

[7] Valgrind memory checking tool: https://www.valgrind.org/docs/manual/quick-start.html.

[8] Massif profiler for the Valgrind tool: https://www.valgrind.org/docs/manual/ms-manual.html.

[9] The “top” program manual page: https://man7.org/linux/man-pages/man1/top.1.html.”

[10] The “time” C++ library, http://www.cplusplus.com/reference/ctime/time/