|  |
| --- |
| 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

January2017

**Acknowledgements**

I would like to thank my professor Dr. Chryssis Georgiou. His ideas, guidance and supervision helped me get throughout the whole process.

**Summary**

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

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 memorization 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**

1. Introduction 8

1.1 What is Dynamic Programming? 8

1.2 Approaches to Dynamic Programming 9

1.3 Objective of the study 10

1.4 Methodology 10

1.5 Document Structure 10

2. Background 11

2.1 Preliminary Research 11

2.2 Experimenting with other Data Structures 12

2.3 Measuring Execution Time 12

2.4 Memory Measurement 13

2.5 System Used 13

2.6 CPU Utilization Measurements 13

3. Most Common Sub Sequence 14

3.1 Problem Description and Dynamic Programming Solution 14

3.2 Experimental Comparison 17

3.3 Results and Discussion 18

3.4 Conclusions 18

Chapter 4 20

4. Longest Increasing Sub Sequence 20

4.1 2D Solution 20

4.1.1 Problem Description and Dynamic Programming Solution 20

4.1.2 Experimental Comparison 21

4.1.3 Results and Discussion 22

4.1.4 Conclusions 22

4.2 1D Solution 23

4.2.1 Problem Description and Dynamic Programming Solution 23

4.2.2 Experimental Comparison 23

4.2.3 Results and Discussion 24

Run 1D for bigger problems 24

4.3 Conclusions 25

4.4 Comparison between 1D and 2D solutions 25

5. Chain Matrix Multiplication 26

5.1 Problem Description and Dynamic Programming Solution 26

5.2 Experimental Comparison 28

5.3 Results and Discussion 28

5.4 Conclusions 28

6. 0-1 Knapsack 29

6.1 Problem Description and Dynamic Programming Solution 29

6.2 Experimental Comparison 33

6.3 Results and Discussion 33

6.4 Conclusions 34

7. Dijkstra 35

7.1 Problem Description and Dynamic Programming Solution 35

7.2 Experimental Comparison 37

7.3 Results and Discussion 38

7.4 Conclusions 38

8. Independent Sets 40

8.1 Problem Description and Dynamic Programming Solution 40

8.2 Experimental Comparison 42

How we obtained tree 42

8.3 Results and Discussion 42

8.4 Conclusions 43

9. K-Trees 45

9.1 Problem Description and Dynamic Programming Solution 45

9.2 Experimental Comparison 47

9.3 Results and Discussion 47

9.4 Conclusions 48

10. Tree Diameter 49

10.1 Problem Description and Dynamic Programming Solution 49

10.2 Experimental Comparison 51

10.3 Results and Discussion 52

10.4 Conclusions 52

11. Conclusion 53

11.1 Summary 53

11.2 Problems Faced 55

11.3 Future Work 55

An interesting extension of this work would be to further investigate what other parameters would affect the performance of the recursive approach. 55

How about programming language? OS? Would these make a difference? 55

Chapter 1

1. Introduction

|  |  |
| --- | --- |
| 1.1 What is Dynamic Programming? | 7 |
| 1.1 Approaches to Dynamic Programming | 8 |
| 1.3 What is my goal? | 9 |

* 1. What is Dynamic Programming?

Dynamic Programming [citation] 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 implementation make their entrance.

* 1. 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 [citation].

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 **memorization**). 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. 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. Methodology

Here you provide a roadmap of your work. E.g., First, a deeper understanding of dynamic programming and its two approached was obtained. Then….

Say that more details are given in Chapter 2.

* 1. Document Structure

In chapter 3…

In chapter 4..

In chapter 5..

Chapter 2

1. Background

|  |  |
| --- | --- |
| 2.1 Preliminary Research | 10 |
| 2.2 Experimenting with other Data Structures | 11 |
| 2.3 Measuring Execution Time | 11 |
| 2.4 Memory Measurement | 12 |
| 2.5 System Used | 12 |
| 2.6 CPU Utilization Measurements | 13 |

* 1. Preliminary Research

An introduction to Dynamic Programming can be had from the courses of University of Cyprus that study algorithms [citation]. To understand the behaviour of each implementation we will follow execution examples as well as pseudo code. Examples are helpful since they may provide a first idea of the limitations as well as the expected performance of an implementation.

Through some examples we will understand how the Top-Down and the Bottom-Up implementations work and how the array is filled. This will help also help us understand the reasoning behind the name of each implementation.

* 1. 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 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 1/3 of the array must be unused. Therefore, the number of unique branches a recursion must make should be less than 1/3 of N (if we assume than an iterative approach uses an NxN array).

It should be noted that the Valgrind tool [citation] retrieves the memory usage of a program in snapshots. Therefore, in rare cases the memory usage may seem inconsistent, 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.

* 1. Measuring Execution Time

After exploring other data structures and concluding to the table as the best option, I began researching ways to measure memory usage and execution time. I realised that some Dynamic Programming problems require some pre-processing, others require some data conversions. I concluded that I could not use any external process to capture the execution time because by doing that i 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.

* 1. Memory Measurement

To measure the memory usage I seeked the help of external tools. I used the **Valgrind** memory checking tool [citation] available for Linux machines. I 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, I used a plugin of the Valgrind tool called **Massif** [citation]. This tool however has a downfall which was not discovered at first. To closely inspect the stack usage of a program each allocation that occurs on the stack is measured, therefore each allocation requires even more memory, this means that recursive programs that usually require plenty of stack were not measured since the demand of the stack was bigger than the available system memory.

To get through this problem I begun studying other memory measuring tools. I realised that the system monitoring tool was my best bet. To capture the memory usage of a program would start the program in a new process to capture its Process ID (PID). Then I would inspect the memory usage of that process using the “**top**” tool [citation] available for Linux. I would capture the memory usage % in intervals of 0.01 seconds, and store it in a temporary file, later finding the maximum amount of memory usage from that file. If possible I would use both the Valgrind and Top tools for the same problem comparing the Results and Discussion, and I found out that both techniques yielded very similar Results and Discussion.

* 1. System Used

The Operating System used during development was Ubuntu 20.04LTS. All programs were developed in C++ (C std 11) in visual studio code. We believe that C++ is the optimal language to allow for precise memory management and avoid unnecessary performance overheads, hence it allows for a more precise comparison between the two approaches, both regarding memory usage and execution time.

* 1. 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.

**Chapter 3**

1. Most Common Sub Sequence

|  |  |
| --- | --- |
| * 1. Description |  |
| * 1. Experimental Comparison |  |
| * 1. Results and Discussion |  |
| * 1. Conclusions |  |

* 1. Problem Description and Dynamic Programming Solution

The Most Common Sub-Sequence problem [citation] 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). 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. able

****Definitions****

* *Sequence A, B*
* *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****

is the most optimal solution for the first *i-th* letters of sequence *A* and *j-th* letters of sequence *B*. Since is the most optimal solution we can assume that:

* ***if******then***
* ***else***

Therefore we derive the following recursive equation:

* *(end case)*

****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 end case. This approach calculates every possible combination of *i* and *j* (*N+1 x M+1* combinations).



|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 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 shows the contents of the array storing sub-problems of the iterative implementation for the example given.

* + 1. ****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 end case. The array is initialized with -1 for the sake of **Memoization**. Recursion stops at the end cases, or when a value that is not ‘-1’ is reached. Which means the sub-problem has already been solved by another recursion instance.



|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 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 shows the contents of the array storing sub-problems of the recursive implementation for the example given.

* 1. Experimental Comparison

Since the problem requires 2 sequences as input we can adjust the length and complexity of these sequences to test the problem in multiple ways. Therefore, we will test 3 different sequence lengths for each problem size. Firstly, when sequence A length amounts for of sequence B’s, similarly we will check for and for equal lengths (length(A) == length(B)). Since the problem is symmetrical it means that by testing sequence A and B we have the Results and Discussion for B and A.

The similarity of these sequences can also be tested, sequences similar to each other may favour one of the 2 approaches, thus we will check randomly generated sequences, as well as sequences generated with a similarity factor introduced.

Randomly generated sequences will also be tested

* 1. Results and Discussion

Figure 1 depicts the effect of problem size in execution time for both implementations

Figure 2 depicts the effect of problem size in memory used (Megabytes) and % memory usage for both implementations

* 1. Conclusions

As one can easily observe for the Most Common Sub Sequence problem the execution time for the recursive approach yields much slower Results and Discussion while consuming more memory (Figure 1 and Figure 2). In both instances the memory usage has quadratic growth in regards to the problem size, however the recursive approach consumes much more memory due to its stack usage which is not needed in the iterative approach. On top of that, the recursive approach yields much slower Results and Discussion, this may be due to the huge number of recursive branches that occur due to the recursive function.

For this problem the iterative approach is a clear winner

Chapter 4

1. Longest Increasing Sub Sequence

|  |  |
| --- | --- |
| * 1. Description | 17 |
| * 1. Experimental Comparison | 18 |
| * 1. Results and Discussion | 19 |
| * 1. Conclusions | 19 |

* 1. 2D Solution
     1. Problem Description and Dynamic Programming Solution

The problem of the Longest Increasing Sub-Sequence 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.

****Definitions****

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

****Sub-Problem****

is the most optimal solution for the *i-th* letter moving to the *j-th* letter, (letter *j* is the next number of the optimal sequence). Since *P(i,j)* is the most optimal solution we can assume that:

* ***if*** *S* ***then***
* ***otherwise***

****Bottom-Up Approach (Iterative)****



Top-Down Approach (Recursive)



* + 1. Experimental Comparison

The problem requires a single number sequence. To test the performance of the approaches with different inputs we will firstly generate random number sequences with specified ‘M’ values.

Lastly, we can generate sequences with different tendencies regarding:

* Their average value (sequences with similar values may favour one of the 2 approaches since similar values generally mean that it is harder to find an increasing sub-sequence)
* An ascending or a descending behaviour (the values may increase or decrease gradually to test the efficiency of each approach)
  + 1. Results and Discussion

Figure depicts the effect of problem size in execution time for both implementations

Figure depicts the effect of problem size in % memory usage for both implementations

* + 1. Conclusions

We can conclude that both approaches behave in a similar manner, both measurements (time and memory) seem to perform equally with either approach.

* 1. 1D Solution

* + 1. Problem Description and Dynamic Programming Solution

This problem is a continuation of the LISS problem solved using a 2D array, however optimizations allow us to solve this problem using a 1D array as well. 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 dimensions by 1).

Bottom-Up Approach (Iterative)



****Top-Down Approach (Recursive)****



* + 1. Experimental Comparison

To compare the Results and Discussion we acquired by using the 2D solution with our new solution, we will use the same input parameters. However, to test the limitations of this solution as well we will test for the maximum memory usage as well.

* + 1. Results and Discussion

Figure depicts the effect of problem size in execution time for both implementations

Figure depicts the effect of problem size in execution time for both implementations

* 1. Conclusions

Comparing our Results and Discussion to the previous solution we can make two observations. Firstly, the 1D solution is much more efficient in regard to memory usage. Secondly the dynamic programming approach used vastly affects execution time. The Results and Discussion are very interesting in the original solution. We can see that the recursive approach has a very poor performance, while the iterative approach is much faster, even faster than the memory optimized solution.

* 1. Comparison between 1D and 2D solutions

Therefore, the Results and Discussion show that the solution that favours execution time is the 2D solution using the iterative approach, while the solution that favours memory usage is the 1D solution using either of the two approaches since they both yield similar Results and Discussion.

**Chapter 5**

1. Chain Matrix Multiplication

|  |  |
| --- | --- |
| * 1. Description | 25 |
| * 1. Experimental Comparison | 27 |
| * 1. Results and Discussion | 27 |
| * 1. Conclusions | 27 |

* 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 and Discussion in a new matrix of size NxM [citation]. This process requires *NxIxM* operations, this is also set to be the cost of the multiplication. When multiple matrices have to be multiplied in a sequence, 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.

We use a ‘step’ variable, let ‘step’ be s. We start solving *P(i, i+s)* 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*).

****Definitions****

* List *L*, of *N* Matrices *(WixHi)*
* *( L(i).w, L(i).h )* are the dimensions of the *i-th* matrix in the list, where ‘*w’* is its width and ‘*h’* is its height.
* *L* contains the matrices in order, therefore we can assume, *L(i).w = L(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. Therefore, we can assume L is a list of N+1 values, and *N(i)* is equal to *L(i).*w and *N(i)* is also equal to *L(i+1).h*.
* Maximum value – matrix width/height *M*

****Sub-Problem****

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

****Bottom-Up Approach (Iterative)****



****Top-Down Approach (Recursive)****



* + 1. ****Example****

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 |

* 1. Experimental Comparison

To test the efficiency and performance of each approach we will generate a list ‘L’ of size N+1 values and a specified maximum value M

* 1. Results and Discussion

Figure depicts the effect of problem size in execution time for both implementations

Figure depicts the effect of problem size in memory usage (Megabytes) for both implementations

* 1. Conclusions

**Chapter 7**

1. 0-1 Knapsack

|  |  |
| --- | --- |
| * 1. Description | 29 |
| * 1. Experimental Comparison | 33 |
| * 1. Results and Discussion | 33 |
| * 1. Conclusions | 34 |

* 1. Problem Description and Dynamic Programming Solution

The Knapsack problem [citation] is to find the selection of items within a certain weight limit C, with the most value. 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 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****

*P(i,j)* is the most optimal value for all items up to the *i-th* item in the *j-th* sack (sack with capaciy *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.

****Example****

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 |

****Bottom-Up Approach (Iterative)****



We begin from item with index 0 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 |

****Top-Down Approach (Recursive)****



|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 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 |

* 1. Experimental Comparison

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].

* 1. Results and Discussion

Figure depicts the effect of problem size in execution time for both implementations

Figure depicts the effect of problem size in memory usage (GB) and % total memory usage for both implementations

* 1. Conclusions

Our Results and Discussion follow a similar trend to previous problems using a 2D array solution. In more detail, 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 quadratic growth in regards to the problem size. This growth is also present in the execution time of the recursive approach, while the growth of the iterative approach is linear resulting in much faster executions as the problem size increases making it even more efficient the bigger the input size is.

**Chapter 7**

1. Dijkstra

|  |  |
| --- | --- |
| * 1. Description | 10 |
| * 1. Experimental Comparison | 10 |
| * 1. Results and Discussion | 10 |
| * 1. Conclusions | 10 |

* 1. 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 and Discussion, 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 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
* Nodes *S* (‘Start node’) and node *T* (‘Finish’ / ‘End node’)
* ‘Visited’ array *V* of size *N*
* ‘Cache’ array *C* of size *N*

****Sub-Problem****

* is the shortest distance between start node S and node j.
* Recursive function for intermediate node i:
* Solution at:

****Algorithm****



****Examples****

Starting Node: A

Finish Node: E

Connected Graph

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Node | A | B | C | D | E |
| 0 | -1 | 17 | -1 | 10 | 18 |
| 1 | 17 | -1 | 4 | 1 | 8 |
| 2 | -1 | 4 | -1 | 19 | 1 |
| 3 | 10 | 1 | 19 | -1 | 7 |
| 4 | 18 | 8 | 1 | 7 | -1 |

State

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Iteration \ Node | A | B | C | D | E |
| 0 - A | **0 \ -** | **17 \ -** | **- \ -** | **10 \ -** | **18 \ -** |
| 1 – D | 0 \ - | **11 \ D** | **29 \ D** | 10 \ - | **17 \ D** |
| 2 - B | 0 \ - | 11 \ D | **15 \ B** | 10 \ - | 17 \ D |
| 3 - C | 0 \ - | 11 \ D | 15 \ B | 10 \ - | 17 \ D |
| 4 - E | 0 \ - | 11 \ D | 15 \ B | 10 \ - | 17 \ D |

* 1. Experimental Comparison

Since the input of the problem is a graph, we can compare the performance of the 2 approaches with different graphs. We introduce the ‘density’ parameter which represents how many connections a node has with all the others. A density of ‘1’ means that all nodes are connected to each other, while ‘0’ means that each node has no adjacent nodes. However since the graph must be a connected graph, a density of ‘0’ is forbidden because, the minimum number of adjacent nodes of every node is 1.

How you choose the number of nodes? You need to explain a bit more how you obtain the graphs used in your experiments

* 1. Results and Discussion

Figure depicts the effect of problem size in execution time for both implementations

Figure depicts the effect of problem size in total memory usage (Gigabytes) and % memory usage for both implementations

* 1. Conclusions

In this instance the Results and Discussion may seem odd at first, however due to the very similar implementations of the algorithm the Results and Discussion start to make sense. Both the memory usage and the execution time look similar when the two approaches are compared, giving us no clear winner. The description of this algorithm is very precise, making the implementations of both approaches very similar, thus yielding similar Results and Discussion as well. Both the memory usage and the execution time have a quadratic growth, while the execution time grows much more gently.

**Chapter 8**

1. Independent Sets

|  |  |
| --- | --- |
| * 1. Description | 10 |
| * 1. Experimental Comparison | 10 |
| * 1. Results and Discussion | 10 |
| * 1. Conclusions | 10 |

* 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 all excluded nodes can 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.

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 rooted at a random node R (root). To solve this problem, we assume that no cycles exist in the transformed tree.

****Definitions****

* Graph of N nodes
* Rooted graph (tree) T
* A root node R

****Sub-Problem****

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

****Bottom-Up Approach (Iterative)****



****Top-Down Approach (Recursive)****



* 1. Experimental Comparison

How we obtained tree

* 1. Results and Discussion

Figure depicts the effect of problem size in execution time for both implementations

Figure depicts the effect of problem size in total memory usage (Gigabytes) and % memory usage for both implementations

* 1. Conclusions

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 and Discussion show that contrary to all previous Results and Discussion, the execution of the recursive approach seems to be much faster while consuming the same amounts of memory as the iterative approach. The Results and Discussion at first seem unreasonable, however after some thorough investigation on the implementations things start 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 and Discussion 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 and Discussion in what we see in the graphs, 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 a quadratic 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.

**Chapter 10**

1. K-Trees

|  |  |
| --- | --- |
| * 1. Description | 10 |
| * 1. Experimental Comparison | 10 |
| * 1. Results and Discussion | 10 |
| * 1. Conclusions | 10 |

* 1. Problem Description and Dynamic Programming Solution

This problem involves finding the number of subtrees of size K, from a tree rooted at R [citation]. 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.

Example

****Definitions****

* Tree of N nodes
* 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.**

****Bottom-Up Approach (Iterative)****



****Top-Down Approach (Recursive)****



* 1. Experimental Comparison

Since the input to the problem is a tree, we can compare the performance of the 2 approaches by changing the tree. To do this we introduce the parameter ‘C’ which represents the connectivity of the nodes. The connectivity of the nodes directly relates to the number of its adjacent nodes (child nodes). How? Explain more

* 1. Results and Discussion

Figure depicts the effect of problem size in execution time for both implementations

Figure depicts the effect of problem size in % memory usage for both implementations

* 1. Conclusions

We seem to find similar Results and Discussion to the K-Tree problem as the last problem. 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, making it unable to run the problem at bigger sizes. What we also see is that even though both approaches run in linear time, the iterative approach is also slower.

The last data point of the graph shows an immediate increase in the execution time, while the memory usage seems to drop a bit 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. The swap memory is far slower than the system memory, this explains the drop on the memory usage, since the program didn’t measure the swap memory used, this also explains the slower execution.

The recursive approach is a clear winner in both aspects.

**Chapter 10**

1. Tree Diameter

|  |  |
| --- | --- |
| * 1. Description | 10 |
| * 1. Experimental Comparison | 10 |
| * 1. Results and Discussion | 10 |
| * 1. Conclusions | 10 |

* 1. Problem Description and Dynamic Programming Solution

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

* + 1. ****Definitions****
* Tree of N nodes
* A root node R
* Caching arrays inc\_array and exc\_array of size N
  + 1. ****Sub-Problem****

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

**Diameter is the length of the longest path of a tree. A node i can either be included in this path or not. If a node is included then there are 2 cases. Either both ends of the path are part of its subtree, or only 1 part of the path is part of its subtree.**

**We calculate the “distance” of every node in DFS traversal, keeping track of the 2 largest values for every node. Then we calculate the max of the 2 values for every node.**

* + 1. ****Bottom-Up Approach (Iterative)****



* + 1. ****Top-Down Approach (Recursive)****



* 1. Experimental Comparison

Since the input to the problem is a tree, we can compare the performance of the 2 approaches by changing the tree. To do this we introduce the parameter ‘C’ which represents the connectivity of the nodes. The connectivity of the nodes directly relates to the number of its adjacent nodes (child nodes). Explain construction of trees

* 1. Results and Discussion

Figure depicts the effect of problem size in execution time for both implementations

Figure depicts the effect of problem size in % memory usage for both implementations

* 1. Conclusions

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

**Chapter 11**

1. Conclusion

|  |  |
| --- | --- |
| * 1. Summary | 10 |
| * 1. Problems Faced | 10 |
| * 1. Future Work | 10 |

* 1. Summary

After close inspection of the obtained results we came to some surprising conclusions. Firstly, as expected the type of the solution affects the performance of either approach, however, even the type of the input of the problem plays a huge role in its performance. We will categorize the problems into the following categories: Input and Solution 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. An exception to this is the MCSS problem. This problem required much more memory for the recursive approach making it a total win for the iterative approach for both aspects |
| 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. For this problem the memory usage was similar for both approaches, however a slight overhead for the execution time of the iterative approach was found, making it slightly slower. |
| 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. |

* 1. Problems Faced

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.

**Smaller sentectes**

The biggest problem faced however was the time it took to collect the data. When dealing 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 I had to change my input sizes, however at first I wouldn’t, at least for the MCSS problem, because I wanted to take the measurements of every program to its upmost limit. However after having to take the measurements a couple of times, then later finding out that my memory measurements were incorrect because of having not enough memory, I decided to change the problem sizes.

* 1. Future Work

I would like to work on more problems, finding out about the behaviour of the recursive approach on different data structures was not something I suspected at first. However just when I begun developing the program I realised how much harder it was. I would like to search for more similar behaviours.

An interesting extension of this work would be to further investigate what other parameters would affect the performance of the recursive approach.

How about programming language? OS? Would these make a difference?

**Βιβλιογραφία**

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

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