**University of Cyprus**

**Department of Computer Science**

**Dynamic Programming:**

**An experimental analysis and comparison of the**

**Top-Down**

**and**

**Bottom-up**

**approaches**

**Nicolas Zachariou**

# Introduction

Dynamic Programming is a family of algorithms that generally works by dividing a problem into sub-problems. To achieve this, the state or result of each sub-problem needs to be stored into some form of data structure. The goal of this study is to analyze and compare different approaches in terms of resource usage, efficiency & execution time. To do this multiple tools were used to extract memory usage data, CPU usage and execution time. Different types of optimizations were also studied, these include optimizations in the data structures used, and other algorithmic optimizations.

Dynamic Programming algorithms are divided into 2 categories. These are the Iterative (also known as Bottom-up) and the Recursive (also known as Top-down) approaches. The key in both categories is using a data structure (usually a table), to store the result of previously solved sub-problems to solve more complex sub-problems, repeating this process until the actual problem is solved.

The goal of this reseach is to study a variety of algorithms using both approaches to find possible patterns, reach some conclusions and classify these algorithms in terms of their performance, both regarding execution time and memory usage for both approaches. What affects an algorithms performance is its input complexity, and the form of its input data. Another factor is the complexity of the solution. The study, solutions and the results of each and every problem presented will be shown, as well as some analysis and some conclussions.

# Background

## Approach Comparison

Dynamic programming problems generally have a common attribute: a description of an optimal solution. This description can be used to divide the 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 once and find the solution to the problem without uneeded computations.

There are generally two approaches as to how we split the problem and how the solution is formed. Generally the easiest to understand approach involves a recursive equation / description, this approach is thought to have the worst performance of the two because of its recursive nature. The other approach involves a more algorithmic approach. 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. This is the essence of dynamic programming.

Bottom-Up is also known as an iterative approach, it uses some baseline cases to build into more complex sub-problems, using previous solutions every time to calculate the next ones. In the form of an array this approach generally start by ‘building’ or ‘filling’ the array from the ‘bottom’ or the ‘start’ until the end. The array represents the states or sub-problems of the problem, therefore we can visuallise this approach as an approach that computes all problems in an order until all the required states for the problem are computed. The solution is calculated when the value of the final sub-problem is found. This process is what is known as ‘Tabulation’.

Top-Down is the approach that directly represents the recursive solution of a problem. The recursive appoach describes the optimal solution of any problem. By using this description we can divide any problem into its direct, optimal sub-problems. This division is also true for any sub-problem, thus for every sub-problem we apply the recursive equation until the sub-problem is small enough, or equal to a base case where a value can either be calculated or returned. Through this process the information is returned and then used to solve bigger sub-problems. However, recursive descriptions generally have a common pitfall, many sub-problem solutions may be required in different parts of this recursive division, meaning that some sub-problems may be computed more than once. This is where memoization comes into play. This is the technique used in dynamic programming where whenever a state or an oprtimal value is computed, it is stored for future use, making sure that no sub-problem is calculated more than once. The recursive nature of this approach generally yields worse results in regards to execution time. This is also thought to be the case in regards to memory usage, since it is known that the deeper the recursion the more the stack usage.

## Tools Used

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

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 was used to monitor cpu utilization specificly.

To monitor execution time the standard libraries of C++ (Time library) were used to capture the execution time of the algoritms only, exactly before and after execution. This means that no memory allocation overhead is measured since in both approaches the table used has the same dimensions.

To monitor memory usage the Valgrind tool was used, specifically the massif version of the tool was used because it allows for stack monitoring. The tool works by creating memory snapshots of the program, and with each snapshot the heap usage is measured. Stack usage is measured more precisely thus slowing down the execution time vastly.

All scripts created for linux do not have any effect in execution and were just used to aid in the collection of the data.

# Most Common Sub Sequence

## Description

The Most Common Sub-Sequence problem regards two letter sequences A and B and seeks the biggest sequence of letters found in both A and B (not necessarily consecutive) in order.

* 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

P(i,j) is the most optimal solution for the first i-th letters of sequence A and j-th letters of sequence B. Since P(i,j) is the most 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 )
  1. therefore:
* P(i,-1) = P(-1,j) = 0 (end case)
* P(i,j) = { if A(i) == B(i) then P(i-1, j-1)+1, else max( P(i-1,j), P(i-1,j), P(i-1,j-1) ) }
* The actual problem is solved when P(N,M) is solved.

## Bottom-Up Approach (Iterative)

Iterate through array elements 1-by-1, and calculate the value of the MCSS at every point of the sequences using the letter of sequence A and B and the already calculated values within the array. We build the array from (0,0) to (N+1,M+1). An extra column and row is used to represent the end case. Each sub-problem is solved once.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| sequence |  | 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 |

## 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 0 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 |

## Scenarios – Preparation

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.

To test the efficiency of the program we will test 3 different sequence lengths. Firstly, when sequence A length amounts for of sequence B’s, similarly we will check for and . Since the problem is symmetrical it means that by testing sequence A and B we have the results 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.

## Results

## Conclusion

# Longest Increasing Sub Sequence 1D & 2D

## Description

This problem is 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 can be solved in 2 manners. A two dimensional table can be used to store the values of the sub-problems just like in MCSS (Chapter 3.). However a more efficient solution also exists where a one dimensional table can be used. In the first case the two dimensions are used to represend the length of the largest increasing sub-sequence starting from a given start-point of the sequence to a given end-point. However this can be ommited by using a single table and a different approach.

* Sequence of length N
* Maximum Value M

## 4.1.1 2-Dimenisonal solution

Each dimension of the solution represents an index of the sequence. One dimensions represents the start point of a sequence and the other one its end point.

## 4.1.1.1 Bottom-Up approach

Int cache[PROBLEM\_SIZE] = {0}

for every start index i: 0..n

for every end index j: i+1..n

array[i][j] = cache[i]+1

if(cache[j] < array[i][j]) cache[j] = array[i][j]

## 4.1.1.2 Top-Down approach

f(prev, curr) {

0 if curr == n,

f(prev, curr+1) if val(prev) < val(curr),

array[prev][curr] = max( (prev, curr+1), 1+ (curr, curr+1) ) otherwise

}

## 4.1.2 1-Dimenisonal solution

## 4.1.2.1 Bottom-Up approach

for every end index i: 1..n

for every start index j: 0..i

if ( val[i] > val[j] && array[i] < array[j] + 1) then array[i] = array[j] + 1

## 4.1.2.2 Top-Down approach

## Scenarios – Preparation

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)

## Results

## Conclusion

# Chain Matrix Multiplication

## Description

Let 2 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. 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).

* List L, of Matrices N WixHi
* ( L(i).w, L(i).h ) are the dimensions of the i-th matrix in the list, where ‘w’ is it’s width and ‘h’ is it’s 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

We initialize the ‘step’ variable to 1, and ‘i’ to 0 then we iteratively solve P(i,i+s). Each sub-problem is solved once.

## Top-Down Approach

Pseudo-recursion, actually same algorithm as 3.4, results are the same

## 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 |

## Scenarios – Preparation

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

## Results

## Conclusion

# 0-1 Knapsack

## Description

The problem 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.

* 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 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.

## 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

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

like 6.3 but we find the problems with Top-Down manner.

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

## Scenarios – 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].

## Results

## Conclusion

# Dijkstra

## Description

The problem is to find the shortest path between 2 specified nodes in a graph of N nodes. The start node S and finish node T are known. The idea is to start with node S and move to the closest node, with every step all neighboring nodes are explored to find the next closest node. When N iterations are done the closest path to T is found. If the cost of every connection between any 2 nodes is the same, we can assume that the closest path to T is found when the next closest node is T even if less than N iterations are done. To guarantee that a path between S and T exists, the Graph should be connected.

* Graph G of N bodes
* Nodes S (Start node) and node T (Finish node)

## Sub-Problem

* P(i) is the state of the problem after the i-th iteration. The state contains N values, each representing the closest path from S to every other node (after i exploration cycles).
* P(N) contains the solution

## Bottom-Up Approach

We build the array in bottom-up fashion, starting from node S, we check if the node provides any new shortest paths to other connected nodes. After every iteration we find the next unexplored node with the shortest distance from S to repeat the process until all nodes are explored.

## Top-Down Approach

The array is build in a similar way, like 5.4

## Example

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 |

## Scenarios – Preparation

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.

## Results

## Conclusion

# Independent Sets

## Description

Find the largest independent set of a graph of N nodes. An independent set is a set of nodes within the same graph, in which there are not nodes directly connected to each other. The idea is that when a node is included in the independent set, its children cannot be included (thus, its children are excluded from the solution). However, this also means that its grandchildren can be included in the tree. To traverse the graph, we have to convert it into a tree.

* Graph of N nodes
* A root node R

## Scenarios – Preparation

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.

## Results

## Conclusion

# K-Trees

## Description

Find 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.

* Tree of N nodes
* A root node R

## Scenarios – Preparation

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).

## Results

## Conclusion

# Tree Diameter

## Description

Find the diameter of a tree rooted at R. The diameter is the maximum distance between 2 nodes inside the tree.

* Tree of N nodes
* A root node R

## Scenarios – Preparation

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).

## Results

## Conclusion

# Appendices

# Conclusions

## Summary

## Problems Faced

## Future Work