A Comparison of Algorithms in Solving Large Instances of the 0-1 Knapsack Problem

Ross Langan

Submitted in partial fulfilment of
the requirements of Edinburgh Napier University
for the Degree of
Computer Science

School of Computing

April 2020

Authorship Declaration

I, Ross Langan, confirm that this dissertation and the work presented in it are my own achievement.

Where I have consulted the published work of others this is always clearly attributed;

Where I have quoted from the work of others the source is always given. With the exception of such quotations this dissertation is entirely my own work;

I have acknowledged all main sources of help;

If my research follows on from previous work or is part of a larger collaborative research project I have made clear exactly what was done by others and what I have contributed myself;

I have read and understand the penalties associated with Academic Misconduct.

I also confirm that I have obtained **informed consent** from all people I have involved in the work in this dissertation following the School's ethical guidelines

Signed: Ross Langan

Date: 08/04/2020

Matriculation no: 40276526

General Data Protection Regulation Declaration

Under the General Data Protection Regulation (GDPR) (EU) 2016/679, the University cannot disclose your grade to an unauthorised person. However, other students benefit from studying dissertations that have their grades attached.

Please sign your name below *one* of the options below to state your preference.

The University may make this dissertation, with indicative grade, available to others.

Ross Langan

The University may make this dissertation available to others, but the grade may not be disclosed.

The University may not make this dissertation available to others.

Abstract

The knapsack problem is a combinatorial optimisation problem with the goal of maximising the value of items included in a knapsack without exceeding the capacity. The 0-1 knapsack problem is a variation of this problem and an active area of research. Analysis of current research shows that several well-known algorithms in solving the problem have not been investigated in solving large instances of the problem. This study aims to investigate this matter, determining the performance of each algorithm in comparison to each other and whether each is suitable for solving large instances of the problem. Building on existing studies, it asks: Which of the algorithms performs best in solving large instances of the 0-1 knapsack problem? In the context of this study, the performance is determined by the solution, the time and memory usage, and the efficiency in finding good solutions.

Each of the four algorithms was optimised and tested on large problem instances with up to 100,000 items and a range of capacities. Analysis of the results reveals that both the branch and bound and genetic algorithms are inefficient at solving large instances of the problem. The dynamic programming algorithm is shown to find the best solutions for small-medium sized problems, however, suffers from very high memory usage in some large problems. The greedy algorithm proves to be the best algorithm for solving large problems with both low time and memory usage and either optimal or near-optimal results consistently, making this the preferred choice of algorithm for large problems.

Contents

1 Introduction 10

1.1 The Knapsack Problem 11

1.2 Project Aims 13

2 Literature Review 14

2.1 Introduction 14

2.2 Computational Complexity 14

2.3 Datasets 15

2.4 Measurables 16

2.5 Algorithms 17

2.6 Algorithm Comparisons in Existing Literature 24

2.7 Conclusion 25

3 Technical Review 28

3.1 Introduction 28

3.2 Project and Algorithm Requirements 28

3.3 Algorithm Choice and Variations 30

3.4 Language and IDE Choices 33

3.5 Conclusion 34

4 Methodology 36

4.1 Introduction 36

4.2 Global Parameters 36

4.3 Evaluation and Examination 37

4.4 Testing Methodology 40

5 Algorithm Implementation and Variation Testing 43

5.1 Generating and Processing the Data 43

5.2 Branch and Bound Algorithm 44

5.3 Dynamic Programming Algorithm 45

5.4 Genetic Algorithm 47

5.5 Greedy Approximation Algorithm 60

6 Results/Evaluation 67

6.1 Introduction 67

6.2 Results 67

6.3 Comparison to Results in Existing Studies 80

6.4 Conclusion of Results 85

7 Conclusion 88

7.1 Future Work 91

List of Tables

Table 1. Example of Results Table 42
Table 2. Operator Types and Parameter Settings for GA 1 48
Table 3. Operator Types and Parameter Settings for GA 2 50
Table 4. Operator Types and Parameter Settings for GA 3 51
Table 7 69
Table 6 69
Table 5 69
Table 9 70
Table 8 70
Table 10 70
Table 11. 50000 Item Problems 80
Table 12. 100,000 Item Problems 80
Table 13. Algorithm Performance Measures 87

List of Figures

Equation 1. The 0-1 Knapsack Problem 12

Figure 1. Structure of the Test .in File..... 43
Figure 2. Dynamic Programming Pseudocode .....**Error! Bookmark not defined.**
Figure 3. Experiment 1, Optimum Value: 6105..... 52
Figure 4. Experiment 2, Optimum Value: 45973..... 54
Figure 5. Experiment 3, Optimum Value: 57029..... 55
Figure 6. Representation of a Solution 56
Figure 7. Genetic Algorithm Fitness Function..... 57
Figure 8. Group Selection Operator..... 58
Figure 9. 1-Point Crossover Operator..... 59
Figure 6. Experiment 1, Optimum Value: 6105..... 61
Figure 7. Experiment 2, Optimum Value: 45973..... 62
Figure 8. Experiment 3, Optimum Value: 57029..... 62
Figure 9. Wall Clock Time..... 63
Figure 10. Memory Consumption..... 63
Figure 11. Greedy Approximation Algorithm Pseudocode..... 66
Figure 12. Low Capacity Problems..... 72
Figure 13. Medium Capacity Problems..... 72
Figure 14. High Capacity Problems 73
Figure 15. Genetic Algorithm Generations per Second 74
Figure 16. Genetic Algorithm Wall Clock Time 74
Figure 17. Dynamic Programming Memory Usage for Medium-High Capacity Problems 76
Figure 18. Memory Usage for Large Problems..... 79
Figure 19. Memory Usage for Large Problems..... 79

Acknowledgements

I would like to thank my project supervisor, Dr. Thomas Methven for his continued interest and support at every stage of the project. I would also like to thank my partner, Olivia for continued support throughout.

1 Introduction

The knapsack problem is a classic combinatorial optimisation problem in which given a collection of items, the goal is to maximise the value of items placed in a knapsack, or backpack, while staying within its capacity. There are many different variations of the problem, but the purpose is always the same- remain within the knapsack capacity and maximise one or more constraints, usually at least the value of the selected items. The problem has been studied extensively since the pioneering work of mathematician Tobias Dantzig in the late 1800s and remains a challenging area of research (Feng, Jia, & He, 2014). In recent years with the advancement of technology and optimisation techniques there has been an influx of new and faster methods of solving the knapsack problem.

The 0-1 knapsack problem is one of the simplest variations of the knapsack problem in which given a set of *n* items, maximise the value of the items placed in the knapsack while staying within the capacity. Each item can only be selected once and cannot be split into smaller fractions, hence the '0' for excluding an item and '1' for including an item. It has immediate applications in industry and financial management such as cargo loading, stock cutting and budget control.

There are several types of algorithm that exist for solving the 0-1 knapsack problem, however the performance of many in solving large datasets remains largely undocumented. In this study four algorithm types are focussed on: branch and bound, dynamic programming, genetic algorithms and greedy approximation algorithms. Each has been selected because of their popularity in literature and the lack of studies performed on their ability to solve large instances of the problem. Dynamic programming is the only algorithm included in this study that can guarantee optimality.

In this study, factors such as the time and memory usage of each algorithm will be recorded and used to compare the performance of each algorithm in solving instances of the knapsack problem. Both the capacity of the knapsack and the

number of items will be changed in order to investigate the effects changing these values has on the performance of the algorithms. The problems will range from 50-100,000 items, and the capacity of the knapsack will also range between 50 and 100,000. In this study, large problems are deemed any problem with over 50,000 items.

In the rest of this section is included a detailed description of the problem and its variations, as well as the project aims. The rest of the document contains a review of existing literature surrounding the problem and the algorithms, a technical review stating the project requirements and technical choices relating to the development of algorithms and results, a methodology containing details about algorithm performance measures and the testing stage, an implementation section detailing the implementation of each algorithm, followed by the results and concluding remarks.

1.1 The Knapsack Problem

There are both single and multi-objective variations of the knapsack problem involving one and multi-dimensional knapsacks (Kumar & Banerjee, 2006). Both the single objective and multi-objective cases have been studied extensively over the past century from a theoretical and practical point of view, as described by Sahni (1975), Li, Liu, Wan, Yin, & Li (2015), Della Croce, Salassa, & Scatamacchia (2017) and Feng et al. (2014). The knapsack problem continues to be a challenging area of research and has become one of the staple problems of combinatorial optimisation because of the simplistic nature and yet complicated workings of the problem. In the last few decades particularly there have been advances in the refinement of algorithms resulting in larger instances of the problem to be solved faster (Martello, Pisinger, & Toth, 2000).

Much of the pioneering research into the knapsack problem in the past three decades has been performed by D. Pisinger of the Technical University of Denmark. Pisinger's work on the knapsack problem and both classic and novel algorithms for solving it have been referenced heavily in other literature due to the innovative studies he has performed, and many of the proofs and elements of his work can be seen directly in many recent studies (Salles da Cunha, Bahiense, Lucena, & Carvalho de Souza, 2010).

The classic 0-1 knapsack problem is the most common form of the knapsack problem- given a set of *n* items, each item *j* having a weight w_j and a value v_j, choose the optimal subset of items to include in a knapsack of capacity *c* so that the total weight is less than *c* and the sum of the values is maximised. The 0-1 variation restricts the number of copies of each item to zero or one and can be formulated as follows:

$$maximise \sum_{j=1}^{n} v_j x_j$$

$$subject\ to \sum_{j=1}^{n} w_j x_j \leq c$$

$$x_j \in \{0,1\},\ \ j = 1, \dots, n,$$

Equation 1. The 0-1 Knapsack Problem

The binary decision variables x_j are used to indicate whether the item *j* is included in the knapsack or not (Pisinger, 2005). It can be assumed in most cases that the weight and value of each item are non-negative, and the weight is less than the capacity of the knapsack. It is this variation of the problem that will be discussed further in this review and used in this study.

Including the 0-1 knapsack problem, there are three main variations of the problem:

- Binary (0-1) Knapsack Problem
- Bounded Knapsack Problem (BKP)
- Unbounded Knapsack Problem (UKP)

In each variation of the problem the goal is always to maximise v_j by choosing the optimal set of items. In the 0-1 knapsack problem each item can only be included in the optimal set at most once. If we allow x_j to be a finite number and allow each item to be included between 0 and x_j times, then it becomes the bounded knapsack problem. This problem variation applies well in situations where there is a stock of

items. If we allow x_j to be infinite the problem becomes the unbounded knapsack problem. In this problem an unlimited amount of each item can be placed in the optimal set (Cho, 2018, p.47).

Historically the knapsack problem was studied intensively for its theoretical uses in operational research, discrete mathematics and computer science (Feng et al., 2014). It was primarily applied to loading problems (Sahni, 1975) and in the stock cutting industry for reducing waste material (Gomory, 1961), however practical applications of the knapsack problem are not limited to packing problems. It is now applied to capital budgeting, crew scheduling and budget control and in various cryptographic schemes (Li, Liu, Wan, Yin, & Li, 2015), and more recently, it has been found to have uses in the smart home sector (Della Croce et al., 2017).

While the application of the problem goes mostly undocumented in literature, as recently as 2015 the 'knapsack problem with setup', a slight variation of the 0-1 knapsack problem has been applied to assist suppliers in the Tunisian glass production industry in deciding how to choose which orders to prioritise as so to maximise the total profit. A dynamic programming approach was developed for this specific case by Chebil & Khemakhem (2015) in order to select the most profitable orders, and highlights the relevance of the problem still to many industries today.

1.2 Project Aims

The aim of this project is to investigate the performance of four algorithms in solving instances of the 0-1 knapsack problem. Specifically, this project aims to include problems larger than those found commonly in literature in order to provide a link between existing literature and a new insight into the performance of algorithms in solving large instances of the problem. Below are three project aims which I will work towards satisfying throughout the course of the project.

- A1 Design and implement four algorithms and optimise them for large problems
- A2 Test the algorithms on a range of problem types with varying numbers of items, coefficients and capacities
- A3 Draw and present verifiable results from testing the algorithms

2 Literature Review

2.1 Introduction

This literature review presents a survey of existing literature regarding the 0-1 knapsack problem- a popular NP-hard combinatorial optimisation problem. The objective of this document is to critically discuss the key findings, concepts and developments in existing studies in reference to the application of the branch and bound, dynamic programming, genetic algorithm and greedy approximation algorithm.

Included in this review is the computational complexity of the problem, the measurables involved and a detailed look at the existing use of each algorithm in solving the 0-1 knapsack problem in literature, specifically outlining areas of disagreement between studies and gaps in research.

2.2 Computational Complexity

Computational complexity is the classification of computational problems and a measurement of how these classes compare to each other. While the decision problem is NP-complete, the optimisation problem of the knapsack problem and its variations are classed as NP hard. NP-hard complexity is also known as non-deterministic polynomial time and has the characteristic that there is no polynomial algorithm which can tell, given a solution, whether it is optimal.

In existing literature it is common for the optimisation problem of the knapsack problem to be referred to as NP-hard, however D. Pisinger‘s ‘Where are the Hard Knapsack Problems’ defines the problem further as “NP-hard in the weak sense” (Pisinger, 2005, p. 2272). This classification of the problem by Pisinger is referenced in other studies (Della Croce et al., 2017; Salles da Cunha et al., 2010), and specifies that there exists a pseudo-polynomial time algorithm that can solve it; the pseudo-polynomial algorithm being the dynamic programming method.

2.3 Datasets

An instance of the knapsack problem consists of a dataset with at minimum a finite number of items with a weight and value and a known finite knapsack capacity. While the most common datasets consist of uniformly randomly distributed weights and values, there exists different types of data in which the weights and values of each item are correlated to each other. In strongly correlated data for example, the value of each item corresponds to its weight plus a fixed constant (Pisinger, 1988). Higher correlation in datasets massively increases the difficulty of the problem and can make it harder for algorithms to solve in reasonable time. In this study however, correlated datasets are not in the scope of the project, as this may make it more difficult to compare the algorithm's performances. Instead, the datasets used in this project will be randomly generated and contain no deliberate correlation.

Most of the datasets used in literature pre-1985 are up to 1000 items in size, with the exception of a 1979 study presenting an algorithm for large knapsack problems which tests on datasets up to 10,000 items and a range of coefficients (Balas & Zemel, 1979). In modern studies, datasets for this problem are usually between 500 and 10,000 items, with only one study testing on large datasets of 100,000+ items (Shaheen & Sleit, 2016). Advancements in computing power mean that testing on larger datasets is easier and can usually be completed in a reasonable amount of time. Further advancements in algorithms specific to the knapsack problem also mean that problems with larger datasets are becoming easier to solve quickly and accurately.

There is a range of sources for the datasets used in literature, however nearly all the data is randomly generated as opposed to manually entered or taken from real sources. D. Pisinger's 'problem generator' featured first in 'Core Problems in Knapsack Algorithms' (Pisinger, 1999, p. 12) is a C program used by other researchers to generate data with specific ranges of coefficients, correlation types and sizes (Salles da Cunha et al., 2010). Other studies also mention the use of random generators to produce their test data however do not name the source, likely because they are created by the researchers themselves. A study comparing different approaches in solving the 0-1 knapsack problem also claims to use well

known benchmark datasets to measure the accuracy of the results found but does not disclose the source of the benchmark datasets, making it extremely difficult for other researchers to emulate the results (Shaheen & Sleit, 2016). Although there is a lack of sources for datasets used in literature, all of the datasets are assumed to be randomly distributed and may implicitly contain some structure which may not be present in real-life instances (Pisinger, 2005).

As first discussed by Pisinger, there are very few real-life instances of the knapsack problem in literature which has led to algorithm design being focussed primarily on synthetic benchmark tests (Kumar & Banerjee, 2006; Pisinger, 2005). The closest we can get to emulating real-life instances is restricting dataset sizes, coefficients and the correlation type. This leaves a huge gap for algorithm design to be focussed on real instances of the knapsack problem, however finding data representing real instances of the problem is likely to be very difficult as this would involve collecting potentially sensitive data from businesses. Observing any changes to how algorithms are designed based on solving predominantly real-life instances would be an interesting field of future research for the knapsack problem.

2.4 Measurables

In order to test the performance of any algorithm it is essential to measure certain aspects of its operation and the results it produces. Interestingly, in existing literature there are a range of different approaches to measuring performance. In certain studies for example, the performance measures focus on the best value found, the selection of items chosen and how close the value found is to the optimum solution (Bhattacharjee & Sarmah, 2014; Feng et al., 2014; Hristakeva & Shrestha, 2005; Shaheen & Sleit, 2016; Zhao, Zhao, & Zhang, 2006). In other studies more pressure is put on the CPU time and how long the algorithm takes to present the solution compared to others.

Generally in the literature the measurements recorded when comparing an algorithm to another algorithm are the CPU time and maximised value (Salles da Cunha, Bahiense, Lucena, & Carvalho de Souza, 2010; Pisinger, 1995; Li, Liu, Wan, Yin, & Li, 2015; Kulkarni & Shabir, 2016; Bhattacharjee & Sarmah, 2014; Della Croce, Salassa, & Scatamacchia, 2017). Although computational time is used as a

measurement in three of these studies, as there is no further definition of the term in any of the literature, it will be assumed in this section that computational time is the same as CPU time- the amount of time in seconds the CPU spends executing the task, including the time spent executing run-time and system services on its behalf.

While CPU time is an accurate measurement, it is difficult to compare how effective an algorithm is compared to another if there is a considerable amount of time between the algorithms being tested. Comparing an algorithm which solves the problem in 100 milliseconds to an algorithm that solves the problem in 60 seconds can be difficult, especially as between studies it is almost guaranteed that they have been tested on different datasets. Because each separate study usually performs experiments on different sizes of problem, with different knapsack capacities and using different computers, comparing even the most basic results can be difficult and potentially inaccurate. Additionally, there are no ‘standard’ problem sizes for testing algorithms, meaning while some studies measure on problems with 1000 items, another might only test on problems with 1250 items, further complicating the comparison of results between studies. For this reason, testing on as many numbers of items as possible should make it easier to compare the results to those in other studies.

2.5 Algorithms

2.5.1 Branch & Bound

Branch and bound, the first of the four methods covered in this study, is an algorithm paradigm used for solving combinatorial optimisation problems. Discovered in the 1960s and first appearing in knapsack problem solving in the early 70s (Martello et al., 2000), the algorithms are based on the construction of a state space tree which is traversed to find an optimal or near-optimal solution. Since its discovery several branch and bound algorithms have been published featuring different traversal and bounding methods or hybridised with other algorithms (Pisinger, 2005; Plateau & Elkihel, 1985).

In a state space tree, each level of the tree represents a choice in the solution space that depends on the level above and partial solutions are generated one at a time (Shaheen & Sleit, 2016). Each branch in the tree is checked against upper and lower

bounds in order to determine whether it is worth exploring or not and branches that have poor solution qualities are pruned (not explored any further). The process of splitting the problem into smaller sub-problems and pruning the tree in order to discard poor or infeasible solutions is continued until the optimal or near-optimal solution is found.

For the algorithm to determine which branches to explore and which to prune, each branch is checked against upper and lower estimated bounds on the optimal solution (Hristakeva & Shrestha, 2005). While there are several techniques for determining bounds, the best bounds according to a study into 'New Trends in Exact Algorithms for the 0-1 Knapsack Problem' are bounds based in Lagrangian relaxation, partial enumeration, construction of valid additional constraints and different relaxations of the latter (Martello et al., 2000). In literature covering branch and bound in solving the 0-1 knapsack problem, linear programming bounds (LP-bounds) determined by linear programming appear to be the prevalent method as seen in algorithms featured in several studies (Shaheen & Sleit, 2016; Martello, Pisinger, & Toth, 2000; Hristakeva & Shrestha, 2005; Cho, 2018, p.47-63). This technique produces simple bounds which are computationally cheap to produce (Pisinger, 2005), as opposed to Lagrangian bounds which are computationally expensive but produce tighter bounds closer to the supremum and infimum. The use of Lagrangian bounds can be seen in a pure branch and bound algorithm developed by Salles da Cunha, Bahiense, Lucena, & Carvalho de Souza (2010) and a dynamic programming hybrid algorithm 'MThard' seen in Pisinger's 'Where are the Hard Knapsack Problems?' (Pisinger, 2005).

The pure branch and bound algorithm designed in the study by Salles da Cunha, Bahiense, Lucena, & Carvalho de Souza, (2010) is based on a Lagrangian relax-and-cut procedure and is specifically aimed at solving instances with large coefficients up to 10^{16} . Tests of the algorithm however were mixed as it performed well on some instances with large coefficients and performed poorly on two test instances found in existing literature. The MThard hybrid algorithm discussed in Pisinger's study performs very similarly, performing very well on nearly all problems but producing anomalies for some large sized almost strongly correlated problems where the bounds fail for an unknown reason (Pisinger, 2005). According to Salles da Cunha, Bahiense, Lucena, & Carvalho de Souza (2010) Lagrangian bounds tend to

outperform LP-bounds, however as discussed by Martello, Pisinger, & Toth (2000), tight bounds are expensive to derive and this may be the reason why LP-bounds are much more prevalent in algorithms found in literature. Additionally, there is a lack of any literature directly comparing the bounding methods to each other which may lead researchers to choose the simpler method to implement. As this study is particularly focussed on solving large instances of the knapsack problem, Lagrangian bounds may be the preferred method, as they are claimed to have better performance than LP-bounds and may help in solving larger problems.

The method of traversal can also differ between algorithms. The most common methods, breadth-first and best-first traversal both stop searching a branch of the tree if it is not worth expanding, however for remaining nodes breadth-first uses a regular queue and best-first uses a priority queue, which assigns each value a priority allowing for these values to be served first. The method of traversal is not discussed in many instances of the algorithm in literature. While the Lagrangian based branch and bound algorithm featured in the study by Salles da Cunha, Bahiense, Lucena, & Carvalho de Souza, (2010) utilises a breadth-first approach, the best-first approach can only be seen once in use for the 0-1 knapsack problem (Shaheen & Sleit, 2016) and is more commonly found in literature relating to the max-min 0-1 knapsack problem and multi-dimensional knapsack problem (Sbhibi, 2007; Iida, 2009; Razzazi & Ghasemi, 2008). This means that future research on the branch and bound algorithm could first determine which approach is best suited to the 0-1 knapsack problem, as well as which bounding method is best before designing an algorithm to use.

An interesting trend that can be seen in literature is the limitations placed on branch and bound algorithms due to memory and time constraints. While the time complexity of branch and bound algorithms are not proven, the running time of the algorithm and the memory use increases drastically as the number of items increases (Shaheen & Sleit, 2016). Researchers can be seen limiting the amount of time the algorithm can run for or the number of items in the problem. The algorithm based on Lagrangian bounds implemented by Salles da Cunha, Bahiense, Lucena, & Carvalho de Souza (2010) is allowed to run for a maximum of 36 seconds, and if the optimum value is not found in this time, the instance is deemed unsolved, perhaps due to the amount

of memory required after this point. A study performed by Hristakeva & Shrestha (2005) places a limit of 750 items due to memory constraints. Alternatively, experiments performed by Shaheen & Sleit (2016) limit the number of items to 60,000 and the capacity to 100, however branch and bound algorithms solving this size of problem cannot be seen in any other studies raising questions about the authenticity of this study, and raising the question of how large a problem a branch and bound algorithm is suitable for solving. The range of limitations placed on the algorithm compared to the lack of limitations placed on other algorithms leaves a huge gap in research to determine the upper bounds for problem sizes on modern machines, and whether this can change based on the performance of the computer.

2.5.2 **Dynamic Programming**

Dynamic programming, introduced by Richard Bellman in the 1950s is an optimisation method which uses a divide-and-conquer approach to break down problems into smaller subproblems, the solutions of overlapping subproblems of which are then reused to avoid the recalculation of solutions (Chebil & Khemakhem, 2015). This reuse of subproblem results reduces the algorithm's time complexity from exponential to polynomial time and is the method responsible for the 0-1 Knapsack problem's NP-hard time complexity (Pisinger, 2005).

Dynamic programming works by splitting the problem into sub-problems, solving each of the sub-problems and storing the results in a table which is then used to obtain a solution to the original problem, generally in a bottom-up approach (Shaheen & Sleit, 2016). The dynamic programming algorithm most ofter referred to and used in literature to solve the 0-1 knapsack problem is the original algorithm based on Bellman recursion (Balas & Zemel, 1979). Alternative algorithms have been presented by Toth (1980), Horowitz & Sahni (1974), and more recently Pisinger, who has proposed minimal algorithms based on dynamic programming and hybrid algorithms using elements of both branch and bound and dynamic programming (Chebil & Khemakhem, 2015).

Dynamic programming is able to solve problems optimally every time, however finding the optimal solution, particularly for harder problems can be slower and substantially more memory intensive (Chebil & Khemakhem, 2015). The algorithms are easy to implement as they do not require the use of any additional structures,

and according to Chebil & Khemakhem (2015) are efficient and easy to implement for small and medium-sized instances. Martello, Pisinger, & Toth (2000) combine dynamic programming with bounds, also seen in branch and bound algorithms, specifically to tackle large, strongly correlated datasets, highlighting the use of this algorithm for all problem types.

2.5.3 Genetic Algorithm

Genetic algorithms are metaheuristic search algorithms discovered in the 1960s that belong to the larger family of evolutionary algorithms. They differ from heuristic algorithms such as the greedy approximation algorithm as they have the means of escaping local optima to find better solutions. Inspired by Darwin's theory of evolution, the algorithms optimise solutions by relying on bio-inspired operators such as selection, crossover and mutation to make slight changes to solutions in order to evolve to better ones. In the knapsack problem, the population contains a set number of chromosomes, each of which is a solution to the problem. Each generation (iteration of the algorithm) we measure the fitness of each chromosome in the population and select two (or more) individuals from the population to be parents. Parents can have crossover applied before they are used to create a child which has a small chance of being mutated before being added to the next generation. As fitter parents are selected and fitter child chromosomes are added to the population, the algorithm moves towards better solutions.

In setting up the algorithm, Shaheen & Sleit (2016) and Hristakeva & Shrestha (2004) use a cell data structure with two fields (weight and value) to represent each item, and an array of cells to represent each chromosome. The index of the chromosome in the array can be mapped to a second array, where the value in that position is a 0 or 1, denoting whether the item is included in the knapsack or not. While there are other ways for representing the data for this problem, many of the academic papers featuring genetic algorithms do not reveal the way in which the data is represented.

While the exact method of storing the data does not affect the solutions found, the selection, mutation and crossover operators can have a huge influence on the effectiveness of a genetic algorithm and must be tweaked for every different optimisation problem. The study mentioned above- 'Solving the 0-1 Knapsack Problem with Genetic Algorithms' by Hristakeva & Shrestha (2004) tests both roulette

wheel selection and group selection, a novel selection operator designed for the knapsack problem, and found group selection to out-perform roulette wheel. While this study has tested more than one operator, other studies including genetic algorithms use primarily roulette wheel selection (J. Zhao, Huang, Pang, & Liu, 2009). Other operators such as tournament selection and rank selection are not prevalent in genetic algorithms for solving the 0-1 knapsack problem and would be a good area of further research.

There is a large range of different crossover operators used between studies, which determines how elements from the parent chromosomes are swapped to create new child chromosomes. J. Zhao, Huang, Pang, & Liu (2009) use uniform crossover and an unusually low crossover probability of 0.15, possibly due to the algorithm being based on a greedy strategy rather than a traditional genetic algorithm. Compared with the crossover probability of 85% in a study performed by Bhattacharjee & Sarmah (2014), and probabilities of 75%, 85% and 95% in 'Solving the 0-1 Knapsack Problem with Genetic Algorithms' (Hristakeva & Shrestha, 2004), regardless of the slight variation of algorithm used in the above study it might be useful to test lower crossover probabilities in future studies to determine whether an improvement in performance can be identified using different probabilities.

While the selection and crossover operators vary between studies, the mutation rate- the rate at which a child is mutated, remains roughly the same at around 0.1%, seen in the algorithms implemented by Bhattacharjee & Sarmah (2014), and Hristakeva & Shrestha (2004). This matches the suggested mutation rate of 0.1% found in 'An Introduction to Genetic Algorithms' (Mitchell, 1998).

A key area which I think has been excluded from many of the studies regarding genetic algorithms is the initialisation of the population, and the advantages of seeding the population. As the dataset must be read into the program, rather than randomly including certain items in order to create an initial population, we can choose some of the better items to include in the initial solutions so that the algorithm starts with a stronger population. This method of initialising the population with good solutions is only mentioned in two studies, both containing greedy-genetic hybrid algorithms (Zhao, Huang, Pang, & Liu, 2009; Liu, & Liu, 2012), however does not appear in any other literature. This may prove to be an important feature to consider

in this study for improving the performance on large datasets and would be an interesting area to start future research. Although this may deviate from the idea of using a ‘pure’ genetic algorithm, if there is considerable improvement using a greedy method for initialising the population, then it may be used in this study.

2.5.4 Greedy Approximation Algorithm

The greedy approximation algorithm was first presented as means of solving the unbounded knapsack problem in 1957 by George Dantzig. Greedy programming techniques are now widely used in many optimisation problems and are particularly common hybridised with other algorithms. In solving the 0-1 knapsack problem, there are 3 main greedy strategies an algorithm can use to calculate solutions. Each strategy is repeated until the capacity of the knapsack is full or there are no more items to add:

1. Sort the objects by weight ascending, then add items with the smallest weights
2. Sort the objects by value ascending, then add items with the highest values
3. Sort the objects by value-to-weight ratio, then add items with the largest ratio

(Hristakeva & Shrestha, 2005; Liu, & Liu, 2012)

Pure greedy approximation algorithms using these methods are not found as often in academic studies as the other three algorithms. This may be because this technique cannot guarantee the optimal solution or that other algorithms are widely known to be more effective. Although not abundant in literature, algorithms based on or making use of the greedy strategy are very common. Genetic algorithms with elements of the greedy strategy, as well as chemical-reaction optimisation algorithms and particle swarm hybridised algorithms are popular algorithms in current literature. These algorithms, using elements of greedy approximation appear to be an active area of research.

A pure greedy algorithm, using the value-to-weight method is featured in ‘Comparing between different approaches to solve the 0/1 Knapsack problem’ (Shaheen & Sleit, 2016), however as some of the results in this study are very high compared to similar studies, specifically the branch and bound as mentioned previously, the reputability of this study may be questionable. In order to confirm the results featured in this

study and accurately determine the performance of the algorithm, experimentation on the algorithm is required in this study.

2.6 Algorithm Comparisons in Existing Literature

In order to facilitate a comparison between the algorithms, measurements such as those mentioned previously in section 2.4 are recorded and used to determine the performance. Due to the stochastic nature of many elements of the algorithms it is essential when conducting tests to run each algorithm several times for each different test and to calculate an average from these results. Ideally, each algorithm will be written in the same language and will be tested on the same computer, ensuring that no algorithm has an advantage over another. In existing studies it is highly probable that algorithms are not all written in the same language and it is guaranteed that computers with different performance levels are being used for testing across different studies, however for the remainder of this section we will assume that the difference in results caused by these factors are negligible.

Direct comparisons between all four of the algorithms included in this study can only be found in two academic studies: ‘Different Approaches to Solve the 0 / 1 Knapsack Problem’ (Hristakeva & Shrestha, 2005) and ‘Comparing between different approaches to solve the 0/1 Knapsack problem’ (Shaheen & Sleit, 2016). Individual studies measuring the performance of each algorithm independently are plentiful for genetic algorithms and branch and bound, however studies on the dynamic programming and greedy approaches are less common, possibly due to the fact that these algorithms are used predominantly for hybridising with other algorithms.

Testing each algorithm on datasets of different sizes, it has been found by Hristakeva & Shrestha, (2005) that in small problems with a capacity of 50 and data coefficients of 1000, dynamic programming outperforms both branch and bound on memory consumption and basic operations performed. In an experiment with different capacities however, it was found that genetic algorithms outperform dynamic programming when the capacity of the knapsack is larger than the number of items. The second comparative study, performed by Shaheen & Sleit (2016), tests on large problems with 100,000–500,000 items, and determines that dynamic programming has a significantly higher execution time than genetic algorithms. It is unfortunate,

however, that key details such as the coefficient of the data are missing from this study, which can easily affect the results. The conclusion of both comparative studies show that dynamic programming algorithms are more suited to average knapsack problems, particularly because they are easy to implement, however for larger problems genetic algorithms are more suitable in terms of execution time and memory consumption.

In terms of the general performance of the algorithms there appears to be disagreement between different comparative studies. A study into genetic algorithms by Hristakeva & Shrestha (2004)- the same authors as the above comapative study- claims that since the knapsack problem is an NP problem, approaches such as dynamic programming and branch and bound are ‘not very useful’ for solving it. This claim is supported by Li, Liu, Wan, Yin, & Li (2015), who expand on this point by stating that dynamic programming based on Bellman recursion is not an effective approach to solving the problem because the space consumption of the algorithm is very large, and the worst-case and best-case computational efforts are generally the same. This analysis, however, fails to demonstrate that the memory consumption of dynamic programming is directly related to the number of items and the maximum capacity, and that for smaller problems dynamic programming does indeed use less memory than genetic programming. This is shown in the results of both comparative studies mentioned in the previous paragraph. It also does not consider that dynamic programming is the only technique to guarantee optimality.

With a range of conclusions as to the effectiveness of each method in several studies, it is left unclear exactly what scale researchers are using to determine how effective one algorithm is compared to another. This leaves a gap for credible research focussing on what factors are deemed to affect the suitability or effectiveness of one algorithm compared to the next.

2.7 Conclusion

Studies around the 0-1 knapsack problem and methods for solving it are plentiful, especially new proposals of novel algorithms designed to be faster and more efficient than ever. Clearly the knapsack problem is still a hot topic in combinatorial

optimisation and there is still a huge market for research on the problem, particularly in developing algorithms that can deal with larger datasets.

Many of the new algorithms feature a combination of both novel and more traditional methods, some of which are hybrids of dynamic programming, branch and bound and greedy approximation algorithms, and highlight the effectiveness of traditional tried-and-tested methods in solving the problem.

Few studies exist comparing the algorithms selected in this study, and the credibility of one of these studies is questionable. Additionally, the problem with the existing comparative studies is that they lack common ground- measuring the same factors, using the same types and sizes of data, using the same algorithm parameters, defining algorithm performance the same. Because of this both studies conclude something slightly different about each algorithm, making it extremely hard to determine solid, verifiable conclusions about the performance of each algorithm and how they compare to each other.

A particularly interesting area of research, and the purpose of this study, is applying the traditional approaches such as those detailed above to larger datasets and optimising them to handle datasets bigger than those used in almost all existing studies- up to 100,000 items. Conducting this experiment effectively and measuring some of the most common factors found in literature would allow the comparison of algorithms both within this study and with algorithms in different studies.

Another area of research is the testing of algorithms on real datasets from real-world sources. As data representing a real knapsack problem is extremely hard to source there lacks any literature or studies testing on this type of data. Finding a real set of data would present a novel instance of a problem for traditional approaches to solve and could highlight differences in algorithm performance not seen when solving generated data.

What is missing in both current and past studies regarding the traditional methods of branch and bound, dynamic programming, genetic algorithms and greedy approximation methods in solving the 0-1 knapsack problems is conclusive and accurate classifications of algorithm performance and comparisons between each

algorithm. This study will hopefully provide both of these and may help clarify some of the results in existing studies.

3 Technical Review

3.1 Introduction

As the 0-1 knapsack problem is a standard combinatorial optimisation problem it can be solved using a range of different algorithms. Behind the scenes there exists many different technologies and methods we can use to apply these algorithms. From the programming language to the development environments and plugins, additional software, and testing methods, many careful considerations must be made before algorithms can be developed. In this section of the project I will present the requirements of the project and use the knowledge I have gained from the literature review to make justified choices that will affect the course of development, testing and the presentation of results.

As details of the implementation are featured in section 5, the scope of this review does not cover specific cases in the development of each algorithm e.g. why the mutation rate in the genetic algorithm is set to a specific value. Rather this review covers specifically the major choices made which affect the project as a whole such as the variation of algorithm chosen, technologies used, main techniques used, and additional software used e.g. software used for memory management or testing, and why I have chosen to use this particular software over other similar products.

3.2 Project and Algorithm Requirements

In conducting a study which compares multiple algorithms to each other certain standards and requirements must be met on my end to ensure that the study is fair, accurate, and does not favour a certain outcome. In order to ensure my study meets the high standards required from academic and scientific studies I have identified and planned a set of essential requirements both for my project in general, and for each algorithm in order to ensure the results are credible. Identifying specific requirements should also help me to select the best algorithm variations for large problem instances.

Many of the key requirements of this study are similar to the requirements in other comparative studies, and so I have noted the way in which other studies are conducted to ensure than any key requirements or angles to approach the problem from are not overlooked. The most important requirement for a comparative study of this nature is that all experiments are conducted fairly to ensure that any results are credible and accurate. In order to guarantee that all experiments are conducted in this manner and that the project runs smoothly, I have identified the following requirements:

R1. **Testing must be consistent for all algorithms and performed in a structured manner**. This means that algorithms must be given the same opportunity to find solutions e.g. the same memory and time allocation. Each algorithm must also be run on the same computer so that no advantage is given to any algorithm by running on a faster computer. In this study each algorithm will be given a maximum of 10 minutes for each problem instance.

R2. **All algorithms must be written in the same language to avoid any language-related advantages.** C language has the advantage that it is faster than many other languages. This could give algorithms written in C or similar languages an advantage over algorithms written in slower languages. For this reason, as well as the fact that using multiple languages could require the use of several IDEs, in this experiment it is essential that all algorithms are written in the same language to avoid any language-based advantage.

R3. **Algorithms must be tested on a range of different datasets.** This will provide an accurate depiction of the algorithm's performance over a range of datasets and avoid any anomalies in the data giving any algorithm an advantage.

R4. **Results must be presented clearly and made available in full.** The results drawn from all experiments must be presented clearly in order to ensure each algorithm is represented fairly and results accurately depict the performance of each algorithm.

R5. **Algorithms must be able to produce valid solutions when given an instance of the 0-1 knapsack problem.** Solutions are not

valid if the weight of the knapsack is exceeded, if items are selected more than once or if items are split into fractions. Every solution must adhere to the constraints of the 0-1 knapsack problem.

R6. Algorithms must be able to solve instances of at least 50,000 items. As the aim of this study is to compare the performance of the chosen algorithms in solving large instances of the problem, each algorithm should be designed and implemented with the aim of solving instances of at least 50,000 items.

R7. Algorithms must accurately output the following values: max value, weight of included items, time taken, memory used. In order to compare the performance of each algorithm, each must accurately output information about the solution, and the time and memory used to produce the solution. Ideally, each will record and output these results using the same method in order to try and make the experiment as fair as possible.

As well as satisfying R7, the genetic algorithm must also display the number of generations taken to produce a solution. Recording the number of generations taken allows the genetic algorithm to be compared to those found in different studies and may help reveal trends in the performance.

Each algorithm will be given a maximum running time of 10 minutes. After 10 minutes, if no valid solution has been found the problem will be deemed unsolved. This decision has been made after consideration of the time limits placed on the branch and bound algorithm in similar studies and initial testing of the genetic algorithm in which for medium-sized problems the algorithm was taking up to 2 hours per run. With 2160 individual tests to perform on each algorithm, without a time limit the process of testing and experimentation could take an extensive amount of time. It is for this reason that the upper limit of 10 minutes per run has been set.

3.3 Algorithm Choice and Variations

There are several variations of each algorithm, particularly genetic algorithms and greedy approximation algorithms. Many of these variations are designed to be particularly effective at solving one type of problem, meaning a large part of the

research required in this project is centred around identifying and understanding key variations of each algorithm type in order to select and implement the variation most effective for solving large instances of the 0-1 knapsack problem. Choosing the best variation could be the difference between good and poor overall results, and so the importance of this decision is key to the results and credibility of the study.

The branch and bound algorithm has multiple variations, many of which are studied and used in literature. The main variations are differences in the calculation of bounds using either LP-bounds or Lagrangian bounds, and the method of traversal, either breadth-first or best-first. As mentioned in section 2.5.1 of the literature review, the use of different bounding methods can be seen across several studies and while LP-bounds are used more often, Lagrangian bounds are claimed to find better bounds at the cost of being more expensive to derive. The method of traversal is not mentioned as frequently in academic literature and therefore it is difficult to tell which method may be more suitable for the 0-1 knapsack problem.

In terms of selecting a variation of the branch and bound algorithm, I chose to try and implement an algorithm that determines bounds using Lagrangian relaxation and traverses the state-tree in a best-first approach. As many sources of literature mention that the branch and bound algorithm can only handle instance sizes of up to 750 items, I chose to try and implement Lagrangian bounds in the hope that using tighter bounds would increase the performance of the algorithm and produce better solutions than the more common LP-bounds. Since the aim of this study is particularly focussed on larger datasets, I took the approach that everything should be done to try and pick algorithm variations suited to large datasets and that meet the algorithm requirements.

Genetic algorithms, unlike branch and bound algorithms have a much larger variation of operators and techniques for finding solutions. Over the last few decades genetic algorithms have become popular in solving many types of optimisation problems. Due to the operators that are used in the algorithm they are very easy to modify and tweak in order to optimise the performance, and for this reason nearly all of the genetic algorithms featured in literature will have different parameters and operator types. As I have previous experience working with genetic algorithms, not only was I

very keen on including the approach in my study but I am also aware of the parameter fine tuning that is involved in the implementation of the algorithm.

In keeping with the standard operators included in genetic algorithms I chose to use selection, crossover, mutation and replace operators in order to find solutions. I have chosen to include tournament selection, as well as a novel operator, group selection, described and featured in “Solving the 0-1 knapsack problem with genetic algorithms” (Hristakeva & Shrestha, 2004). The decision to try the group selection method is due to the impressive results found in the study, as well as the in-depth documentation revealing how the operator is implemented. I have also chosen to include 1-point, 2-point and uniform crossover as they are simple to implement, and testing should reveal if any crossover method has an advantage over the others.

I have chosen to implement two different types of mutation operators in order to examine which method is more suited to this problem. As we are optimising a binary string with genetic algorithms, I have chosen an operator which flips the bit of a predefined number of bits in each solution. As for the predefined number of bits to flip, as mutation should only slightly mutate the chromosome, I intend to experiment with low numbers based on the length of the chromosome and only mutate a child at a rate of 0.1% to begin with. The second mutation operator I will include is also featured in “Solving the Knapsack Problem with Genetic Algorithms”, and rather than mutating a percentage of children, mutates every bit in every child chromosome with a 0.1% chance. This means that every child is extremely likely to be modified very slightly and should in theory guide the algorithm towards better solutions quicker as a wider search space is covered. Due to the stochastic nature of genetic algorithms, I have made the decision to include multiple variations of each operator so that I can determine through the process of testing, which variation is most suited to large instances of the 0-1 knapsack problem.

Although I am implementing some of the operators used in ‘Solving the Knapsack Problem with Genetic Algorithms’ in the hope of reproducing the results found in this study, I may need to alter the algorithm to handle larger datasets (Hristakeva & Shrestha, 2005). This is because the algorithm found in this paper is tested on very small datasets with less than 20 items. For this reason, it would be expected that this algorithm may not be as effective on larger datasets, and so in order to satisfy the

requirements it may require significant modification. Initial testing will be performed to determine the right variation most suited to large datasets.

Interestingly, unlike branch and bound and genetic algorithms, I have found no major variations for the dynamic programming approach included in academic sources other than versions of the algorithm which have been hybridised with other algorithms. The standard dynamic programming algorithm, before testing, looks very promising for satisfying all the algorithm requirements.

The greedy algorithm proves a very interesting algorithm in terms of solving the 0-1 knapsack problem, as there are three greedy methods, or variations that can be used, seen in section 2.1.4.4. Due to the nature of the problem, it would be expected that the value-to-weight or weight ascending variations would be most effective at filling the knapsack with valuable items. The method of sorting the items by their value-to-weight ratio can be seen in two comparative studies (Hristakeva & Shrestha, 2005; Shaheen & Sleit, 2016). In order to confirm that this is the most effective method, comparative testing will be performed on all three variations.

3.4 Language and IDE Choices

The diverse range of programming languages available to use today presents the decision of which to use for the development of the algorithms. While some programming languages by design cannot be used, there are many languages that can be used, each with advantages and disadvantages. In this project, I have selected a programming language I think is suited to the problem, and that I am comfortable using. Each algorithm will be written in the chosen language to ensure that no algorithm has language or compiler-based advantages over the others.

For the purpose of creating the fastest possible algorithms to solve optimisation problems, the ideal languages to use are C and C++ as these languages are faster than many others. Java, an object-oriented programming language is also very suited to this problem, and although not known to be as fast as C and C++ has some major advantages. The main advantages with using Java is that it is an object-oriented language and will be useful for maintaining high standards of code readability and simplicity and it is also the language I have the most experience with. Using Java

over C and C++, although in theory not the fastest language to choose, will allow for much easier implementation and will not affect the performance of the algorithms in relation to each other because they all share the same language. For these reasons I have chosen to use Java for this project.

The integrated development environment (IDE) I will be using for developing each algorithm is Eclipse. Eclipse is a Java IDE that I have experience using and has an extensive range of add-ons and plugins that I can source if needed during the project. It also has a range of features that will be useful to me during the testing stages such as custom launch configurations and debugging. As this is the IDE I have the most experience using, I have chosen this platform over others. For the requirements of this project I can see no problems arising from the use of Eclipse.

An Eclipse plugin I intend to use for project management is EGit. EGit is the Git plugin for Eclipse that integrates GIT versioning into Eclipse and allows the management of projects and the ability to push, pull and merge projects. I will use this feature for backing up the algorithms and retaining previous versions in case the programs are lost.

In order to present the results of the study in compliance with requirement 4 (R4), the software I will use to generate graphs and charts is Matlab. Although I have no prior experience with Matlab, I have used the software briefly in preparation for this project while testing different software, and concluded that Matlab is a good choice for generating the graphs and charts, particularly due to the ease of editing elements of the graphs after they have been generated. Research into the kind of graphs that may be used for presenting the results including line graphs and box plots reveal that Matlab has a huge amount of functionality in terms of graph design.

3.5 Conclusion

The key requirements of the project have been identified in this section after careful consideration of exactly what this project aims to achieve and the manner in which it should achieve these. Examination of similar comparative studies has also been conducted to ensure that this project meets the standard of academic and scientific studies and is conducted in a fair and honest manner.

The decisions made in this section regarding the technical aspects of the project, including the coding language, IDE and algorithm choices and testing process have all been made with the ultimate objectives of satisfying the project requirements.

The variation of branch and bound and dynamic programming algorithms have been selected after consideration of the requirements, while the genetic algorithm and greedy approximation algorithm require further testing to determine which variation should be used in the remainder of the project.

4 Methodology

4.1 Introduction

In the previous section of this document the choice of algorithms, software and coding language were selected. This section details the datasets used for the experimentation and testing, including details of the type, size and coefficients of the data. These parameters, and the capacity of the knapsack are grouped under the term ‘global parameters’ as they are set for the entire project and for all algorithms.

In order to determine the performance of the algorithms, certain elements of each algorithm must be examined to determine how well it performs in solving large datasets. As in some academic studies the measures of performance are left unclear, the performance measures in this study are clear and defined and are detailed in this section. This is followed by details of the testing process and recording of results.

4.2 Global Parameters

4.2.1 Number of Items/Data Size

The main purpose of this study is to determine how good the algorithms included are at solving large instances of the 0-1 knapsack problem. In order to change the size of a problem instance, the number of items available to include in the knapsack must change. A very small instance for example, may have 15 items. To make the problem larger, we can increase the number of items available to the algorithm for inclusion in the knapsack.

In this study I would like to determine how good the algorithms are at solving problems larger than those commonly seen in existing academic studies. As most of these studies include problem sizes with up to 10000 items, I have decided to increase the number of items, or data size, to up to 100,000 items.

In order to see how the algorithms scale based on the size of the problem, seven data sizes seen commonly in literature and two much larger sizes are included. The data sizes used in this study are: 50, 100, 500, 750, 1000, 5000, 10000, 50000 and 100,000. Using data sizes featured in similar studies allows for easy comparison of results between this study and others.

4.2.2 Knapsack Capacity

The capacity of the knapsack determines the maximum weight that the included items must not exceed. This means in a problem instance with a maximum capacity of 50, only items with a combined weight of 50 or less can be included in a valid solution. This is likely to cause interesting results when very small capacities and very large datasets meet.

The capacities I will test each algorithm are: 50, 100, 500, 1000, 5000, 10000, 50000, and 100,000. For every knapsack capacity, the algorithms will be tested with every data size, resulting in 72 problem instances for each algorithm to solve.

4.2.3 Weight and Value Coefficients

The data coefficient is the range of numbers that the weight and value of each item can between. Testing on a range of coefficients, as well as a range of capacities and data sizes would result in a huge amount of experimentation and data. For this reason, the coefficient of the data in this study will be set at 100. This means that in any given dataset the weights are in the range of 1-100, and the values are also in the range of 1-100.

4.2.4 Datasets

In order to ensure that the experiment is fair and to satisfy R3, eight different sets of randomly generated data will be used for testing the algorithms on. Each set of data will be used for one of the capacities, e.g. dataset 1 used for all problems with 50 capacity, dataset 2 used for all problems with 100 capacity etc. All four algorithms will be tested with the same set of eight datasets so that the results are comparable.

4.3 Evaluation and Examination

In performing this study I hope to determine and compare the performances of the four included algorithms. The term performance in this context, and even in academic literature appears to be slightly ambiguous, and so to clarify the term performance in this study: it is in part a measurement of the algorithmic efficiency of each algorithm- relating to the speed and memory consumption of an algorithm and in part a measurement of other problem-specific measurements such as the best solutions found, the efficiency and the success rate. In the following sections I will detail the parts of the algorithm I am evaluating, and how I will determine the performance of

the algorithm based on these measurements in order to facilitate a comparison between them.

4.3.1 Speed

Given a situation in which an algorithm must solve an instance of the knapsack problem, one of the most important factors is how fast the algorithm can compute a solution to the problem. In order to measure the time taken for each algorithm to solve the problem I will use the measurement of wall clock time. This is simply the standard measurement of time for the algorithm to complete solving the problem and for this study does not include the time used for initialising the data or writing the results to file.

The alternative time measurement frequently used in literature is CPU time which measures the total time during which the processor is actively working on the task. However, in this study the wall clock time is measured as this is representative of the time the algorithm takes to solve the problem from the user's point of view. For a user in business or a developer, this measurement of time is the most important, as this is the real time they will spend waiting for a solution to be calculated.

Examinable output: Time in milliseconds of the execution time of the algorithm only.

4.3.2 Success Rate

The success rate denotes how often the algorithms are able to find the optimum solution to the problem. This can be determined by comparing the value found to the optimum found by dynamic programming.

Examinable output: Percentage of runs where the optimum value is found.

4.3.3 Effectiveness

The effectiveness of an algorithm is a measure of the average solution quality obtained during the set of experiments. This can be found by determining the mean average value found over the series of experiments. This measurement allows credit to be given to algorithms which have a low success rate if they are very close to the optimum. For example, if an algorithm has a low success rate and a high effectiveness, then the algorithm consistently gets good results but rarely reaches the optimum value. Conversely, if an algorithm has a high success rate and a low

effectiveness, then it is able to find the optimum regularly, but also returns some very poor results.

Evaluating how good the results are- specifically how high the best value is- reveals how effective the algorithm is at finding good solutions to the problem. As R5 states that valid solutions must be produced, and R6 states that each algorithm must be able to solve instances of at least 50,000 items, this aspect of performance is a hugely important factor in the overall performance of the algorithms.

The dynamic programming algorithm, by design, can solve the 0-1 knapsack problem optimally, and is the only algorithm able to guarantee optimality. The advantage of this is that the best value found by the dynamic programming algorithm in this study can be used as a benchmark for the other algorithms to determine if they are optimal, or how close to optimal the results they produce are.

Examinable output: Average solution quality determined by the mean value of 30 runs

4.3.4 Memory Usage

How much memory the algorithm uses in total when solving an instance is important particularly for larger datasets. As the size of the data increases, the memory required by each algorithm would also be expected to increase. This measurement is particularly important because if an algorithm requires a substantial amount of memory to run it may not be suitable for some computers.

This measurement will also be useful for determining how the memory usage scales with the size of the problem, and what factors of a problem affect the memory usage of each algorithm.

Examinable output: Memory in kB, MB or GB used by the program.

4.3.5 Large Datasets

As the main aim of this study is to compare the performance of each algorithm when tested on large datasets or up to 100,000 items, and R6 states that algorithms must be able to solve large problem instances, the ability of the algorithm to solve large datasets, and the scalability of the algorithm is an essential factor in the overall performance measure.

Examinable output: Judgement of how the algorithm scales based on previous measurements.

4.4 Testing Methodology

The testing stage of this project is the most significant process in this project. This is the stage in which each algorithm must be run thousands of times in order to gather the results needed to determine the performances of each algorithm and is likely to take a considerable length of time. As stated in R1:

"Testing must be consistent for all algorithms, and performed in a structured manner"

In this section I will detail the manner in which the testing stage will be conducted in compliance with this requirement, and the thought process behind each decision.

4.4.1 Expectations

Through initial research I have determined that the branch and bound, dynamic programming and greedy approximation algorithms are deterministic, and will return the same solution every run. This means at least the maximum value, weight, and items chosen will remain the same every time the algorithms are run on a single instance of the problem. The genetic algorithm however is stochastic and will produce different results every time it is run on a problem. This means that from run to run, the maximum value, weight and items chosen are likely to change for genetic algorithms and remain the same for the other algorithms.

While the maximum value and weight of the included items will not change for at least three of the algorithms, the measurements of time and memory used are extremely likely to change for every individual run and will have to be averaged to determine the average usage/time for each problem. For this reason, I will run each algorithm 30 times for each problem instance in order to draw credible and accurate results from the algorithms. As I am testing the algorithms on 9 data sizes per capacity, and there are 8 capacities, each algorithm will run a total of 2160 times (9 * 8 * 30). In order to facilitate this many runs I will use the launch configuration feature of Eclipse which allows the consecutive running of programs. Using this feature, each algorithm will complete all 2160 runs in one continuous sitting. Using this structure satisfies R1.

4.4.2 Recording of Results

Recording the results of each run will involve first the output of raw data from each algorithm. Each time the algorithm completes a run, the maximised value, weight of included items, wall clock time in milliseconds and the memory used in kilobytes will be appended to a text file. An example of one line of the file, representing one run of the program can be seen as:

6031,500,12.93991,6144,

As each algorithm is run 30 times for each problem, every 30 lines of the text file represents the results for one problem. In order to calculate the mean average for each problem, a separate program is used which reads 30 lines of the text file at a time and produces averages for each value. These averages are then appended to a separate file and copied into an excel spreadsheet containing the mean results for each problem for all four algorithms.

As the number of generations for the genetic algorithm is being recorded, the genetic algorithm will output five numbers. A slight modification of the program which averages the results allows for the number of generations to also be averaged and appended to file. Outputting these values to file satisfies R7.

The spreadsheet used for storing the results contains 8 tables per algorithm, each representing all problem sizes for the relevant capacity. Table 1 represents an example of the first two tables for the dynamic programming algorithm, with 9 problem sizes for each capacity shown, and an example of an averaged result for two problems.

I have chosen to conduct my testing in the manner above for the main purpose of meeting the requirements of the project. The testing process stated above is consistent for each algorithm, and has a clear structure, meeting R1. Each algorithm will be tested on a range of different datasets, meeting R2, and each algorithm will output the required outputs of maximum value, weight, time and memory, meeting R7. For reference, the full results table can be seen in appendix 4.

		Dynamic Programming			
Number of Items	Capacity	Profit of Included Items	Weight of Included Items	Wall Clock Time (ms)	Memory (kB)
50	50	316	49	2.5	2048
100	50	467	47	4.5	3072
500	50				
750	50				
1000	50				
5000	50				
10000	50				
50000	50				
100000	50				
50	100				
100	100				
500	100				
750	100				
1000	100				
5000	100				
10000	100				
50000	100				
100000	100				

Table 1. Example of Results Table

5 Algorithm Implementation and Variation Testing

5.1 Generating and Processing the Data

As mentioned in section 2.1.2, D. Pisinger's problem generator can be used to generate different types of random datasets and can be found online alongside example testcases and algorithms[1]. For testing purposes, eight different sets of data are generated using this test generator, and each is solved by all four algorithms. Each dataset is made up of 100,000 uncorrelated weights and values, and the data has a coefficient of 100 meaning that the weights and values are in the range 1-100.

In order to read in the data to each program I implemented a generic data processor which can be reused for each algorithm. The data processor can read in the data from an input file (.in extension) to a 'weights' and a 'values' array of a predefined size. This means that when testing on a data size of 50 (50 items), the data processor will read in the first 50 values, and for a data size of 100,000, the processor will read in the first 100,000 values.

The data processor calls the java scanner class to read the file and uses a loop to read in tab separated values to the corresponding arrays. The input file has the following structure:

1	28	14
2	53	45
3	45	18
4	68	71
…		

Figure 1. Structure of the Test .in File

where the first value is the number of the item, the second is the value of the item and the third is the weight. The loop skips the first number, adds the second number to the 'values' array and adds the third number to the 'weights' array every iteration.

[1] http://hjemmesider.diku.dk/~pisinger/codes.html

5.2 Branch and Bound Algorithm

The branch and bound algorithm featured in this project is a modified version of an algorithm first implemented and published by Shalin Shah in his knapsack project[2]. The algorithm in the project is a branch and bound algorithm which uses a greedy approximation algorithm as an incumbent. This allows an initial solution to be presented by greedy approximation, and then the branch and bound algorithm to further solve the problem using this solution as a starting point. This often improves on the value found, and in many cases finds the optimum value. In order to use just the branch and bound algorithm I modified the program to remove the greedy approximation algorithm.

This algorithm has been selected after facing difficulties in implementing the branch and bound algorithm. After extensive research and effort, I was unable to implement a working branch and bound algorithm with the ability to solve problems accurately. In order to continue studying this algorithm's performance, I have chosen to use the above algorithm, which not only traverses the state space tree in a best-first approach, but also calculates bounds using Lagrangian relaxation. These are two features I was very keen on using after conducting research, in the hope that they would help me satisfy the project and algorithm requirements. As most of the branch and bound algorithms in literature use LP-bounds and breadth-first traversal, I chose to use this version as theoretically it should be better suited to larger problems and may improve on the results seen in existing studies.

In order to allow this branch and bound algorithm to work with the generated datasets, I removed the pre-existing example dataset and implemented the generic initialisation class detailed in section 5.1, allowing the datasets to be read into the program.

Initial testing suggests that the branch and bound algorithm is unable to handle instances of the problem with more than 750 items, and although completely in line with the majority of results in existing literature, this result is disappointing. In order to try and fix this problem I increased the available memory for Eclipse to 12Gb. This however, had no impact on the ability of the algorithm to solve large datasets. In

[2] https://github.com/shah314/knapsack

order to definitively confirm that this did not allow the algorithm to solve larger problems, I left the algorithm to solve for several hours and no solution was found. The extent of the poor performance will be determined in the full testing of the algorithm, however, can likely be attributed to the linear space complexity of the algorithm, meaning that the amount of space required by the algorithm increases with the amount of input values.

A lack of detailed descriptions of the implementation of the algorithm in existing literature and online sources means that implementing an algorithm suited to large datasets is extremely difficult. I discovered the above algorithm, implemented by Shalin Shah, when conducting research into the use of genetic algorithms (Shah, 2019).

5.3 Dynamic Programming Algorithm

The dynamic programming algorithm is based on a standard bottom-up, iterative dynamic programming approach with no additional features or abilities as this algorithm is already known to solve the problem optimally. Both pseudocode and code are available across academic and non-academic sources for this version of the dynamic programming method with only slight variations in the layout of the loops, strongly suggesting that for the 0-1 knapsack problem this version is the accepted method for solving the problem.

The method for populating the table in the algorithm in this study is a slightly different approach to many of the algorithms featured in other sources in the fact that the algorithm populates the table column-by-column, rather than row-by-row. This small difference, which is caused simply by the inner and outer loops of the algorithm being switched only has the effect that the included items must be worked out by traversing the table in an inverse manner.

The pseudocode for the algorithm is detailed in Figure 2. As seen in the pseudocode, the first step is to create the table to store the results of the subproblems. The dimensions of the table are always [dataSize+1][capacity+1]. The reason for adding 1 to each dimension is to account for having 0 items, and to account for a knapsack capacity of 0. It is for this reason that the second step is to fill both the whole row and column 0 with the value 0- because with 0 items the maximum value is 0, and with a

capacity of 0, we cannot fit any items into the knapsack so the maximum value is also 0.

After creating the table and populating the first row and column, the table is ready to be filled with solutions to the subproblems. In an iterative manner using an inner and outer loop, the algorithm calculates the subproblems. The inner loop works out a column of the table each iteration of the outer loop, resulting in the table being populated one column at a time, left to right. The value found in the last row of the last column is the optimum value of the problem and is returned once the table has been filled.

The process of returning the items included in the knapsack involves finding a route through the table from the optimum value back to the root of the table. In order to find the items, the rule ‘if the current value is equal to the value to the left, the item is not included’ is applied. If this rule is not satisfied on the basis that the current value is not equal to the value to the left, then the item is included in the knapsack and the current column is printed (number of item). The variable *m* is then set to the previous weight in the dataset and *n* is reduced by 1 to move to the next column. If the item is not included in the knapsack we retain the value of *m* and just move to the next column.

```
1   CREATE int table[dataSize+1][capacity+1]
2
3   SET first row and column of table to 0
4
5   // Fill the rest of the table (Outer loop- Items, Inner loop- Capacity)
6   FOR (i = 1 to i == dataSize)
7           FOR (j = 0 to j == capacity)
8                   IF (j > weight[i] AND table[j][i] < (j - weight[i-1]) + values[i-1])
9                           SET table[i][j] to table[i-1][j-weight[i-1] + values[i-1]
10
11  RETURN table[dataSize][capacity]   // maximum value
12
13  // Find the included items in bottom up approach
14  n = dataSize
15  m = capacity
16  Loop WHILE n != 0
17          IF table[n][m] NOT table[n-1][m]
18                  PRINT n
19                  m = weight[n-1]
20          n - -
```

Figure 2. Dynamic Programming Pseudocode

5.4 Genetic Algorithm

5.4.1 Variation Selection and Parameter Tuning

Genetic algorithms are stochastic, meaning they can produce different results each time they run. As the genetic algorithm in this study must adhere to requirements R5-R7, the parameters and operators used must be modified to suit large knapsack problems. The process of designing a genetic algorithm suited to these problems took a considerable amount of time, and from three variations of the algorithm I was able to select the algorithm with the best performance.

The initial genetic algorithm implemented is based on the algorithm detailed extensively in ‘Solving the 0-1 Knapsack Problem with Genetic Algorithms’ (Hristakeva & Shrestha, 2004). I chose to implement this algorithm as it is extremely well documented and has good performance in the above study. However, I was cautious as the datasets used in the study are extremely small compared to the datasets used in this study. For this reason, I modified the operators and tuned the parameters to see if the performance can be improved for larger datasets.

5.4.1.1 Genetic Algorithm Variation 1 (GA 1)

This is the initial algorithm I implemented for the problem and is based on the algorithm featured in in ‘Solving the 0-1 Knapsack Problem with Genetic Algorithms’ (Hristakeva & Shrestha, 2004). The fitness function in this algorithm is a repair function, which flips the ‘included’ Boolean bit of included items in invalid solutions until the solution is under the maximum capacity. The initialisation of the population is random, with items set to ‘included’ at a rate of 5% for all problems. The other main feature of this algorithm is that up to 85% of the population is replaced each generation by offspring of the selected parents. In order to retain the best solutions in the population, this algorithm uses 2-Elitism, which copies the best two solutions in each generation to the next, ensuring that if these are picked as parents, the best solutions are not lost.

An immediate problem I have identified with this algorithm is the amount of processing power required to replace up to 85% of the population every generation. This method is likely to affect how fast the algorithm runs, especially for larger

problems in which each chromosome in the population will have thousands of items. The operators and parameters used in this algorithm can be seen in table 2.

Operator/Feature	Type
Initialisation	Random- 5% chance of being included
Elitism	2-Elite
Selection	Group Selection
Crossover	1-Point Crossover
Mutation	Flip-Bit
Replacement	Replace Worst (only if better fitness)

Parameter	Value
Population Size	350
Crossover Number (Replace up to this percent of population)	85%
Mutation Rate	0.1%
Stopping Function	No improvement in 2000 generations or time over 10 Minutes

Table 2. Operator Types and Parameter Settings for GA 1

Initial testing of this algorithm suggests that the performance is good for small problems with 50-500 items, in which the optimum value was found several times. However, on medium and large problem instances with 10000+ items, the performance looks very poor. Testing this algorithm on an instance with 10000 items and a maximum capacity of 100, the algorithm was unable to find any solution with a maximum value over 812, and of the 15 test runs performed on this instance, all 15 ran for the full 10 minutes. Through dynamic programming I was able to find the optimum value to be 6105. These results suggest that some aspect or operator of the algorithm is performing poorly in larger problems.

5.4.1.2 Genetic Algorithm Variation 2 (GA 2)

This algorithm contains some major changes from GA 1. Through trial-and-error I was able to identify the main cause of the poor performance to be the replacement strategy, in which up to 85% of the population is replaced by the offspring of two

chromosomes, and the mutation rate of 0.1%. This mutation rate appeared to be a good rate for small-medium sized problems with up to 5000 items, however larger problems appear to benefit from a smaller mutation rate.

The first major change is the replacement strategy. Rather than replacing most of the population each generation, GA 2 replaces the worst chromosome in the population if the new child chromosome's fitness is higher than the fitness of its main parent. This technique works better for all problem instances than the replacement strategy in GA 1.

The second major change to this algorithm is the varying mutation rates. In performing tests with smaller mutation rates of 0.01% and 0.001%, I found that these mutation rates significantly improve the performance of the algorithm of larger datasets. Through further testing I was able to identify the best mutation rates for each problem size and set the rate for each problem size programmatically. The reason that lower mutation rates work better for larger datasets is that each large chromosome that is mutated has a smaller chance of becoming an invalid solution if less of the items are flipped to included.

Further changes include reducing the elitism to one chromosome per generation. Although in theory when only replacing a maximum of two chromosomes per generation through replacement, elitism should cause the population to converge much faster, it does in fact help the population keep improving at a fast rate. This is likely due to the best chromosome being duplicated each generation, and therefore increasing the selection pressure as strong solutions are duplicated.

Initial testing of this algorithm shows that the value found is substantially closer to the optimum value than GA 1. However, for large instances of the problem with data sizes of 50000 and 100,000 items, the value produced is still very small compared to the known optimum. As the aim of this project is focussed on large problems, further improvement is required to tune the algorithm to larger problems with high capacities and data sizes.

Operator/Feature	Type
Initialisation	Random- 5% chance of being included
Elitism	1-Elite
Selection	Group Selection
Crossover	1-Point Crossover
Mutation	Flip-Bit
Replacement	Replace Worst (if fitness better than parents)

Parameter	Value
Population Size	350
Crossover Number *Replace up to this percent of population	N/A
Mutation Rate	See Table Below
Stopping Function	No improvement in 2000 generations or time over 10 Minutes

No. Items	Mut. Prob.
50	0.1%
100	0.1%
500	0.1%
750	0.1%
1000	0.1%
5000	0.01%
10,000	0.01%
50,000	0.001%
100,000	0.001%

Table 3. Operator Types and Parameter Settings for GA 2

5.4.1.3 Genetic Algorithm Variation 3 (GA 3)

The third variation of the genetic algorithm, GA 3, is the version used and referenced in the rest of the project. After considerable effort in tuning the parameters and experimenting with different operators, the final change to the algorithm involves partially hybridising the algorithm with the greedy approximation algorithm. This idea is mentioned previously in section 2.5.4.1 of the literature review.

This algorithm uses a greedy strategy to initialise the population. When initialising the data, instead of including items in the initial population with a 5% probability, items

are only included if they have a value-to-weight ratio of over 2. This value may need modified for datasets with coefficients different to the coefficient of the data in this study.

Operator/Feature	**Type**
Initialisation	Greedy, all items with value-weight ratio > 2
Elitism	1-Elite
Selection	Group Selection
Crossover	1-Point Crossover
Mutation	Flip-Bit
Replacement	Replace Worst (if fitness better than parents)

Parameter	**Value**
Population Size	350
Crossover Number *Replace up to this percent of population	N/A
Mutation Rate	See Table Below
Stopping Function	No improvement in 2000 generations or time over 10 Minutes

No. Items	Mut. Prob.
50	0.1%
100	0.1%
500	0.1%
750	0.1%
1000	0.1%
5000	0.01%
10,000	0.01%
50,000	0.001%
100,000	0.001%

Table 4. Operator Types and Parameter Settings for GA 3

Using the greedy initialisation method allows the algorithm to start optimising from a much higher value than with a random initialisation. Initial experimentation suggests that this technique is superior for large datasets with 50,000 and 100,000 items, and consistently produces good results for small problems. Further experimentation is

required, and detailed in the next section, to determine the effect changing capacities has on the algorithm.

5.4.1.4 Genetic Algorithm Variation Experimentation

Experiment 1: Small Capacity, Medium Data Size

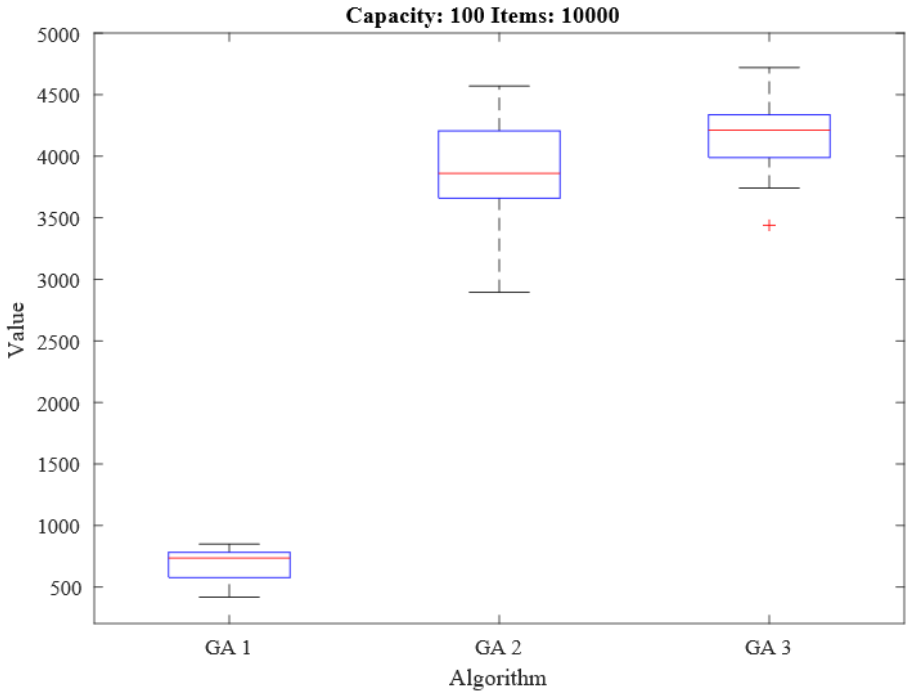


Figure 3. Experiment 1, Optimum Value: 6105

The first experiment conducted is on a problem with a small capacity of 100, and 10000 items. Problems of this nature where the capacity is smaller than the number of items can be very difficult in the sense that the capacity is very limited, so the items included must only be the best of the population with high value-to-weight ratios. This problem has the same number of items as the maximum number found in most literature, and therefore algorithms should perform well in this experiment if they are to solve much larger problems in the rest of the project.

The results of this experiment, seen in Figure 3, immediately show that algorithm GA 1 has a very poor performance compared to GA 2 and GA 3. The range of maximised values is in a very poor range both compared to the other algorithms and to the optimal value, and the time required to produce these results was the full 10-minute limit for all 30 test runs.

GA 2, with an altered replacement strategy and mutation rate is able to significantly improve on the range of values found. The values found are roughly within the range of 50-75% of the optimum. As the only differences between GA 2 and GA 1 are the replacement strategy and mutation rate, it can determined that the poor performance of GA 1 is primarily affected by the replacement strategy of replacing large amounts of the population with children derived from the same parents. Changing this method to replacing a maximum of two chromosomes per generation has the effect of speeding up the algorithm, allowing for more generations within the same timeframe, and significantly increases the maximum value found.

GA 3, which has the advantage of a greedy initialisation is able to further improve on the range of maximised values found over 30 runs and has a median noticeably higher than that of GA 2. The range of values is in also in a higher range and smaller solution space, meaning that the algorithm is more accurate in finding good solutions often. This is further supported by calculating the standard deviation of both algorithms, which results in a value of 410.25 for GA 2, and 280.30 for GA 3.

In order to confirm that the difference between the values found by GA 3 and GA 2 are significant enough to prove that there is a performance increase, I performed a t-test on the results. This produced a p-value of 0.001062, therefore rejecting the null hypothesis and confirming that there is significant difference between the results of GA 3 and GA 2.

Experiment 2: Small Capacity, Large Data Size

This experiment is conducted on a problem with a larger capacity and number of items. The performance of GA 1 in this experiment is extremely poor, with a median of around 2000- less than 5% of the optimum value. For this reason, this variation of genetic algorithm is not suitable for large instances of the knapsack problem.

GA 2 is able to significantly improve on the performance of GA 1, further proving that the replacement and mutation methods in GA 1 are ineffective for large problems. For this problem however, all 30 runs of GA 2 ran for the full 10 minutes, suggesting that if given longer it may produce better results. From this result the aim is to

increase the speed at which the population evolves in order to reach better solutions faster.

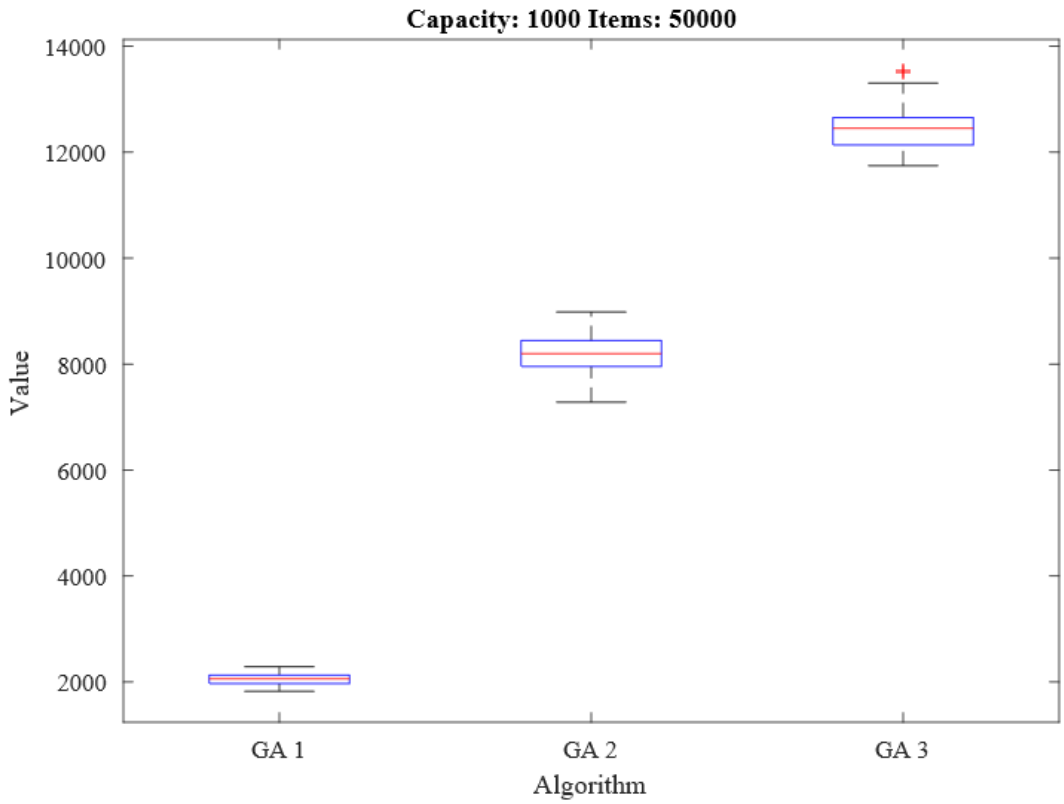


Figure 4. Experiment 2, Optimum Value: 45973

As seen in Figure 4, initialising the population from a better starting point increases the maximised value found, shown by GA 3. This algorithm also runs for the maximum allotted time of 10 minutes for all 30 runs. As the values found are so low compared to the optimum value, even for GA 3, this type of problem may highlight a weak point for the genetic algorithm.

Experiment 3: Medium Capacity, Medium Data Size

This experiment is conducted on a problem with a higher capacity than the number of items. In theory, this should make the problem easier as a higher percent of the items are going to be included in the final solution. Using the dynamic programming approach, the optimum value has been determined as 57029.

GA 2 in this experiment has a fairly good performance, consistently producing results within 14% of the optimum value. GA 3 however, produces superior results within 1% of the optimum. As the only difference between these algorithms is the greedy

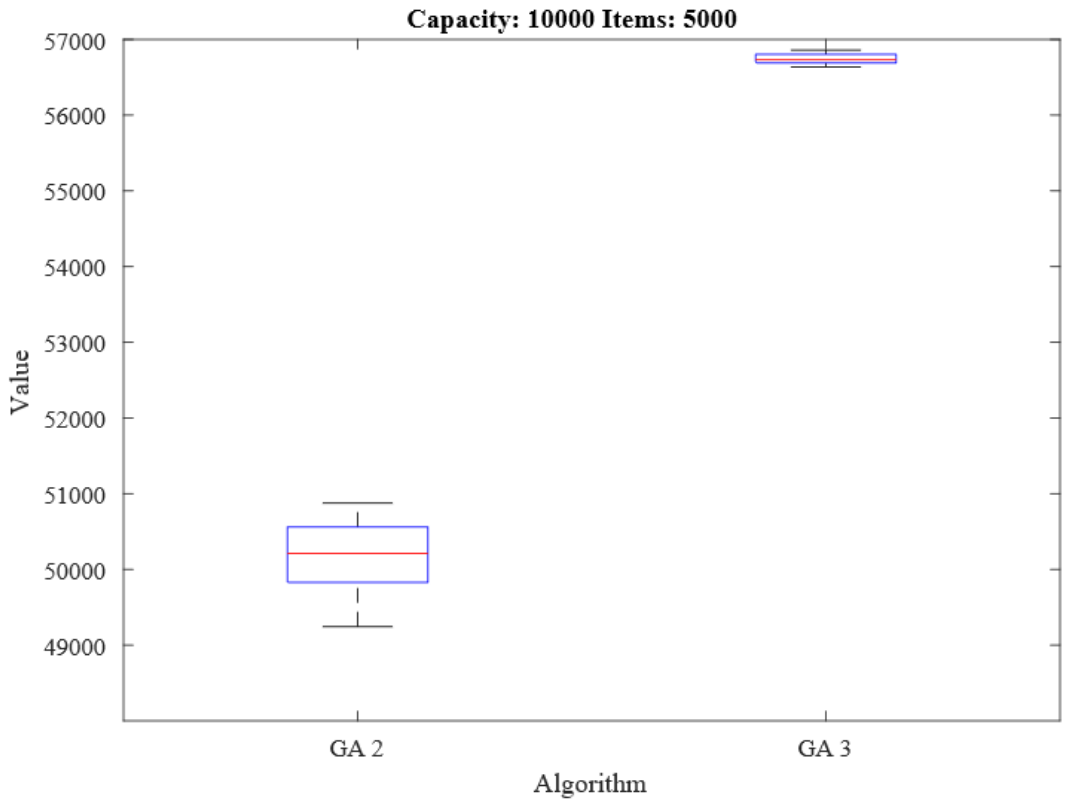


Figure 5. Experiment 3, Optimum Value: 57029

initialisation, it can be concluded that the greedy initialisation makes the algorithm more suitable for large problems.

An interesting observation about this experiment is the time and memory usage of each algorithm. For all 30 runs GA 2 took 10 minutes. GA 3, however, takes an average of 31 seconds before the stopping function is called. This further supports the usage of greedy initialisation in the genetic algorithm for this study.

5.4.2 **Implementation of GA 3**

In order to represent a solution to the problem (the list of items either included or not included in the knapsack), an object of type ‘Individual’ is used which contains three arrays. The first two arrays are of type int and hold the weights and values of each item. An additional Boolean array is also created- ‘included’, which represents whether an item is included in the knapsack or not. The collection of these three arrays, known as an individual or chromosome, represents a single solution to the problem, and looks like the following, where item 3 and item 7 are included in the knapsack:

Values	14	46	49	35	48	68	12	9	74	…
Weights	2	90	10	44	75	32	2	33	7	…
Included	0	0	1	0	0	0	1	0	0	…

Figure 6. Representation of a Solution

The fitness value of each chromosome is representative of the sum of the values of all the included items. In order to calculate the maximum value of an individual and make sure it is within the capacity of the knapsack, a fitness function is implemented.

This algorithm utilises a repair fitness function which modifies invalid solutions so that every chromosome in the population cannot be invalid. This works by calculating the sum of all the weights and values of the included items in an individual. Once calculated, if the sum of the weights exceeds the capacity of the knapsack, the function randomly selects included items and flips the Boolean value in the included array to false, excluding the item from the knapsack. This process is repeated until enough items have been excluded to bring the sum of weights below the capacity. This process ensures that all chromosomes are valid solutions (do not exceed the capacity of the knapsack) at all times and can be seen in Figure 7.

The initialisation of the data, involving reading in the data and populating the arrays determines the initial population of chromosomes that the algorithm begins with. While this may not appear the most important aspect of the algorithm, seeding the algorithm with a good initial population can have a monumental effect on the results the algorithm returns.

The approach to seeding this population involves including every item with a value-to-weight ratio of 2 or above in each chromosome, as discussed in section 5.4.1.3. This technique was found to make the algorithm significantly more effective for large datasets than randomly including items in the initial chromosomes.

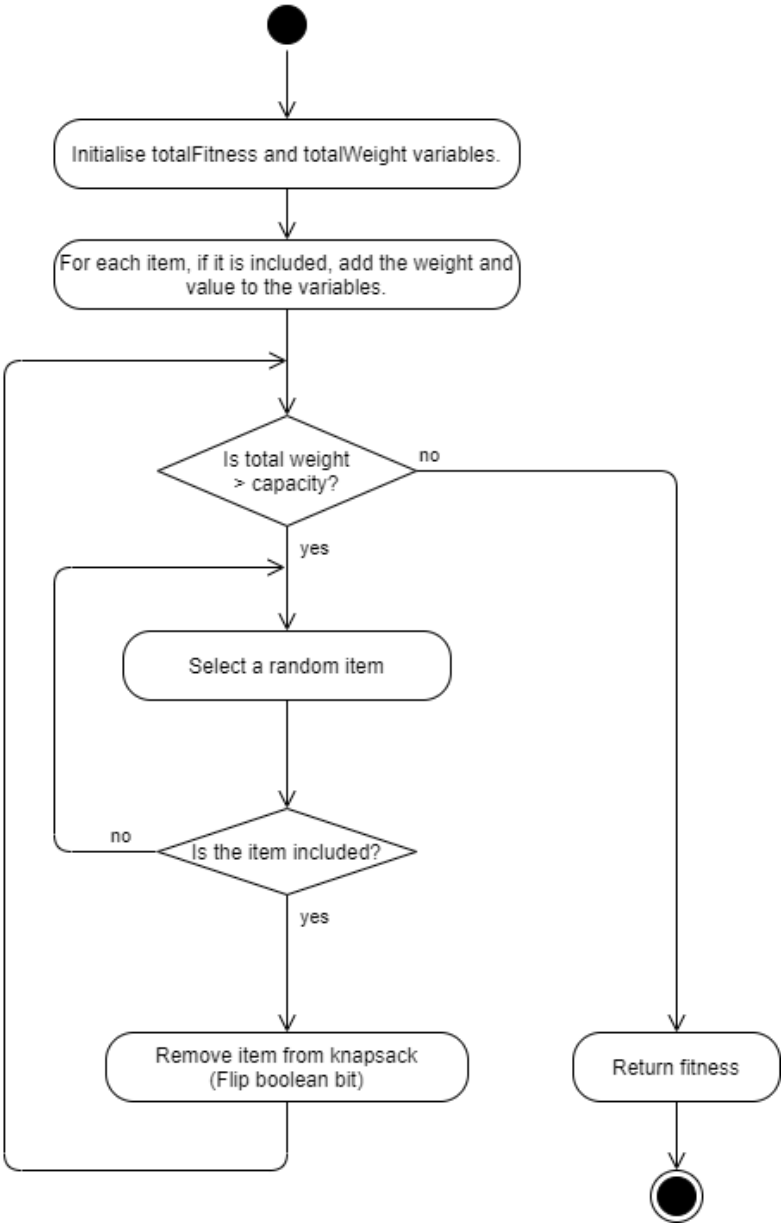


Figure 7. Genetic Algorithm Fitness Function

Using the greedy initialisation, every item in the knapsack that has a value more than or equal to twice the value of the weight is included in the knapsack. If the weight of the items included in the initialisation is more than the knapsack capacity, then items will be removed at random until the weights of the chromosomes do not exceed the capacity.

The result of this change is that for almost all problems the initial population will have a higher average fitness and will start optimising from a higher initial value. Not only can this save thousands of generations, but also a considerable amount of time.

The selection operator included in this algorithm is based on a novel operator featured in ‘Solving the 0-1 Knapsack Problem with Genetic Algorithms’ (Hristakeva & Shrestha, 2004). The group selection function was shown in the study to outperform roulette wheel selection and through experimentation I found it also exceeds the performance of tournament selection. The group selection operator works by sorting the chromosomes in the population by their fitness in a descending order, so that the chromosomes with the highest fitness are at the beginning of the collection. The sorted collection is split into groups, each representing 25% of the chromosomes. Using a generated random number, a chromosome is randomly selected from the first group with a 50% probability, from the second group with a 30% probability, from the third group with a 15% probability and from the final group with a 5% probability. This ensures that fitter chromosomes are selected as parents more often than weaker chromosomes. In Figure 8 the population is made up of 20 individuals. Each number represents the index of the individual in the population, so the fittest individual in this example can be found at index 5 of the population.

Figure 8. Group Selection Operator

Implementing this part of the selection function proved more challenging than anticipated. As the population is stored in an arrayList, which cannot be ordered, I had to implement a pairs class to store the fitness of the individual and its index in the population. This allowed me to order the list of pairs by the fitness and retain the original ordering of the population. After selecting a random pair using this selection method, the parent chosen is found at the selected index in the population. The group selection method is repeated twice in order to select two parents from the population.

The purpose of the crossover operator is to merge the ‘included’ value of both parents together to create a new child chromosome. I implemented three crossover operators, 1-point, 2-point and uniform, and found 1-point crossover to be the most

effective for merging the parents. This is achieved by randomly selecting a crossover point in the chromosome. Everything on the left side of the point is copied from parent 1, and everything right of the point is copied from parent 2. Crossover is applied at a rate of 85%, meaning that both parents will produce a child chromosome 85% of the time. The remaining 15% of the time, parent 1 is copied in full to the child chromosome. This process is repeated twice, with the parents inverted the second time, to form two child individuals.

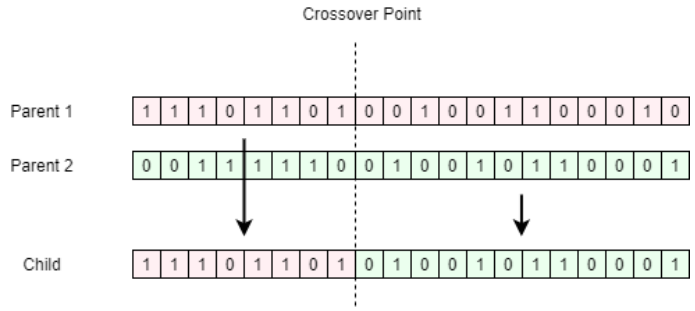


Figure 9. 1-Point Crossover Operator

Mutation is essential for maintaining diversity in the population and helps to avoid convergence. Once both child chromosomes have been created, mutation is applied to each bit of each child at a very low rate. Each bit representing whether an item is included in the knapsack or not is flipped from true to false or false to true at a predefined rate of 0.1% for small problems with 50-1000 items, 0.01% for medium instances with 5000-10000 items and 0.001% for large instances of 50000-100,000 items.

Once the parents have been selected, the children have been created and mutation has been applied, the new children replace their parents in the population if they have a higher fitness. As discussed in section 5.4.1.2, this method was found to be better than replacing large amounts of the population each generation.

An additional feature included is elitism, which is the process of copying a number of the fittest individuals, unchanged, into the next generation. This has been shown to significantly speed up the performance of the genetic algorithm, and also helps to prevent the loss of good solutions once they have been found (Deb, Agrawal, Pratap,

& Meyarivan, 2000). In this genetic algorithm, at the start of each generation the fittest individual is copied, which then replaces the worst individual in the population at the end of each generation, ensuring that if it was selected as a parent the strong genetic material is not lost.

In order for the algorithm to determine when to stop, it records the best value found every 2000 generations, and checks if the best value has improved or not. If the population is not progressing, then the algorithm is stopped. This technique allows the population to be converged for long periods of time, up to 2000 generations, and while the population may converge, crossover and mutation can allow the algorithm to escape local optima and progress further towards better solutions.

5.5 Greedy Approximation Algorithm

5.5.1 Variation Selection

As mentioned in section 2.5.4 of the literature review, there are three main strategies a greedy algorithm can use to calculate solutions. In order to select the best strategy for satisfying the requirements of this study, several tests have been conducted measuring the maximised value. In a second round of experiments the memory and time usage are recorded to determine the difference in performance between an algorithm found in literature and the most effective variation below.

5.5.1.1 Weight Ascending Strategy (WE)

This algorithm uses the weight ascending strategy to fill the knapsack. Using this method, the list of items will be sorted in ascending weight order, and items will be added one at a time from the lightest until the weight of the items in the knapsack is equal to the capacity, there are no more items to add, or adding the current item exceeds the capacity. In this case, the next item that is small enough to add to the knapsack without exceeding the capacity is added.

5.5.1.2 Value Ascending Strategy (VA)

This algorithm uses the value ascending strategy to fill the knapsack. Using this method, the items are sorted in ascending or descending value order. The items are added in the same way as above, with the highest value items added first.

5.5.1.3 Value-to-Weight Strategy (V2W)

This algorithm uses the value-to-weight strategy to fill the knapsack. Using this method, the items are sorted by the ratio between the value and the weight. The higher the ratio, the higher the value is compared to the weight and the more valuable the item is for finding good solutions. Items are added from the highest ratio to the lowest.

5.5.1.4 Greedy Algorithm Variation Experimentation

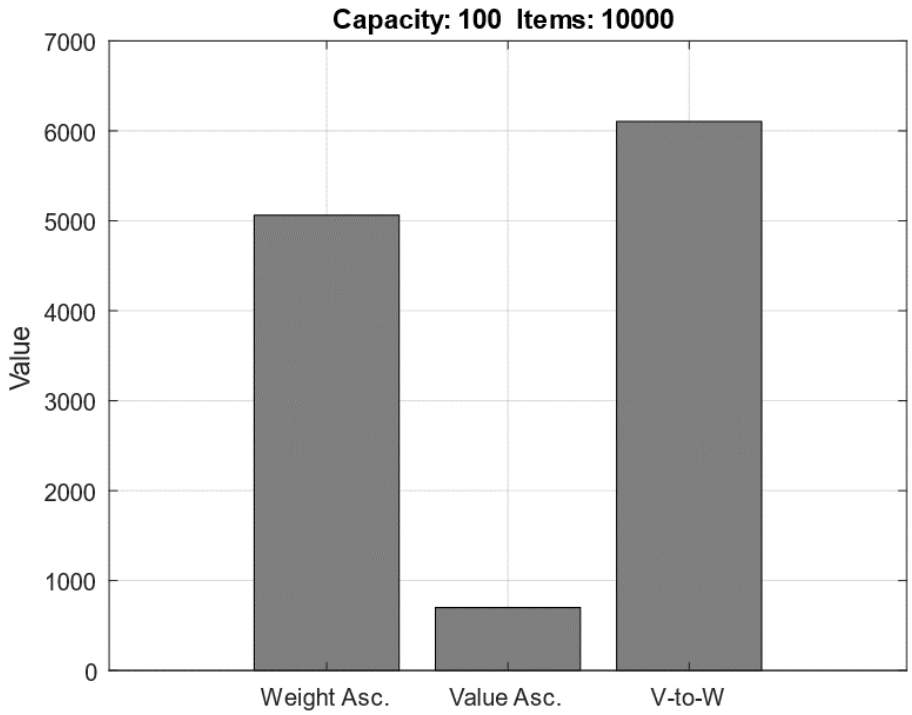


Figure 10. Experiment 1, Optimum Value: 6105

As seen in Figure 10 and Figure 11, the value ascending algorithm is very poor at finding good solutions in problems where the capacity is much lower than the number of items. This is because problems of this nature increase the need to select only the best items in the dataset with high values and low weights. The weight ascending algorithm performs much better and is able to present good solutions with high values. However, as expected, the value-to-weight variation is able to perform even better for both of these problems as it selects only the best items in the dataset with the highest value-to-weight ratios.

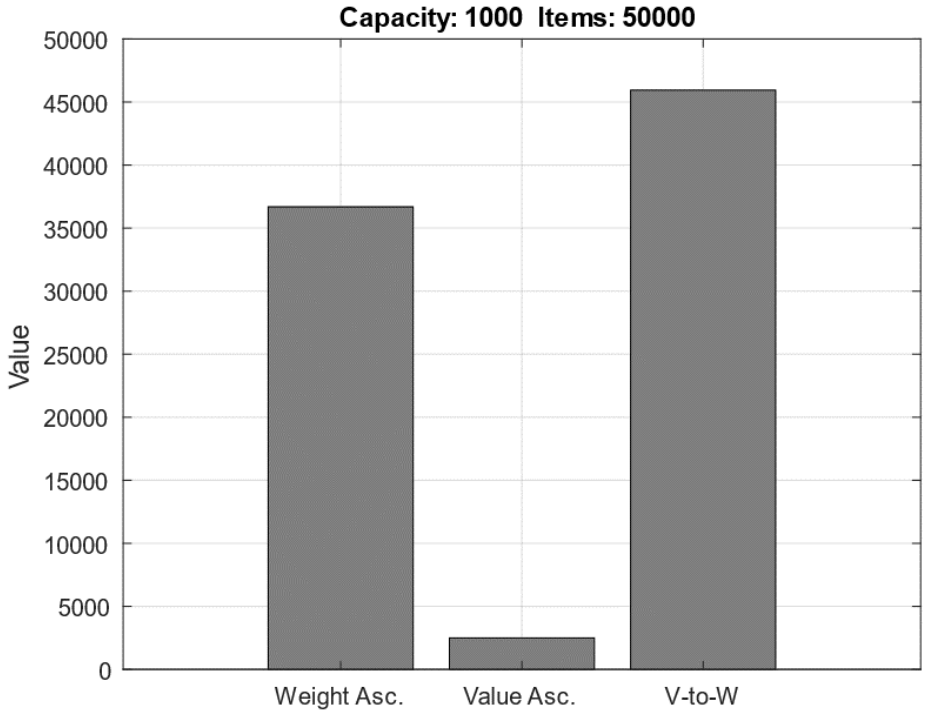


Figure 11. Experiment 2, Optimum Value: 45973

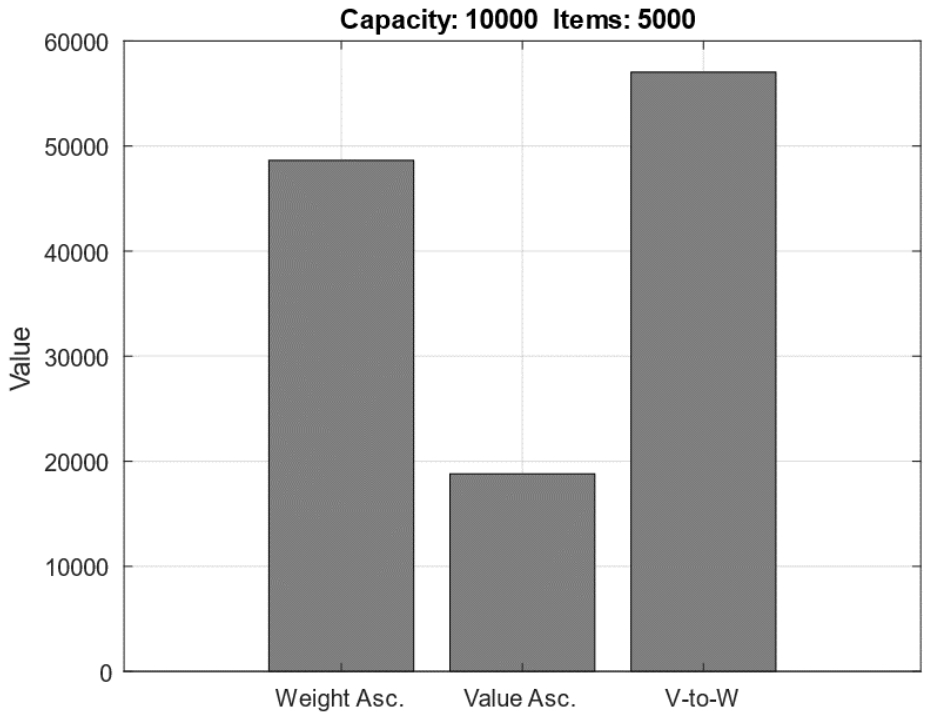


Figure 12. Experiment 3, Optimum Value: 57029

Experiment 3, as seen in Figure 12 has a higher capacity than the number of items, lowering the need to select only the best items and allowing the value ascending algorithm to get slightly better results. However, the value-to-weight variation is still the best algorithm in terms of maximising the value for a range of problem sizes in this study, including large instances.

Taking forward the value-to-weight variation of the algorithm, a second set of experiments is conducted in order to confirm that this algorithm is effective for large problem instances. Included in these experiments is the value-to-weight algorithm featured in the previous experiments (V2W), and a similar value-to-weight algorithm previously used as an incumbent for the branch and bound algorithm featured in section 5.2 (GAA). After being removed from the branch and bound program, it is now a pure value-to-weight greedy approximation algorithm represented below as 'GAA'.

The purpose of this experiment is to determine whether the performance in terms of value, memory consumption and time is better in the V2W algorithm featured in the previous experiments, or the GAA algorithm featured in the branch and bound project. If the GAA is found to have an improved performance over the algorithm implemented for this study (V2W), then GAA will be used in the next stage of the project.

Figure 13 and Figure 14 represent the range of problem sizes 50-100,000 with a capacity of 5000.

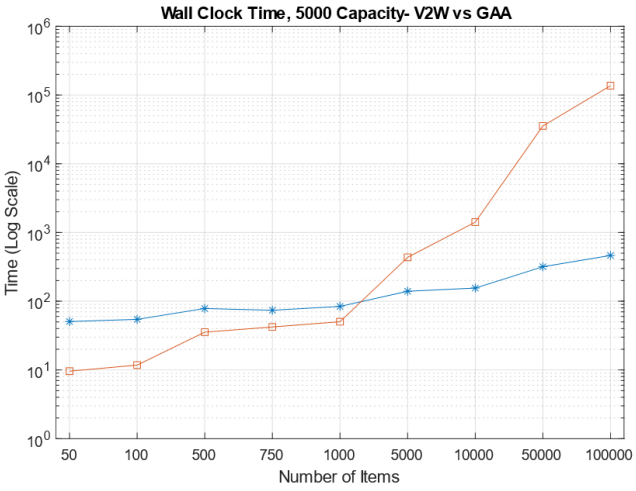


Figure 14. Wall Clock Time

Figure 13. Memory Consumption

Figure 13 uses a logarithmic scale to represent the wall clock time in milliseconds. As seen marked in blue, the V2W algorithm has a slightly higher wall clock time for problems with 50-1000 items ranging from 50-85ms. However, as the number of items increases above 1000 the wall clock time of GAA rises significantly above that of V2W. In the problem with 100,000 items, V2W takes 0.46 seconds, while GAA takes 136.8 seconds- nearly 300 times longer. This shows that V2W is optimised for large problems, and GAA is optimised for small problems. The memory usage also sees a slight improvement in the V2W algorithm, with lower memory usage for all but one problem.

A further experiment on problems with the largest capacity of 100,000 return almost exactly the same results, with V2W solving the largest problem instance in just 0.96s, compared to 208 seconds for GAA- 217 times longer. The memory usage also follows the same trend and is lower for V2W in 8 of the 9 problem instances.

5.5.2 Implementation of Value-to-Weight Greedy Algorithm

The implementation of the value-to-weight greedy approximation algorithm is simple compared to that of the branch and bound and genetic algorithms, and one algorithm suits all problem sizes without the need to modify or tweak any part of the algorithm.

The method for initialising the data involves reading the weights and values from file into respective value and weight arrays. An additional array is used to store the ratio of each item. Each time an item is read from file, the value to weight ratio is calculated by dividing the value by the weight and added to the ratio array. This allows an items weight, value and ratio to be accessed in the same way in the algorithm. If required, this can also be performed inside a loop in the algorithm, rather than as the data is read in from file, as shown in Figure 15.

The pseudocode for the greedy approximation algorithm can be seen in Figure 15. The initial step involves creating a list of pairs. The left-hand side of the pair represents the ratio of the item, and the right-hand side represents the index of the item in the list of items. The list of pairs is then populated with the ratios and indexes of each item and sorted by ratio in a descending order.

Using a list of pairs referencing the index of each item allows the original list to retain its order. This is important when listing the items included in the knapsack. After the

pairs list is populated and sorted, each item is added one-by-one from the highest ratio until either the capacity is exceeded, or the sum of included item weights is equal to the capacity. If the list is sorted in ascending order, items must be added from the end of the list rather than the start.

Once the knapsack is full, a simple loop which sums the values of every item included in the knapsack returns the maximised value. As the knapsack only contains the index of the items in the original list, by sorting the knapsack in ascending order the list of included items can be determined by printing the values in the knapsack. The weight of included items can be determined in the same way as the value by summing the weights of included items rather than the values.

```
1   // Create a list of pairs- the left value represents the ratio and the right represents the index in the original list
    // of items
2   CREATE list<Double, Integer> pairs
3
4   // Populate the pairs list with each items ratio, and its index in the items
5   FOR (i = 0 to i == dataSize)
6           double ratio = items.RATIO[i];
7           Pair<Double, Integer> p1 = Pair.of(ratio, i);
8           pairs.add(p1);
9
10  SORT pairs DESCENDING
11
12  // Create an empty knapsack
13  CREATE arrayList<Integer> knapsack
14
15  // Create an int to hold the sum of the weights, and a counter j
16  int sum = 0;
17  int j = 0;
18
19  WHILE (j < pairs.size)
20          sum = 0;
21
22  // For each index in the knapsack, add the weight to the sum
23          FOR (i = 0; i < knapsack.size; i++)
24                  sum += items.WEIGHTS[knapsack.get(i)]
25
26  // If the sum + weight of current item <= capacity, add the item
27          IF (sum + items.WEIGHTS[pairs.get(j).right] <= Capacity)
28                  knapsack.add(pairs.get(j).right)
29
30  // If the weight is equal to the capacity, then stop
31          IF (sum == Capacity)
32                  STOP
33
34  j++;
35
36  // Find the value by adding the values of the items in the knapsack, and return value
37  int finalValue = 0;
38
39          FOR (i = 0; i < knapsack.size; i++)
40                  finalValue += items.VALUES[knapsack.get(i)]
41
42  RETURN finalValue;
43
```

Figure 15. Greedy Approximation Algorithm Pseudocode

6 Results/Evaluation

6.1 Introduction

Included in this section is the findings of the experiments detailed in section 4.4. The averaged results can be found in appendix 4, and both the raw and averaged results can be found in full in a designated GitHub repository[3]. The experiments were conducted on a Dell Optiplex 7010 with a 3.4Ghz Intel i7-3770 CPU and 16GB of RAM. As stated in section 4.4.1, all algorithms are run a total of 30 times on each problem instance and can run for up to 10 minutes. All results in the following experiments are mean averages of the 30 runs.

The first part of this section details the findings of this study including how the algorithms performed in terms of satisfying the requirements and in comparison to each other with visualisations of the results. The second part details how the algorithms performed in comparison to the same algorithms featured in other existing studies. This is followed by a concluding section which details the most important findings and recommendations on the use of each algorithm.

6.2 Results

Across existing studies on the branch and bound algorithm a range of different restrictions are applied to the algorithms including a maximum of 36 seconds of runtime, maximum data sizes of 750 items and maximum capacities of 100 due to known time and memory constraints. Of particular interest to this study, however, is the results included in the study conducted by Shaheen & Sleit (2016), which uses a branch and bound algorithm to solve problem instances of up to 60000 items. Problems of this size are significantly larger than any problems solved by pure branch and bound algorithms in any other existing studies, and if replicable in this study would allow R6 to be satisfied.

In conducting experiments on the branch and bound algorithm, it is confirmed by this study that it can only solve instances of 750 items- far less than the 60000 hoped for. Furthermore, this is only the case for problems with a small knapsack capacity of 50 or 100 items. In problems with larger capacities of 500 or 1000, the branch and

[3] https://github.com/ross0605/KnapsackLargeProblemsDis

bound algorithm is only able to solve problems with up to 100 items, despite being limited to 600 seconds; significantly longer than the 36 second limit placed in ‘A new Lagrangian based branch and Bound Algorithm for the 0-1 Knapsack Problem’ (Salles da Cunha, Bahiense, Lucena, & Carvalho de Souza, 2010).

Tables 5-8 represent four problems, each with a maximum capacity of 50. In each of the problems the branch and bound algorithm successfully finds the optimum solution and is the only algorithm other than dynamic programming to achieve this in all four problems. However, the algorithm is unable to find the optimum value in any problems with a capacity larger than 50. Tables 9 and 10 show the highest capacity problems that the branch and bound algorithm can solve. With a capacity of 1000, the algorithm is unable to find the optimal solution, produces the smallest value out of all four algorithms, and is unable to solve any problems larger than 100 items.

Despite the branch and bound algorithm consistently producing the optimum value for problems with very low capacities and numbers of items, the time taken for the algorithm to find the solution is longer than both the dynamic programming and greedy approaches. As the number of items increases, the difference in time also increases further, with over 10 seconds difference between branch and bound and the dynamic and greedy algorithms in the problem with 750 items. The higher execution time is also reflected in the problems with higher capacities, in which the time is many times higher than both the dynamic programming and greedy approaches.

The memory usage of the algorithm is determined by the size of the state space tree. If a solution is found quickly the tree will be smaller, and therefore some problems will require less memory than others regardless of the number of items or the capacity. As seen in tables 5-7, the memory usage is high in both the 50 item and 100 item problems and drops over half for the 500 item problem, suggesting that the solution was found in a higher level of the state space tree requiring less branching and therefore less memory usage. This can also be seen in table 9 and 10, in which the memory usage is significantly smaller for the problem with more items.

Although the memory usage of the algorithm fluctuates, it is consistently higher than both the dynamic programming and greedy algorithm approaches, in some cases by over 100Mb. Although this is unlikely to cause memory issues for any modern

computers, paired with a higher execution time and lower maximum value for problems with larger capacities, this makes branch and bound an unfavourable choice for even the smallest problems. As the algorithm was also unable to solve problems over 750 items, it does not satisfy R6 and is therefore unsuitable for solving large instances of the 0-1 knapsack problem.

The reason that the branch and bound algorithm featured in the study conducted by Shaheen & Sleit (2016) can solve instances of 60000 items with 100 capacity is unknown. The algorithm in the referenced study is described as a standard implementation of a pure branch and bound algorithm, however as concluded by this study, and other existing studies, branch and bound algorithms are unable to solve instances of the problem this size.

50 Items – 50 Capacity

Algorithm	Total Value	Total Weight	Time(ms)	Memory(kB)
Dynamic Programming	316	49	0.8	2048
Genetic Algorithm	314	45	1541.1	97541.87
Branch & Bound	316	49	11.2	89866.8
Greedy Algorithm	311	45	48.5	2048

Table 7

100 Items – 50 Capacity

Algorithm	Total Value	Total Weight	Time(ms)	Memory(kB)
Dynamic Programming	467	47	0.9	2048
Genetic Algorithm	467	47	1921.13	17588.13
Branch & Bound	467	47	21.63	90316.43
Greedy Algorithm	467	47	50.83	2048

Table 6

500 Items – 50 Capacity

Algorithm	Total Value	Total Weight	Time(ms)	Memory(kB)
Dynamic Programming	1102	50	3.23	3072
Genetic Algorithm	1005	50	9290	104293.8
Branch & Bound	1102	50	954.87	43981.43
Greedy Algorithm	1083	49	66.43	3072

Table 5

750 Items – 50 Capacity

Algorithm	Total Value	Total Weight	Time(ms)	Memory(kB)
Dynamic Programming	1244	50	4.43	3447.47
Genetic Algorithm	1111	50	14746.2	110885.8
Branch & Bound	1244	50	10875.7	124054.53
Greedy Algorithm	1244	50	71.1	4096

Table 9

50 Items – 1000 Capacity

Algorithm	Total Value	Total Weight	Time(ms)	Memory(kB)
Dynamic Programming	1743	1000	4.8	2048
Genetic Algorithm	1694	996	3030.73	70833.87
Branch & Bound	1689	999	121.2	107416.43
Greedy Algorithm	1742	998	48.13	2048

Table 8

100 Items – 1000 Capacity

Algorithm	Total Value	Total Weight	Time(ms)	Memory(kB)
Dynamic Programming	2639	1000	5.13	2048
Genetic Algorithm	2513	998	4403.1	78930.03
Branch & Bound	2366	1000	3598.8	23336.03
Greedy Algorithm	2623	990	50.87	2048

Table 10

The results of the study referenced above also show that a 20000-item problem is solved by branch and bound in 1.37 seconds. This is significantly lower than the 10.88 seconds required to solve a 750-item problem seen in table 8. Both the size of the problem and execution time cannot be supported by this study or any of the other referenced studies, strongly suggesting that the results featured in Shaheen & Sleit's 2016 study are implausible for a pure branch and bound algorithm. An explanation for this could be that the branch and bound algorithm is hybridised with another algorithm, for example a greedy approximation algorithm. An example of this kind of hybridised branch and bound algorithm can be seen in section 5.2.

The results for the branch and bound algorithm in this study show that the limitations placed on the algorithm in existing studies are well-founded as both the memory and time consumption grow rapidly as the capacity and number of items increases. The algorithm is suitable for very small problems, however, is not the most time or

memory efficient for any problem sizes and cannot solve problems with over 750 items.

The genetic algorithm, like the branch and bound algorithm, is able to find good solutions for small problems, particularly those with 50-1000 items. The success rate for these problems remains low, with optimum values only found for a small number. However, the effectiveness of the algorithm for small problems is high, as seen in Figures 16-18, which shows the difference between the value found by the genetic algorithm and the optimal found by dynamic programming. These graphs show that despite having a low success rate, the genetic algorithm has high effectiveness for small problems with 1000 items or less regardless of the capacity of the knapsack.

As the number of items increases above 1000, the algorithm's effectiveness decreases substantially. In Figures 16 and 17, in large problems with 100,000 items the maximum value found is only 16.6% and 21.4% of the optimum respectively. This is considerably less effective than in smaller problems, and in part can be attributed to the 10-minute limit placed on the algorithm.

As seen in Figure 20, for very small problems with 50-100 items, the wall clock time is reasonable at less than 5 seconds for all but one problem. The time rises at a steady rate until 1000-item problems, with higher capacity problems taking slightly longer. However, as the number of items is increased to 5000 the wall clock time increases significantly, with several problems using the full 10-minute allocation. For problems with 50000 and 100,000 items the wall clock time is 10-minutes for every problem.

As the number of items and time increases, the number of generations taking place each second reduces significantly. With more items, the algorithm takes longer to apply each operator, meaning that although the algorithm is running for the same amount of time, the reduction in generations means that less optimisation is performed on the population, and the likeliness of reaching good solutions is hampered by the limit on time. As seen in Figure 19, each time the number of items increases, the number of generations can be seen decreasing. In 50 item problems, the number of generations ranges from 2576-3788 per second. In problems with

100,000 items, the number of generations per second reduces to an average of just 5.

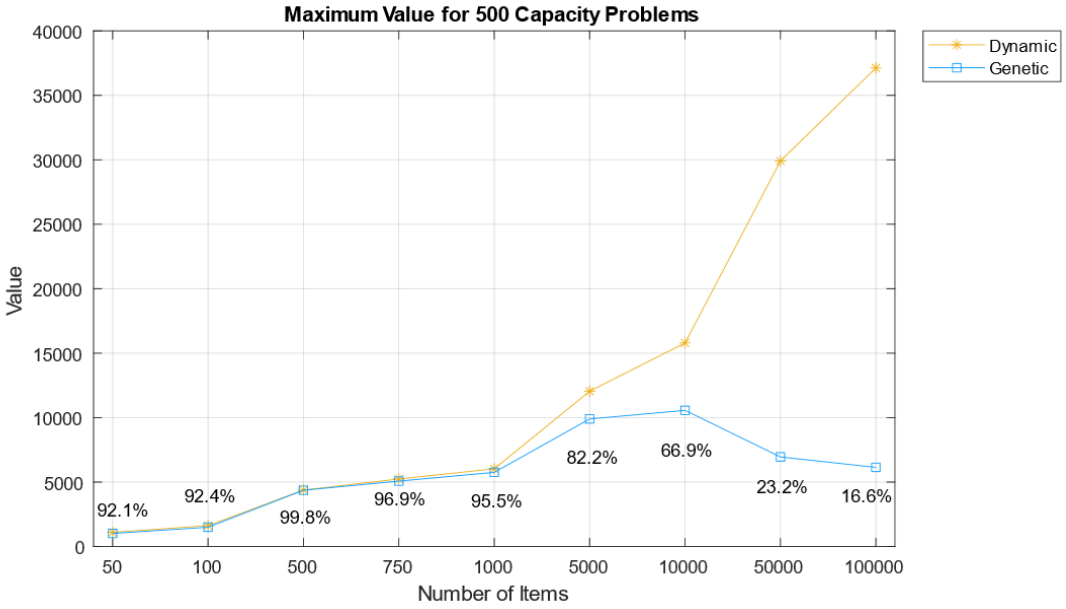


Figure 16. Low Capacity Problems

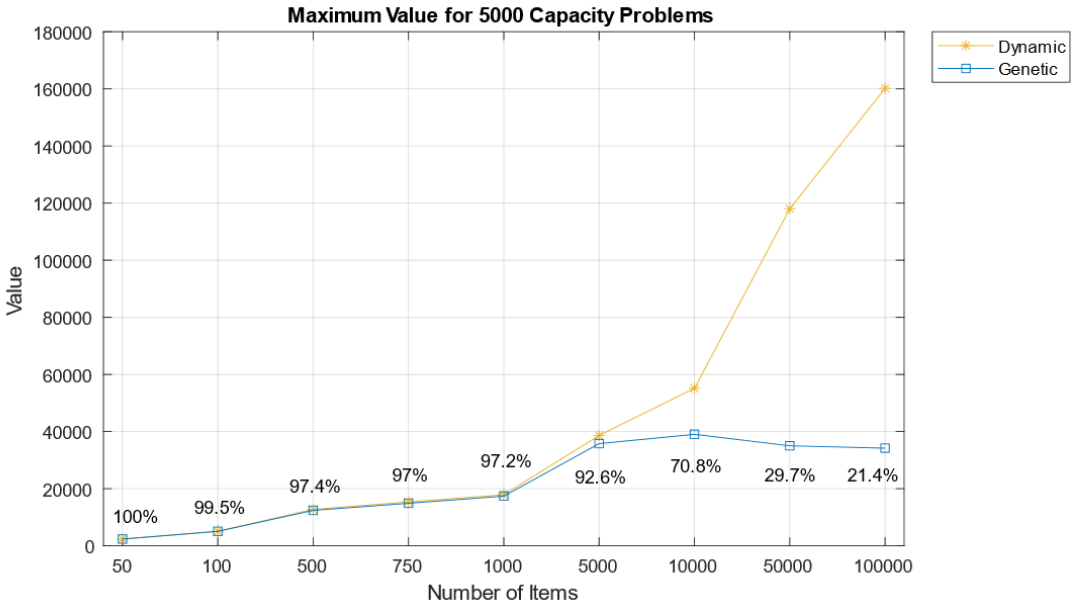


Figure 17. Medium Capacity Problems

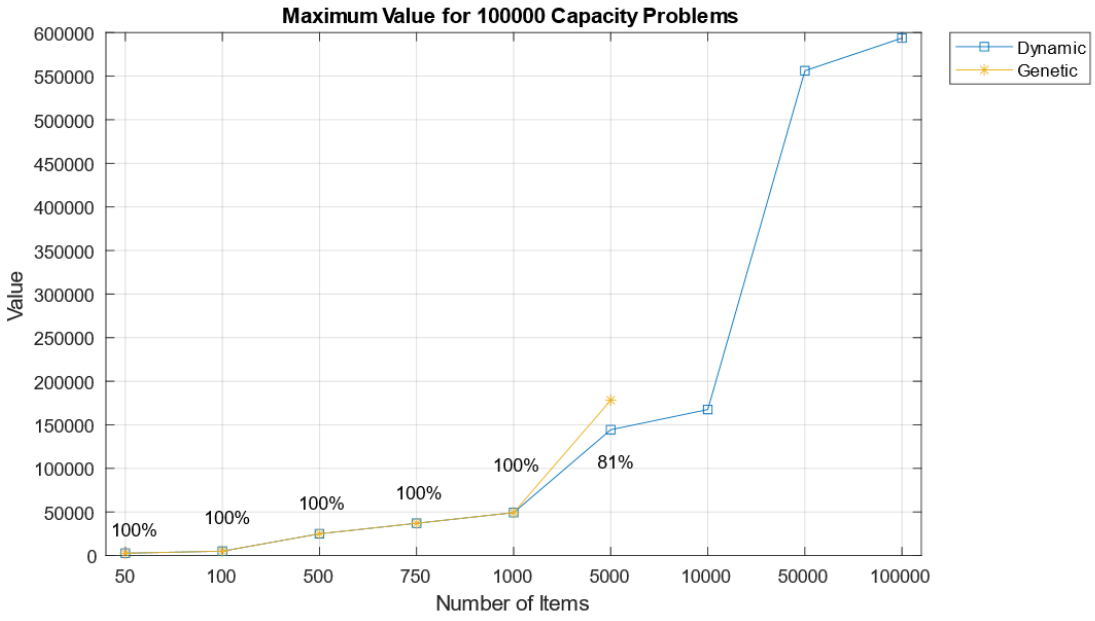


Figure 18. High Capacity Problems

The ability to find solutions in reasonable time is completely removed as the number of generations reduces. Compared to the other algorithms, the wall clock time on average for the genetic algorithm is higher for every single problem, regardless of the number of items or the capacity of the problem. Particularly for the largest problems, the genetic algorithm takes the entire 10-minutes for most problems, and while removing this limit would increase the value found by the algorithm, the run time could last several hours if allowed to continue until the stopping function is called.

The memory consumption of the algorithm is extremely high for all small and medium problems compared to both the dynamic and greedy methods, as seen in tables 5-10. Like the branch and bound algorithm, the memory usage is based on how quickly solutions are found, and so the memory is higher is some problems, and lower in others.

For larger problems with 50000 and 100,000 items it is much harder to identify trends as the time limit means that the algorithm is stopped prematurely for every problem. In problems with 50000 items, the memory fluctuates between 0.5Gb and 2Gb, seen in Figure 22. This memory usage is extremely different to problems with 100,000 items, in which problems with small capacities use little memory, and problems with capacities of 1000+ use roughly the same memory- between 2Gb and 2.5Gb. The memory usage for problems with 100,000 items can be seen in Figure 23.

It is clear from the above experiments that the genetic algorithm is better suited to small problems, in which it has a low success rate but high effectiveness, meaning that although it is unable to get the optimum value for most problems, the value returned is in a very close range to the optimum. Despite taking significantly longer than the branch and bound algorithm, the genetic algorithm is a more favourable choice for small problems, especially where the capacity might change or be set to above 1000, as the branch and bound algorithm is unable to solve many problems of this specification.

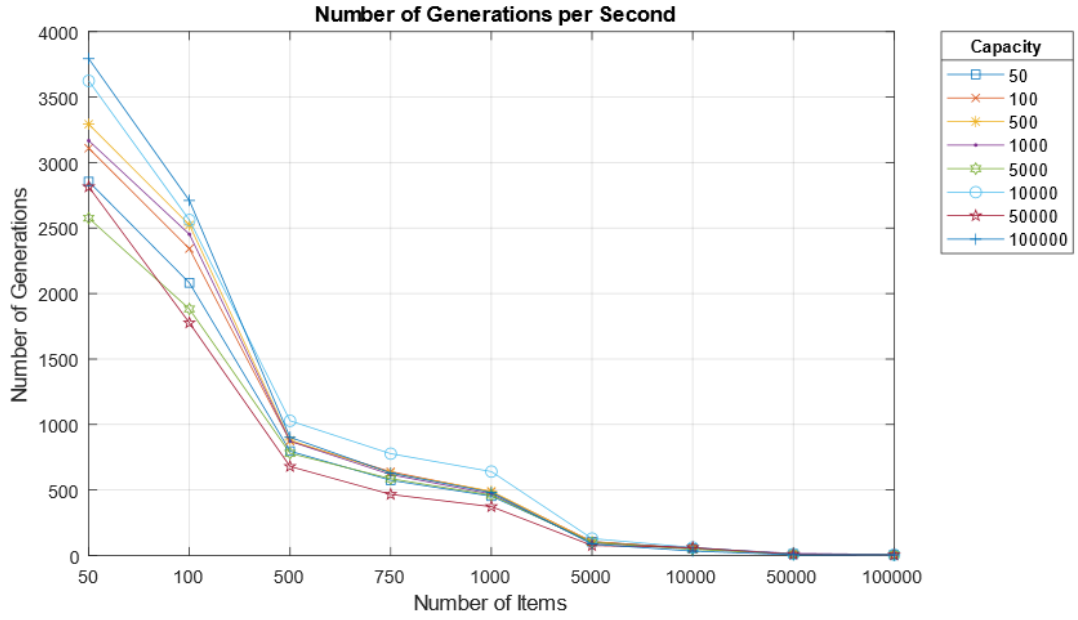


Figure 20. Genetic Algorithm Generations per Second

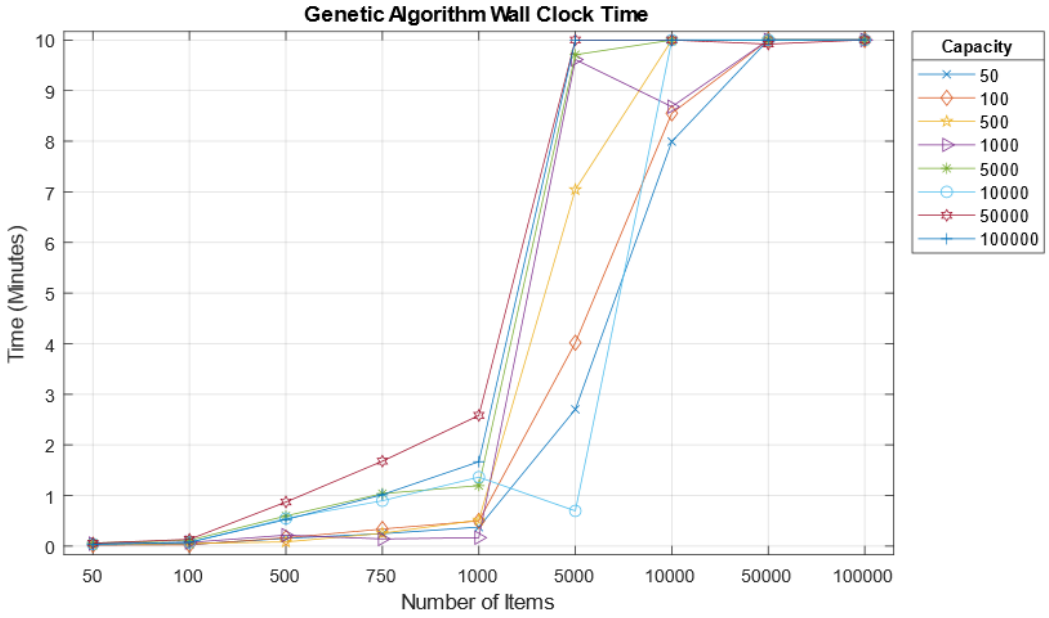


Figure 19. Genetic Algorithm Wall Clock Time

In solving larger instances of the problem, the solutions are of a lesser quality than those of smaller problems and compared to the optimum are in a very poor range. Additionally, the wall clock time of the algorithm is extremely long compared to the other algorithms, and without a 10-minute limit could exceed several hours each run for the largest problems.

A likely reason for the reduced number of generations and high memory usage is the group selection function. Despite being considerably more effective than other selection operators at evolving the population, it also requires the entire population to be sorted each generation in the same way as the greedy algorithm sorts the items. Swapping this operator for an operator that does not have to perform any sorting or modification of the population may help keep the number of generations higher in larger problems, allowing for more optimisation in the same timeframe.

Dynamic programming is the only exact algorithm featured in this study, meaning that it can solve every instance of the 0-1 knapsack problem optimally. While the value of the items in the knapsack is the most important constraint of the problem, as discovered in this study, the memory and time consumption of the algorithms in solving these problems are just as important, in some cases hindering the algorithm's ability to find solutions to the problems.

Dynamic programming calculates the optimum value by generating a table to store the results of subproblems with the dimensions [number of items + 1][capacity + 1]. This means that the memory consumption of the algorithm is directly affected by both the number of items and the capacity of the knapsack. The effects of this can be seen in larger problems, in which the algorithm was unable to solve several problems with large capacities and large numbers of items. When attempting to solve these problems on Eclipse, a 'Java heap space error' was returned after several seconds. A screenshot of this error can be seen in appendix 5.

For small problems with low capacities and numbers of items, the dynamic programming algorithm uses a very small amount of memory. For problems up to 1000 items, with capacities up to 1000, the memory ranges from 2048kB-8192kB (2MB-8MB). Compared to the memory usage of the branch and bound, genetic

algorithm and greedy algorithm for these problems, the dynamic programming algorithm uses a very small amount of memory. The results for some of these problems can be seen in tables 5-10 or in Appendix 4.

As the number of items and capacity are increased above 1000, and particularly if both are above this value, then the memory consumption increases very significantly. Included in Figure 21 is four groups of problems with the largest capacities- between 5000 and 100,000. In problems up to 1000 items, the memory usage remains very low, especially for the 5000 and 10000 capacity problems. However, as the capacity of the problem surpasses 1000, the memory usage rises significantly for the highest capacity problems and rises slightly for the smaller capacity problems due to the dimensions of the tables increasing.

In the problem instance with 5000 capacity and 100,000 items (the farthest right point of the blue marked line), the memory usage reaches just under 2GB. Increasing the capacity of this problem to 10000 doubles the dimensions of the table, from 500 million table fields to 1 billion fields. The effect of this is that the memory usage is doubled, as seen in the point above marked in orange which shows usage of around 4Gb. This trend of the memory usage doubling as the dimensions of the table are doubled can be seen in all problems with capacities of 5000 and 10000.

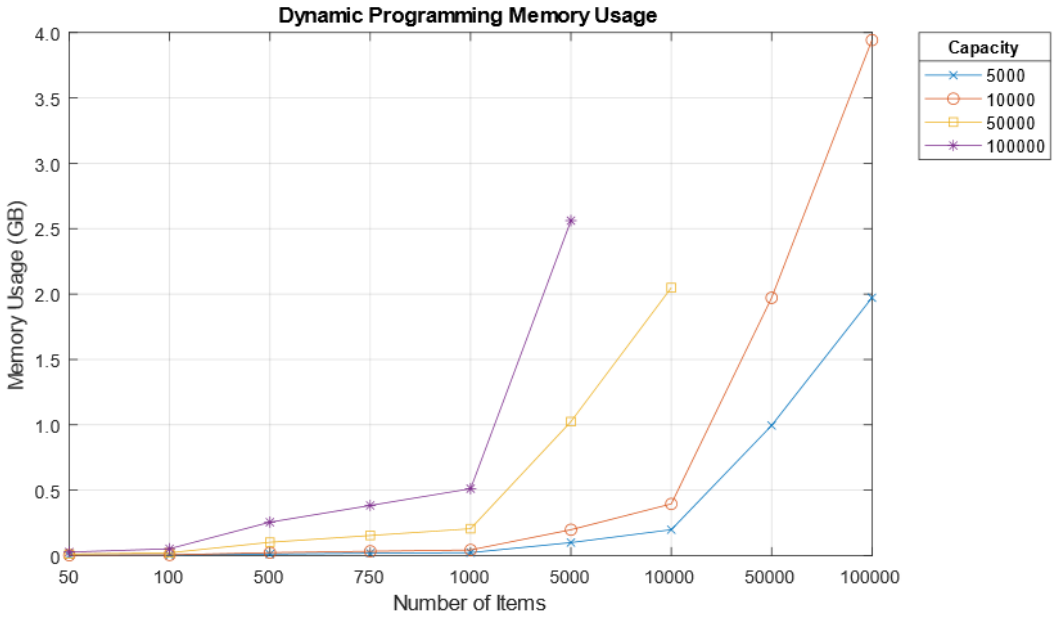


Figure 21. Dynamic Programming Memory Usage for Medium-High Capacity Problems

Despite being able to solve every problem with 5000 and 10000 capacity, the algorithm was unable to solve problems with both very large capacities and numbers of items. The unsolved problems can be seen in Figures 21-23, where the unsolved problems are not marked on the graphs. In running these problems, a heap space error is returned, meaning that there is insufficient heap space to construct the table. Calculating the number of fields in the table for one of the unsolved problems with 50000 capacity and 50000 items returns a total of 25 billion fields. This is significantly larger than the tables generated in smaller capacity problems and shows that problems in which both the capacity and number of items is high might pose memory issues due to the required size of the generated results table.

The wall clock time of the dynamic programming algorithm is extremely fast for problems in which both the capacity or the number of items is low, clocking just 0.8ms for the smallest problem with 50 items and 50 capacity. In problems where one is larger than the other, the algorithm is still very fast, in most problems taking less than 100ms. The largest item problems can be seen in tables 11 and 12. In these problems it can be seen that the time consumption of the algorithm remains very small showing that the dynamic programming algorithm is a very fast method of solving the knapsack problem.

The greedy approximation algorithm is able to find extremely good values for every problem, including large problem instances with high capacities and large numbers of items, and is able to solve all of the problems that the dynamic programming algorithm cannot due to much lower memory consumption.

Due to the workings of the greedy approximation algorithm and the small data coefficient of 100, when the greedy algorithm does not find the optimum solution it is always extremely close. Out of 67 problems with known optimum solutions, the greedy algorithm was able to solve 47 problems optimally. As the optimum value is not known for the 5 other problem instances, it cannot be determined whether they were solved optimally or not. If the coefficient of the data is increased, the difference between the optimum value and the greedy approximation value would be expected to increase, as the range of values is larger.

Unlike dynamic programming, the greedy algorithm is able to solve even the largest problem instances in this study with both high capacities and high numbers of items.

As the method the greedy algorithm uses to solve the problem is primarily based on the sorting of the data by value-to-weight ratio, the memory consumption of the algorithm is affected only by the number of items. For this reason, the memory usage is extremely low, and remains roughly the same for all problems with the same number of items, regardless of the knapsack capacity.

As seen in Figure 22 and 23, the memory usage of the algorithm does not increase in the same way as the other algorithms when the capacity is increased. Rather, because each of the problems shown in these graphs has the same number of items the memory does not change by more than a few megabytes for each problem. Figure 22 shows the memory usage for 50000 items, regardless of capacity, is around 93Mb for all problems. Surprisingly, the memory for the 100,000 item problems, seen in Figure 23 is lower, at 53Mb for all problems. The memory consumption of this algorithm is particularly impressive when compared to the consumption of the dynamic programming algorithm, which runs out of memory for several of the larger problems.

Not only is the greedy approximation algorithm the superior choice for memory usage, it also has very impressive wall clock times which scale extremely well with increasing numbers of items. The wall clock time for small problems with 50-1000 items ranges between 47ms and 106ms. Compared to the dynamic programming algorithm, these wall clock times are slightly higher for problems with low capacities. However, as the capacity is increased to above 10000, the greedy algorithm is able to solve small problems with low numbers of items faster.

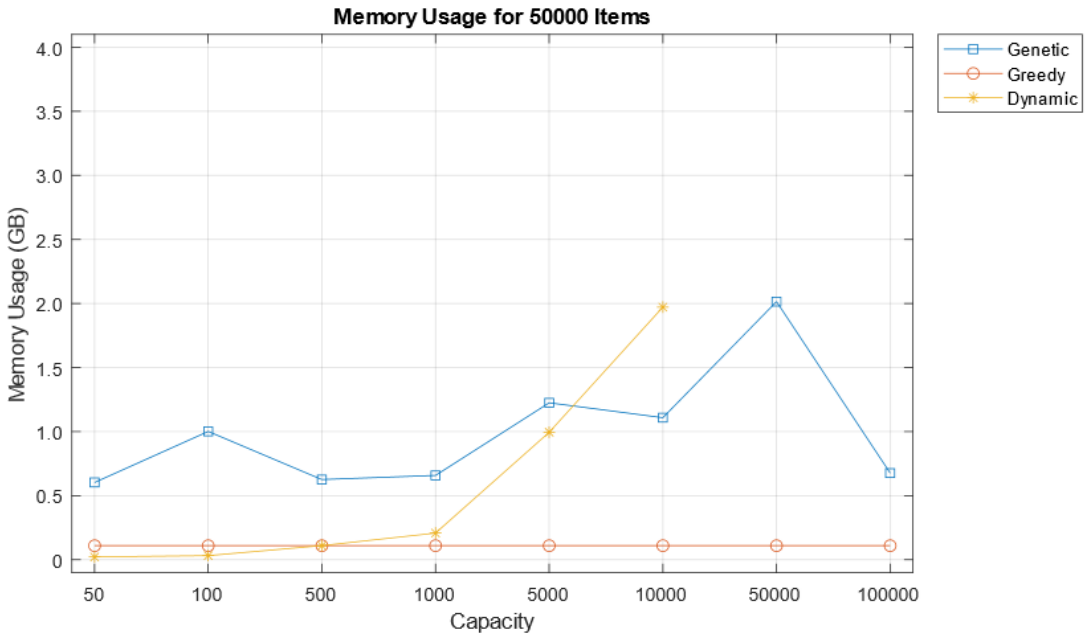


Figure 22. Memory Usage for Large Problems

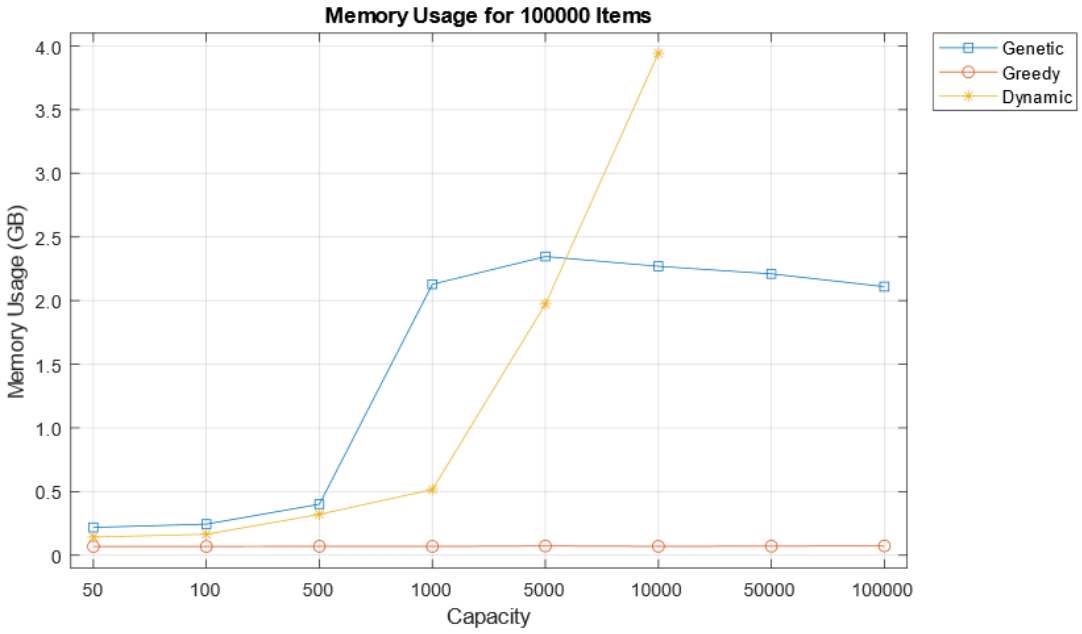


Figure 23. Memory Usage for Large Problems

In large instances of the problem, with 50000 and 100,000 items seen in tables 11 and 12, the time of the greedy algorithm rises very slightly each increase of capacity, remaining under 0.5 seconds for all problems with 50000 items and under 1 second for all problems with 100,000 items. Compared to the dynamic programming algorithm, the increase in time scales much better with the increase in capacity and highlights the effectiveness of the greedy algorithm in solving large instances of the 0-1 knapsack problem.

Capacity	Dynamic Programming (ms | s)	Greedy Algorithm (ms | s)
50	27.7ms | 0.03s	292.73ms| 0.29s
100	36.83ms | 0.04s	285.87ms | 0.29s
500	115.9ms | 0.12s	285.43ms | 0.29s
1000	229.63ms | 0.22s	313ms | 0.31s
5000	1101.47ms | 1.10s	317.67ms | 0.32s
10000	2437.63ms | 2.44s	333.17ms | 0.33s
50000	-	391.87ms | 0.39s
100000	-	460.77ms | 0.46s

Table 11. 50000 Item Problems

Capacity	Dynamic Programming	Greedy Algorithm
50	44.53ms | 0.05s	435.57ms | 0.44s
100	67.87ms | 0.07s	425.83ms | 0.43s
500	272.27ms | 0.27s	432.3ms | 0.43s
1000	496.53ms | 0.50s	497.37ms | 0.48s
5000	2443.97ms | 2.45s	463.7ms | 0.46s
10000	4497.77ms | 4.50s	483.9ms | 0.48s
50000	-	702.27ms | 0.7s
100000	-	956.33ms | 0.96s

Table 12. 100,000 Item Problems

6.3 Comparison to Results in Existing Studies

The branch and bound algorithm has been described as performing well for small sized “easy” problem instances (Martello et al., 2000), and can be seen in ‘Different Approaches to Solve the 0/1 Knapsack Problem’ solving problems with a maximum of 750 items (Hristakeva & Shrestha, 2005). In a third study, a 36-second limit is placed on the algorithm for solving problems (Salles da Cunha, Bahiense, Lucena, & Carvalho de Souza, 2010). For these reasons, the expectation for the algorithm would be that it might solve problems of 750-1000 items in a reasonable time under 36 seconds. However, a comparative study by Shaheen & Sleit (2016) determines that the branch and bound algorithm can solve problems with up to 60000 items in considerably less time than 36 seconds, casting doubt on the previous results.

After conducting the testing, it has been determined that the branch and bound algorithm in this study is able to solve problems of 750 items, supporting the findings of the first set of studies. Furthermore, this is only the case for problems with very small capacities, further supporting that the algorithm can only solve small sized “easy” problem instances.

In terms of the memory usage of the algorithm in relation to other studies, the only study to record these values is ‘Different Approaches to Solve the 0/1 Knapsack Problem’ (Hristakeva & Shrestha, 2005). The memory usage of the algorithm in the referenced study is assumed to be in kilobytes as there is no mention of the units. On this basis, it is considerably smaller for the 100-item problem, however, increases to twice the memory usage by the 750-item problem. Although unquestionably different than the memory usage in this study, both the dataset used and the coefficient of the data could play a big part in the difference, especially as different datasets will result in different sized state space trees.

Despite the above study measuring the memory usage it does not measure time in any form. The only similar comparative study to measure the time is ‘Comparing between different approaches to solve the 0/1 Knapsack problem’, which claims that a 20000 item problem can be solved by branch and bound in 1.37 seconds- significantly lower than the 10.88 seconds required to solve the 750-item problem in this study. As mentioned previously, it is likely that these results featured in Shaheen & Sleit’s 2016 study are implausible for a pure branch and bound algorithm, and therefore cannot be compared to.

The results for the branch and bound algorithm in this study show that the limitations placed on the algorithm in existing studies are well-founded, as the memory and time consumption grow rapidly as the capacity and number of items increases, eventually preventing the algorithm from solving the problem. The branch and bound algorithm is suitable for very small problems with up to 750 items and with low capacities. It is however, not the most time efficient or memory efficient algorithm for any problems, making it an unfavourable choice for all problems.

Studies on the genetic algorithm are more abundant in literature, and therefore it is easier to compare the results of this study to the results of others. However, the

majority of existing studies only test on very small datasets of less than 5000 items making comparison for large problems impossible.

The genetic algorithms featured in ‘Genetic algorithm with directional mutation based on greedy strategy for large-scale 0-1 knapsack problems’ (Liu & Liu, 2012) is tested on problems with 1000-5000 items, and like the algorithm included in this study is able to solve these problems with a high accuracy compared to the known optimum values. Another study, ‘Different Approaches to Solve the 0/1 Knapsack Problem’, although testing on smaller problems shows very similar results, with the genetic algorithm in most cases very close to the optimal value. In this referenced study, testing on problems with 100-1000 items the optimum value is found 2/5 times, and the margin between the results and the optimum value is very small in problems not solved optimally (Hristakeva & Shrestha, 2005).

The memory usage recorded in the study referenced above for the problems with 100-1000 items ranges from 15000kB-150,000kB. Although this is smaller than the memory usage of the algorithm in this study, features such as the operators and stopping function can have a huge effect on this measurement.

In comparing the time usage of the algorithm with other studies, the only study that has problems of similar sizes and measures the execution time is carried out by Shaheen & Sleit 2016, and as mentioned previously is not a credible source. In this study, problems with 100,000 items are suggested to have been solved in 0.7325s., significantly lower than the 10 minutes used in this study. Although the genetic algorithm in this study has been written in C++, a speed improvement this large is unlikely to be attributed to the algorithm being written in a different language.

Despite extensive existing research on genetic algorithms in solving the 0-1 knapsack problem, finding studies which have credible results for large datasets is extremely difficult as the majority of studies test on problems of 5000 items or less. For this reason, it was hard to predict how the algorithm would perform on large datasets. As there are no studies with results comparable to the results of this study, it is impossible to determine if the performance of the algorithm on large datasets in this study would be similar to the performance of other existing genetic algorithms on problems of this size.

The dynamic programming approach, unlike genetic algorithms, has the problem that that there are few studies on the performance of the algorithm. This may be due to the fact that because the algorithm is already well understood, or people already know how they expect the algorithm to perform, removing the need for any studies.

As the datasets between studies are different, the optimum values found by dynamic programming cannot be compared across different studies. However, in problems with the same data size and capacity the memory usage and time can be compared.

In the study performed by Shaheen & Sleit, 2016, the algorithm is tested on data sizes of up to 600,000 items. The results of this referenced study so far have shown the branch and bound and genetic algorithms to be unrealistically fast. However, the results of the dynamic programming algorithm appear to be more realistic in comparison to the results in this study. In the referenced study, in a problem with 50000 items and 100 capacity, the execution time is recorded at 0.6363 seconds. The algorithm in this study solves the same problem in just 0.0036 seconds- considerably faster.

While the dynamic programming algorithm used in this study appears to be faster than the algorithm featured in the above study, it also uses less memory for small problems than the algorithm featured in ‘Different Approaches to Solve the 0/1 Knapsack Problem’ (Hristakeva & Shrestha, 2005). For 100 item problems in the above study, the memory usage is 5100kB, compared to a usage of 2048kB by the algorithm in this study. This trend is continued for a 500-item, 500-capacity problem in which the memory usage is 250500kB, compared to just 4096kB in this study. For all comparable problems between this study and the referenced study, the memory usage of the dynamic programming algorithm featured in this study is lower, showing that it is has higher memory efficiency. Comparing the algorithm between this study and others, the algorithm in this study appears to be both faster and more memory efficient than the dynamic programming algorithms featured in similar comparative studies.

The greedy approximation algorithm suffers from a similar problem to the dynamic programming approach, in that there exists several studies on the algorithm in solving the 0-1 knapsack problem, however the majority do not perform any kind of experimentation and do not contain results. For this reason, the only studies with

comparable results are ‘Comparing between different approaches to solve the 0/1 Knapsack problem’ (Shaheen & Sleit, 2016) and in ‘Different Approaches to Solve the 0/1 Knapsack Problem’ (Hristakeva & Shrestha, 2005). - those used for comparing the results of dynamic programming to.

In Shaheen & Sleit’s 2016 study, testing on large datasets of 50000 and 100,000 items with a capacity of 50, the results are more in line with the greedy algorithm than they are for any of the other algorithms, which see massively reduced execution times as discussed previously. In the 50000-item problem, the time in the referenced study is 0.255 seconds, compared to 0.285 seconds in this study. In the 100,000 item problem however, the time is over double at 0.862 study compared to 0.425 seconds in this study. As these times are similar for large problem instances, it can be assumed that the times would be in a similar range compared to each other for smaller problems and suggests that the algorithms featured in both studies have similar performances for large datasets.

The memory consumption of the algorithm does not fall in line with the consumption in ‘Different Approaches to Solve the 0/1 Knapsack Problem’, which tests on smaller problems of 100-1000 items. In these problems, the memory is denoted as 100 for the 100-item problem, and 1000 for the 1000-item problem. As mentioned above, the unit of measurement is assumed to be kilobytes, however at 2048kB and 4096kB in this study respectively, the memory consumption does not match up.

Despite the discrepancies between the memory usage of the algorithm in this study and the above comparative study, the algorithm in this study has at least as good a performance compared to the algorithms seen in other studies, and has been shown to have a considerably higher performance than the greedy algorithm discussed in section 5.5.1.4.

The comparative study by Hristakeva & Shrestha, 2005 concludes that the choice in algorithms for solving the 0-1 knapsack problem are dynamic programming and genetic algorithms, with the choice based on the capacity of the knapsack and the number of items. The other study mentioned frequently above, performed by Shaheen & Sleit, 2016 concludes that the branch and bound and dynamic programming methods outperform the greedy and genetic algorithms. This study only partially agrees, giving credit to the dynamic programming method for its ability to

solve small and medium sized problems efficiently, however, finds that in solving large instances of the problem the greedy approximation algorithm has a superior performance over the other algorithms.

6.4 Conclusion of Results

The performance of the algorithms on the 72 problems have revealed interesting trends for each. In comparison to each other, they all have very different performance levels and are suited to different sizes of problem. Each algorithm has been tested in compliance with all relevant requirements detailed in section 3.2, and the results are both accurate and credible.

The fastest algorithm for problems with low capacities is dynamic programming, which solves all low-capacity problems including those with high numbers of items faster than the greedy and branch and bound algorithms. For all problems with a capacity less than 1000, the wall clock time for the dynamic programming algorithm is less than 0.5 seconds.

For problems with higher capacities, particularly over 1000, the greedy algorithm is much faster for most problems than dynamic programming, especially for problems with higher numbers of items. For large problems, in which the number of items is 50000 and 100,000, the greedy algorithm is extremely fast with a maximum wall clock time of less than 1 second for all problems. As the dynamic programming algorithm is unable to solve many of the large problem instances, the greedy algorithm is the most time-efficient algorithm featured in this study.

The success rate of the algorithm denotes how often it finds the optimum solution to the problem. For this measurement, problems in which a solution is not presented are not counted in the rate. The dynamic programming algorithm, as expected, has the highest success rate at 100%, however was unable to solve five of the 72 problems. The greedy algorithm was able to solve 72% of the problems optimally where the optimal solution is known and solved the rest of the problems to a very high degree including those that dynamic programming could not solve. The branch and bound algorithm solved 4 out of the 12 problems optimally, while the genetic algorithm solved just 19% of problems optimally.

The effectiveness of an algorithm is determined by the quality of solutions and is very good for all algorithms in small problems, suggesting that all four algorithms are effective methods for solving small instances of the problem. For medium and large sized problems, the branch and bound algorithm is unable to solve all problems and the genetic algorithm returns very poor solutions to these problems in a reasonable time. The greedy algorithm is able to produce very good solutions to problems of all sizes, and dynamic programming produces the optimal value for all solvable problems.

Despite the dynamic programming algorithm having such high performance in terms of the wall clock time and solutions found, the memory usage does not scale well for large problems. When the number of items and the capacity are both high, the algorithm has a very high memory usage and some of the largest problems cannot be solved due to memory constraints. Conversely, the greedy approximation algorithm has very low memory usage for all problems as the memory is not affected by the capacity of the knapsack. This means that even very large problems with both high capacities and numbers of items have a very low memory usage.

In a situation where finding the optimum solution is required, or is very important, the dynamic programming algorithm is a very good choice of algorithm. It is easy to implement, very fast, and guarantees the optimal solution every time. However, if the capacity of the knapsack or the number of items is likely to be high at any time then the greedy algorithm is by far the best algorithm to choose. It is extremely fast, easy to implement in all languages and has a high success rate. Despite not finding the optimum solution every time, the algorithm is guaranteed to find very good solutions very close to the optimum extremely quickly.

The suitability of the algorithms for use on large datasets ranges. Both branch and bound and the genetic algorithm are unsuitable for large problems due to extremely poor solutions or no solutions at all. The dynamic programming algorithm is suitable for most large problems, however where the capacity and number of items are both too high, the memory consumption is likely to cause issues for most computers. The greedy algorithm is by far the most suitable algorithm for solving large instances of the 0-1 knapsack problem, proving more time and memory efficient for all large

problem instances, and answers the research question: “Which of the algorithms performs best in solving large instances of the 0-1 knapsack problem?”

For ease in choosing the most appropriate algorithm for an individual problem, all measures of performance can be seen in table 13.

	Dynamic Programming	**Genetic Algorithm**	**Branch and Bound**	**Greedy Algorithm**
Speed	Very fast	Very slow	Fast for very small problems	Very fast
Success Rate	100%	19%	33%	72%
Effectiveness	Optimal	Effective for small problems. Ineffective for medium/large problems	Effective for very small problems, reduces as capacity increases	Very effective for all problems
Memory Usage	Dependent on number of items and capacity. Ranges from very low to very high. (See page 76)	Very high	Very high	Very low
Large Datasets	Suitable for some large datasets if the capacity is low/medium	Suitable for large datasets if the capacity is very low or larger than the sum of all item weights	Unsuitable	Suitable for problems of 100,000 items (and very likely 100,000+)

Table 13. Algorithm Performance Measures

7 Conclusion

This comparative study performed on the dynamic programming, branch and bound, greedy approximation and genetic algorithms shows that the nature of the 0-1 knapsack problem, particularly when large datasets are involved, makes some of the algorithms more applicable than others. Despite the existence of extensive research and materials, it is difficult to find credible sources which detail the performances of the algorithms in solving large instances of the problem in the range used in this study.

As discovered by this study, the best algorithm for solving large instances of the problem with 50000-100,000 items is the greedy approximation algorithm, which displays both very low wall clock time and memory usage. Despite prior knowledge of high memory usage, it is surprising to find that the dynamic programming algorithm, which is recommended in many similar comparative studies, is unsuitable for solving some of the larger problems on the average computer due to memory limitations.

Despite extensive parameter and operator tuning, the genetic algorithm proved to be a very poor candidate in solving large instances of the problem. Although in other studies it displays high performance compared to other algorithms, the genetic algorithm in this study was only effective in problems with up to 1000 items. The performance of the genetic algorithm was significantly worse than expected compared to those featured in existing studies, and while further optimisation of the algorithm could improve the performance, this study shows that it requires a huge amount of effort and problem specific knowledge to be suitable for even medium sized problems.

It was surprising to find that the capabilities of the branch and bound algorithm fall exactly in line with the algorithms featured in most existing studies. This study was also able to disprove the results in one study which was identified as having suspicious results. For the complex implementation of the algorithm, it was disappointing to find that it is only suitable for solving very small problems. Interestingly, the algorithm with the easiest implementation- the greedy approximation algorithm- is the most suitable for large problem instances, highlighting that in some cases simple methods can be the most effective.

The two algorithms which best satisfied requirements R5-R7 were the greedy algorithm and the genetic algorithm. Both algorithms were able to solve all problem instances with varying degrees of success, as the genetic algorithm had a very poor performance for many problems. However, the real choice of algorithm is between the greedy algorithm and dynamic programming algorithm and ultimately depends on the requirements of the user. If it is essential to find the optimum value, or if the data size or capacity is low then dynamic programming is a suitable choice of algorithm. However, if the data size or capacity is likely to be large, or if high memory usage could pose an issue then the greedy algorithm is a very good choice of algorithm for providing optimal or near-optimal solutions very quickly.

During the course of the project there were several times when problems and issues surfaced that required additional efforts to amend in order to ensure that every aim and requirement was satisfied to the best possible standard.

The first aim, A1, detailed in the project aims in section 1.2 was only partially satisfied due to difficulties with implementing the branch and bound algorithm. During the implementation stage of the project the requirement was to design and write each of the four algorithms in Java. While I was able to implement the dynamic programming, greedy and genetic algorithms, I faced difficulties in producing a working branch and bound algorithm. Despite initial attempts resulting in an algorithm able to solve instances of up to 25 items, I was unable to optimise the algorithm to solve any problems larger than this, with either no solution or infeasible solutions returned by the algorithm for unknown reasons. I consulted numerous online sources including existing studies and tried several different versions of the algorithm, however I was unsuccessful in designing and implementing the algorithm of my own accord.

As the branch and bound algorithm is a key component of the project, I was able to source a branch and bound algorithm through a project referenced in literature and modified this program to fit into this study. Although disappointed that I could not produce a working algorithm, in utilising an algorithm known to work from a different source I was able to adapt to the situation by modifying a pre-existing algorithm.

The other aim which was satisfied to a very good degree, but not completely satisfied is A2, which states ‘Test the algorithms on a range of problem types with varying numbers of items, coefficients and capacities’. Initially, the plan for this project was to

test the algorithms on a range of problem types, including those with different data coefficients. After planning the testing stage however, I realised that this would massively increase the amount of testing required and the size of the results. Ultimately, it was decided that testing on a range of coefficients, as well as numbers of items and capacities would increase the complexity of comparing the results, not only within this study, but also between existing studies. Additionally, to perform rigorous testing to the same degree as the testing performed for more than one coefficient would significantly increase the time required for testing and may have caused timescale problems. In choosing a single coefficient for all problems, I was able to prioritise the number of items and capacity of the knapsack and provide the highest quality results possible in a sensible timescale.

The requirements, laid out in section 3.2, were created to keep the algorithm development focussed on large instances of the problem, and were a useful guideline to refer to when faced with decisions that could affect the project. For example, after implementing the first version of the genetic algorithm, it became clear that it did not satisfy as many of the requirements as would be hoped. This made it important to modify and optimise the algorithm further in order to best satisfy the requirements. I have learned that part of performing a successful project is knowing exactly what you wish to achieve and the ways in which these achievements can be made. In writing a list of aims and requirements, I was able to make important decisions regarding several aspects of the project and stay focussed on task in order to produce a successful project.

A time in which a difficult decision had to be made was when the 10-minute limit had to be placed on the runtime of the algorithms. During the testing stage each algorithm had to be run a total of 2160 times. I became aware during the genetic algorithm testing that the time required for medium-sized problems was taking increasingly long- in some cases over 60 minutes per run. Ideally, for the most accurate depiction of the algorithm’s performance it would be allowed to run until the stopping function is called, regardless of the time. However, I estimated that to perform the testing in this manner, allowing the algorithm to run until the stopping function is called for each run could have taken several weeks of continuous running. Although I was very keen not to impose a time-limit, I had a responsibility of timekeeping and had to make the choice in order to guarantee finishing the project on time. Imposing this limit still

means that the testing is performed in a structured and fair manner as the limit is applied across all algorithms, and therefore still satisfies the project requirements.

Imposing a 10-minute limit on each run, I was able to complete the testing of the genetic algorithm in roughly 153 hours, and although nearly a week straight of continuous testing, this put me only slightly behind schedule on the testing and left more than enough time to continue with other tasks labelled on the Gantt chart (Appendix 3.1). This situation helped reinforce the idea of good timekeeping skills and planning ahead. If I had not identified early enough that the testing of the genetic algorithm could cause time issues, then I may have struggled to complete the testing stage, impacting on the rest of the project.

Although there were challenges during the course of the project, these were expected, and the project as a whole was a major success. One of the most important requirements of the project was that algorithms are tested in a ‘consistent and structured manner’. This requirement was fully achieved with algorithms all written in Java, tested on the same computer and all given the same opportunities to solve the problem. The result of these actions is accurate, credible results which fit into active areas of research: combinatorial optimisation and the knapsack problem.

In looking back at my expectations at the beginning of the project, I am very satisfied with how the project has been presented. There are far more interesting trends in the data than I first anticipated, and the results- especially in favouring the greedy algorithm- are accurate and credible. I am very satisfied that this project fits in with other existing comparative studies, and strengthens the arguments made in several, while casting doubt on the results in a study that cannot be supported by other comparative studies.

7.1 Future Work

Future work that should immediately follow this study is an investigation into how changing the coefficients of the data affects the performance of the dynamic programming and greedy approximation algorithms in relation to each other, and their ability to solve large instances of the problem. Due to the nature of the greedy algorithm, increasing the coefficient of the data could have an impact on how close the solutions found are to the optimum value. Similarly, this could also further restrict

the problems that dynamic programming is suitable for if changing the coefficient increases the memory usage.

Expanding on this, different algorithms could be added to the study, for example more modern hybridised algorithms or algorithms designed to be particularly efficient in solving the 0-1 knapsack problem. Many of these ‘improved’ algorithms are being published frequently, and it would be interesting to compare the performance of more traditional algorithms with modern, novel algorithms.

Another area of research could increase the size of the problems even further and focus on the upper bounds of the algorithms- how large can the problems get before algorithms become unsuitable? The upper bound of the greedy approximation algorithm is unknown at this stage, and with extremely low memory and time consumption, determining how large problems can be when using the greedy algorithm may help in choosing an appropriate algorithm in situations with extremely large problems.

Finally, as mentioned first by D. Pisinger, there are very few real-life instances of the problem in literature, leading to algorithm design being focussed primarily on synthetic benchmark tests (Pisinger, 2005). A very good area of future research would be finding real instances of the problem, and optimising algorithms for these real sets of data to determine if there is any change in performance compared to the randomly generated test data used in the majority of studies. Observing these changes could result in different methods or algorithm designs in solving real instances of the problem.

References

Balas, E., & Zemel, E. (1979). *An Algorithm for Large Zero-One Knapsack Problems*.

Bhattacharjee, K. K., & Sarmah, S. P. (2014). Shuffled frog leaping algorithm and its application to 0/1 knapsack problem. *Applied Soft Computing Journal*, *19*, 252–263. https://doi.org/10.1016/j.asoc.2014.02.010

Chebil, K., & Khemakhem, M. (2015). A dynamic programming algorithm for the Knapsack Problem with Setup. *Computers and Operations Research*, *64*, 40–50. https://doi.org/10.1016/j.cor.2015.05.005

Cho, M. (2018). *Analysis of Applied Mathematics*. *11*.

Deb, K., Agrawal, S., Pratap, A., & Meyarivan, T. (2000). A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *1917*, 849–858. https://doi.org/10.1007/3-540-45356-3_83

Della Croce, F., Salassa, F., & Scatamacchia, R. (2017). An exact approach for the 0–1 knapsack problem with setups. *Computers and Operations Research*, *80*, 61–67. https://doi.org/10.1016/j.cor.2016.11.015

Feng, Y., Jia, K., & He, Y. (2014). An improved hybrid encoding cuckoo search algorithm for 0-1 knapsack problems. *Computational Intelligence and Neuroscience*, *2014*. https://doi.org/10.1155/2014/970456

Gomory, P. C. G. and R. E. (1961). *A Linear Programming Approach to the Cutting-Stock Problem*. *9*(6), 849–859.

Horowitz, E., & Sahni, S. (1974). *Computing Partitions with Applications to the Knapsack Problem*. *21*(2), 277–292.

Hristakeva, M., & Shrestha, D. (2005). *Different Approaches to Solve the 0 / 1 Knapsack Problem Simpson College The Knapsack Problem (KP)*. 0–14.

Kumar, R., & Banerjee, N. (2006). Analysis of a Multiobjective Evolutionary Algorithm on the 0-1 knapsack problem. *Theoretical Computer Science*, *358*(1), 104–120. https://doi.org/10.1016/j.tcs.2006.03.007

Li, K., Liu, J., Wan, L., Yin, S., & Li, K. (2015). A cost-optimal parallel algorithm for the 0-1 knapsack problem and its performance on multicore CPU and GPU implementations. *Parallel Computing*, *43*, 27–42. https://doi.org/10.1016/j.parco.2015.01.004

Liu, W., & Liu, G. (2012). Genetic algorithm with directional mutation based on greedy strategy for large-scale 0-1 knapsack problems. *International Journal of Advancements in Computing Technology*, *4*(3), 66–74. https://doi.org/10.4156/ijact.vol4.issue3.9

Martello, S., Pisinger, D., & Toth, P. (1999). Dynamic programming and strong bounds for the 0-1 Knapsack Problem. *Management Science*, *45*(3), 414–424. https://doi.org/10.1287/mnsc.45.3.414

Martello, S., Pisinger, D., & Toth, P. (2000). New trends in exact algorithms for the 0-1 knapsack problem. *European Journal of Operational Research*, *123*(2), 325–332. https://doi.org/10.1016/S0377-2217(99)00260-X

Melanie, M. (1999). *An Introduction to Genetic Algorithms*. https://doi.org/10.1016/S0898-1221(96)90227-8

Pisinger, D. (2005). Where are the hard knapsack problems? *Computers and Operations Research*, *32*(9), 2271–2284. https://doi.org/10.1016/j.cor.2004.03.002

Pisinger, D. (1999). Core problems in knapsack algorithms. *Operations Research*, *47*(4), 570–575. https://doi.org/10.1287/opre.47.4.570

Pisinger, D. (1995). *Algorithms for Knapsack Problems*. Retrieved from http://www.ghbook.ir/index.php?name=فرهنگ و رسانه های نوین&option=com_dbook&task=readonline&book_id=13650&page=73&chkhashk=ED9C9491B4&Itemid=218&lang=fa&tmpl=component

Pisinger, D. (1998). A fast algorithm for strongly correlated knapsack problems. *Discrete Applied Mathematics*, *89*(1–3), 197–212. https://doi.org/10.1016/S0166-218X(98)00127-9

Plateau, G., & Elkihel, M. (1985). A hybrid method for the 0-1 knapsack problem. *Methods of Operations Research*, *49*, 277-293.

Sahni, S. (1975). Approximate Algorithms for the 0/1 Knapsack Problem. *Journal of the ACM (JACM)*, *22*(1), 115–124. https://doi.org/10.1145/321864.321873

Salles da Cunha, A., Bahiense, L., Lucena, A., & Carvalho de Souza, C. (2010). A new Lagrangian based branch and Bound Algorithm for the 0-1 Knapsack Problem. *Electronic Notes in Discrete Mathematics*, *36*(C), 623–630. https://doi.org/10.1016/j.endm.2010.05.079

Sbihi, A. (2007). A best first search exact algorithm for the Multiple-choice Multidimensional Knapsack Problem. *Journal of Combinatorial Optimization*, *13*(4), 337–351. https://doi.org/10.1007/s10878-006-9035-3

Shah, S. (2019). *Genetic Algorithm for the 0/1 Multidimensional Knapsack Problem*. 5–8. Retrieved from http://arxiv.org/abs/1908.08022

Shaheen, A., & Sleit, A. (2016). Comparing between different approaches to solve the 0/1 Knapsack problem. *IJCSNS International Journal of Computer Science and Network Security*, *16*(7), 0–10. Retrieved from http://paper.ijcsns.org/07_book/201607/20160701.pdf

Toth, Paolo. "Dynamic programming algorithms for the zero-one knapsack problem." *Computing* 25.1 (1980): 29-45.

Zhao, J., Huang, T., Pang, F., & Liu, Y. (2009). Genetic algorithm based on greedy strategy in the 0-1 knapsack problem. *3rd International Conference on Genetic and Evolutionary Computing, WGEC 2009*, 105–107. https://doi.org/10.1109/WGEC.2009.43

Zhao, P., Zhao, P., & Zhang, X. (2006). A new ant colony optimization for the Knapsack problem. *2006 7th International Conference on Computer-Aided Industrial Design and Conceptual Design, CAIDC*, 4–6. https://doi.org/10.1109/CAIDCD.2006.329439

Appendix 1 Project Overview

Initial Project Overview

SOC10101 Honours Project (40 Credits)

Title of Project: Branch & Bound vs. Dynamic Programming vs. Genetic Algorithms in solving the 0-1 Knapsack Problem

Overview of Project Content and Milestones

The aim of this project is to compare the effectiveness of three different algorithms when solving the 0-1 knapsack problem. The initial research into the problem and its applications in the real world will be essential for determining the size of datasets I will create and measurables recorded in the software. This research will also form the foundations of my literature review, which should help to support my findings.
The software I am creating will take in a dataset and apply each algorithm to solve the problem. The results, including several measurables will be used to form graphical representations of the results of each algorithm. Using graphs will help to visualise the results and compare each of the algorithms directly against each other in a way that can be easily understood.
Milestones include completing the literature review, application of algorithms and testing, data collection and comparison of results.

The Main Deliverable(s):

- Literature review containing research, results and findings from recent published literature.
- Simple program which accepts both user input and predefined datasets and can perform branch & bound, dynamic programming and genetic algorithm methods on the datasets and output a series of results.
- Clear comparison of results in graphical and textual form including recommendations on the use of each algorithm.

Research Question(s):

Which of the algorithms when solving the 0-1 Knapsack problem has the highest accuracy?
Which of the algorithms performs best when solving large datasets (>10000) in the 0-1 Knapsack problem?

The Target Audience for the Deliverable(s):

People involved with solving real-world instances of the knapsack problem such as the cutting of raw materials (to save money and avoid wastage), selection of investments (to maximise profit and stay within budget) or loading of shipping containers (to pack effectively).
Researchers, students and professionals interested in combinatorial optimisation.

The Work to be Undertaken:

- Research into history, variations, real-world applications, existing literature and studies, and known ways of solving the 0-1 knapsack problem.

- Building of datasets ranging in size from several items to thousands of items.
- Building and implementation of software, including optimisation of algorithms.
- Rigorous testing of software to ensure the accuracy of results.
- Collection and storage of sets of results.
- Creation of graphs from recorded results.
- Analysis and evaluation of results and graphs.
- Formation of thoughts, findings, recommendations and questions from result of investigation.

Additional Information / Knowledge Required:

Prior/further knowledge required in following software: Microsoft Visual Studio, R Studio (especially packages such as ggplot2).
Extension of knowledge on the theory behind the knapsack problem, further uses and variations of the problem.
Extension of knowledge on the theory behind the algorithms I have chosen and variations of each (if any).

Information Sources that Provide a Context for the Project:

Hristakeva, M. and Shrestha, D. (2004). *Different Approaches to Solve the 0/1 Knapsack Problem.* Available at: https://pdfs.semanticscholar.org/7647/b5adfd7a326bdc8795a3ab2491700b321ce1.pdf.
Shaheen, A. and Sleit, A. (2016). *Comparing between different approaches to solve the 0/1 Knapsack problem*.
Available at: https://www.researchgate.net/profile/Azzam_Sleit/publication/306021086_Comparing_between_different_approaches_to_solve_the_01_Knapsack_problem/links/57c4486208aee50192e89aef/Comparing-between-different-approaches-to-solve-the-0-1-Knapsack-problem.pdf
Sathyajit, B. P. and Shunmuga Velayutham, C. (2018). *Visual Analysis of Genetic Algorithms While Solving 0-1 Knapsack Problem*
Available at: https://link.springer.com/chapter/10.1007/978-3-319-71767-8_6

The Importance of the Project:

This project is important because there are many real-world applications of the knapsack problem including investment portfolios and raw material cutting, and several ways to solve these problems. The research and experiments I am conducting is important because not only does it compare multiple algorithms in one place, but also it compares a large range of sizes of datasets using graphs plotted in R Studio. This means that it can provide individuals the knowledge to determine which algorithm would be most effective at solving their instance of the knapsack problem, regardless of the size of the problem.
I am aware that this is an active and ongoing area of research, and many previous studies have been done on this problem. After finding many instances of conflicting results from several sources, my study will perform tests on all three of the stated algorithms and I will draw my own set of results from these.

The Key Challenge(s) to be Overcome:

- Ensuring I stay relevant to my topic in research, investigation and development.
- The size of the datasets, ensuring the range of weights and values are appropriate for the problem and that these are processed correctly, including custom datasets input by the user.
- The development of the software- ensuring the algorithms are working exactly as desired and output the same set of measurables.
- The creation of graphs from the results- ensuring these are plotted correctly, represent the data perfectly and can be interpreted easily by both those who have a strong understanding of the problem and those who have a limited understanding of the problem and algorithms.
- Satisfying time constraints and completing the project on time.

Appendix 2 Second Formal Review Output

SOC10101 Honours Project (40 Credits)

Week 9 Report

Student Name: ROSS LANGAN
Supervisor: THOMAS METHVEN
Second Marker: Zakwan Jaroucheh
Date of Meeting: 27/11/19

Can the student provide evidence of attending supervision meetings by means of project diary sheets or other equivalent mechanism? **yes** **no***

If not, please comment on any reasons presented

Please comment on the progress made so far

Is the progress satisfactory? **yes** **no***

Can the student articulate their aims and objectives? **yes** **no***

If yes then please comment on them, otherwise write down your suggestions.

Clear objective and good progress
literature review is still in progress (expected to be finished by this time).

* Please circle one answer; if **no** is circled then this **must** be amplified in the space provided

Does the student have a plan of work? **yes** **no***

If yes then please comment on that plan otherwise write down your suggestions.

Does the student know how they are going to evaluate their work? **yes** **no***

If yes then please comment otherwise write down your suggestions.

The student has to find the matrixs to evaluate the algorithms.

Any other recommendations as to the future direction of the project

Identify the contribution, as exact.

The student has to find a way to evalute his work

Signatures: Supervisor  Second Marker 

Student

The student should submit a copy of this form to Moodle immediately after the review meeting; A copy should also appear as an appendix in the final dissertation.

* Please circle one answer; if **no** is circled then this **must** be amplified in the space provided

Appendix 3 Diary Sheets (or other project management evidence)

02/10/19
Initial meeting with supervisor Dr. Thomas Methven. In the meeting we discussed the project idea, my aims and expectations for the project and some relevant information on how to start the project. We also discussed the use R Studio and the inclusion (or rather exclusion) of a user interface in my software. We also discussed time management and balancing work for this project with work for my other two modules, agreeing that there would be times where coursework from other modules may be my main focus at that time.

I came up with some aims for the next meeting including completing the IPO, signing up for Mendeley and doing some background research in order to find novelties for my project.

09/10/19
Second meeting with supervisor. I presented my IPO which was nearly complete and received feedback on it. Again, I needed to think about the novelty of my project, however after finding conflicting results in several studies and datasets smaller than I have in mind, I think I can use these as my novelties to create a new set of results from the algorithms I make, and I can draw my conclusions from these.

We have set targets for next week's meeting, including submitting my IPO, bringing in a copy of my gantt chart and continuing background reading on the knapsack problem in preparation for starting my literature review.

14/10/19
I have submitted my IPO. I made several changes including a bit on the novelty of the project, as well as including links to relevant articles, including those with conflicting results. Today I plan on refining my gantt chart in preparation for my meeting on Wednesday. My project is set as 'A comparison of algorithms in solving large knapsack problems'

16/10/19
In today's meeting we briefly discussed my IPO again, and I am going to add in a couple of research questions and resubmit it before my second marker has had a look. I will email my second marker as soon as I have resubmitted it to make sure he knows I had uploaded my IPO and hopefully to receive some feedback on it.

Me and Thomas discussed further reading on the Knapsack problem and the kind of studies and articles I should be looking for. I have taken away from this meeting that I need a good range of articles, including some older articles to show how people approached the problem then compared to how our approach has advanced to present.

21/10/19
I have received feedback on my IPO today from my second marker Zakwan Jarouchec. He said that the project looks good and has a clear objective. He

suggested that as well as performance comparison of the algorithms, he would like to see some recommendations of what algorithm to use in specific contexts. I will take this into consideration and try to include this is my project. I also gained that it is important to know exactly how I am evaluating the performance of the algorithms in order to facilitate a comparison between them. For next week I will try and determine this.

23/10/19
Today I missed our meeting because I thought my supervisor was unable to attend, however I was mistaken as it is next week he cannot attend. I will spend the next 2 weeks adding more articles to my collection and start analysing and comparing each of them. I have determined exactly how I wish to evaluate my algorithms as per last weeks aim.

25/10/19
I have been continuing my reading and have found that many of the studies, like I hoped, do not use datasets above 1000 items. This is good for my study as I hope to include datasets up to 10,000 items to discover how the algorithms compare when used on larger datasets and problems.

An interesting thing I have noticed is that a few of the studies I have been reading include time as a measurable (seconds/milliseconds). I will have to research if time is a good measurable to use for this kind of experiment because from my understanding the number of operations is a better measurable to use than time, which can differ depending on the computer used.

I have agreed to email Thomas my Gantt chart (I lost the print-out just before our last meeting), project diary, and I plan to include an update on my progress and a screenshot of some of the articles I have found. At this point I have found 16 relevant articles. I aim to make this 30 by the time I start my literature review.

29/10/19
I am emailing Thomas today with my progress. At this point I have 21 articles/studies in my database. After another few I will start noting the links between each articles. I have already noticed several links between the studies and it has become clear to me how the way we approach the problem has changed in recent times, including the advancement of algorithms we use.

1/11/19
I have started copying all the interesting and relevant points from each study into a document with different sections so that I can compare/contrast them. I have used a similar approach on a previous literature review and so I am confident that this is the way that best works for me.

My plan for this process is to read each paper one by one, starting new headers where needed in my comparison document and copying sections, phrases, interesting points (such as units of measurement etc.) to the document. Once I have read every paper, I should be able to see exactly how they compare and make links between them.

5/11/19

Finally, I have read all 21 studies. I have found another few which I will also read. I have started writing my literature review today, as I think I have a good enough understanding of the existing literature and the context of the problem to start writing.

6/11/19
In today's meeting we discussed starting the literature review and the appropriate headers to use and the layout of the document, including what should be included in each section and why. We also discussed the importance of critically analysing the existing literature rather than just reiterating what each already says.

13/11/19
I brought along with me today a copy of my literature review so far. I have completed the introduction, which sits at around 1 page, and I have started the actual literature review.

Thomas advised me that much of the literature review I have started belongs in the introduction, especially the section on the applications of the knapsack problem in the real world. This is because this section, and others relate more to the background information on the problem itself rather than being a part of the literature. I believe now I have a good understanding of my objective in this review, and from this point I should be able to stick to critically reviewing the studies I have selected and keeping this separate from background and general information about the problem itself.

We also discussed my use of references, which looked like this: [name of author]. This was because I had not numbered each article yet and did this for simplicity. I will fix this for next week's meeting.

20/11/19
After organising the interim meeting for today I was expecting to go over this is my meeting, however due to some confusion about the room we have reorganised the meeting for next week. At the moment my literature review sits at around 2500 words. I am aiming ideally for 4000-5000 words in total, or 7-10 pages.

The confusion about the meeting does give me more time to add to my review and send an updated copy to my second marker before next week.

27/11/19
Today I had my interim meeting. It went poorly for several reasons. Firstly, I don't think I was prepared for the formality of the meeting or the type of questions I was going to be asked. Secondly, I had trouble understanding the meaning of many questions, and although I knew the answer and was able to discuss this after, at the time I had trouble determining what each question meant. Working on answering questions relating to my project is going to be a major aim for me in the next few months.

Ultimately though, I am now aware exactly what I need to be doing, and that I perhaps need to work hard over Christmas to get a good start on my program development before the start of term. Despite a poor performance in the interim meeting, it has really helped me understand the kind of language that is used in relation to my project, and the kind of topics I need to feel very comfortable speaking about. Hopefully with practice I will be much more prepared for the viva voce.

04/12/19
Me and Thomas had our final meeting before Christmas. It was agreed that due to upcoming deadlines and exams we would take a break from meeting until the first week back after holidays. I brought with me and showed all the work I have completed so far, and we discussed that over Christmas I would finish my literature review and make a good start on my programs. For the next week or so though, I will be taking a break from the project in order to revise for exams.

22/12/19
I have decided today to cease work for one week over Christmas, as detailed in the gantt chart. I have made a small start on the development of the dynamic programming algorithm in the last week, and hope to have it finished very soon.

16/01/20
Today is the usual day for our project meeting, however due to changes in the timetable today's meeting will not go ahead, and instead will be rescheduled for another time next week. At this stage I have completed the dynamic programming algorithm. I am now trying to develop the branch and bound algorithm, however I am facing difficulties making the program work.

23/01/20
Meetings have now been rescheduled for Thursday's at 1pm. Today's meeting went very well. Discussed was the literature review, development of the algorithms and the progress of the project as a whole so far. For the next few weeks it is key to complete the development of the algorithms and start thinking about other parts of the project. We have also discussed a technical review and the methodology sections.

30/1/20
Today's meeting included further discussion on the contents of a technical review and methodology section, as well as an update on the progress of the algorithms. The aims for next week are to continue finish the technical review and try to complete the branch and bound and genetic algorithms.

13/02/20
Today me and Thomas discussed the technical review, which I have now completed a first draft of. I will email him the first draft for feedback.

The main topic of today's meeting is the genetic algorithm, which is proving to have a very poor performance for most problem instances. We have discussed ways in which I can fix the algorithm, including seeking advice form Prof. Hart if needed. For next week the aims are to continue with the development of the genetic algorithm and start the methodology if possible.

27/02/20
This week I have brought all the material I have, including the literature and technical reviews, and the methodology I have written. Thomas and I discussed some of the documents, and about the next steps including finishing the testing stages and what kind of results will be expected. We also discussed the use of MATLAB to graph the results, rather than R studio as planned. For next week, the aim is that testing should be completed.

05/03/20
Testing is very close to being complete as of this meeting. The genetic algorithm is proving to take extremely long, as so I may need to think of an alternative plan such as imposing a 10-minute time limit on the runtime. I brought with me an A3 copy of most of the results, as we discussed what kind of trends are obvious, and how these might look in graph form. For next week I must finish the genetic algorithm testing and start graphing the results. Next week's meeting is cancelled, giving me plenty time to graph the results and continue with the implementation section of the project.

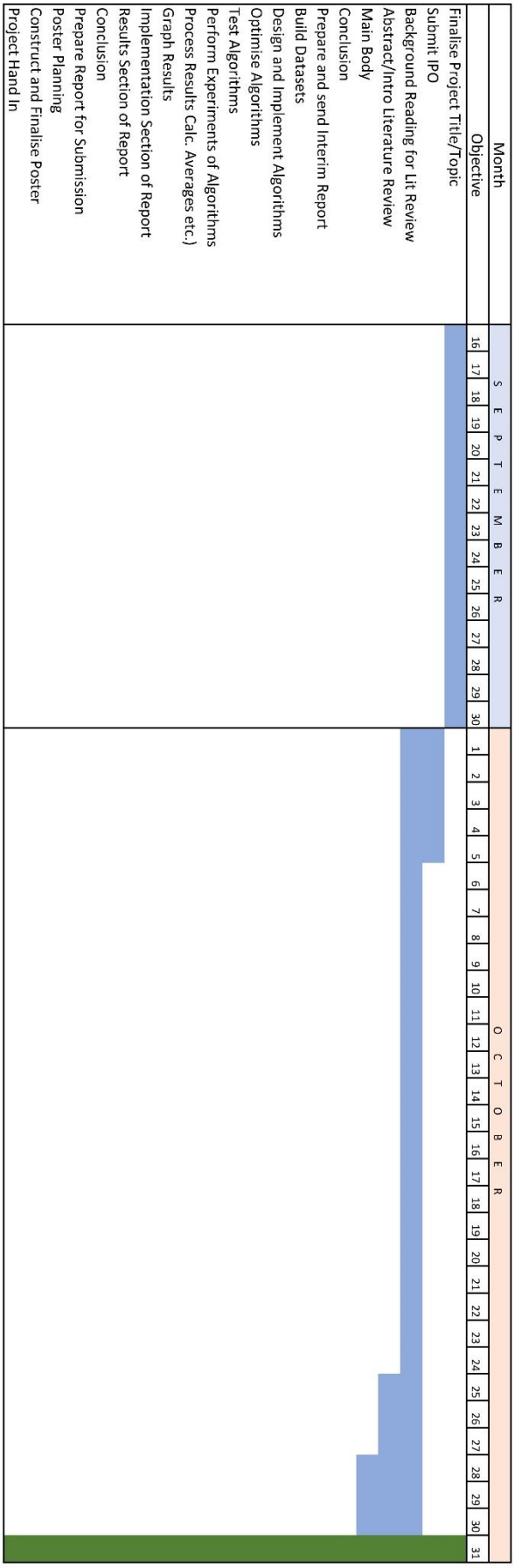
16/03/20
As of today, the spread of the COVID-19 virus means that no face-to-face meetings can be held. Through email me and Thomas have organised to continue online meetings through Skype. Our meetings will be scheduled for the same time- Thursdays at 13:00h.

19/03/20
Today was the first Skype meeting. We discussed the progress of the project, including the results and how the document is shaping up. At this stage much of the project has been completed. I have made a very good start on the results section of the document, including graphing the results in MATLAB. I expect that within 3 days I will have completed the results section and be ready to write the conclusion.

26/03/2020
Today me and Thomas discussed the effects of the Covid-19 outbreak on the project, and the extension that has been granted until the 8th of April.
The meeting was primarily focussed on the finishing touches of the project, and I presented several questions regarding small parts of the report, such as including the parameter tuning for the genetic algorithm and including links to webpages.
At this stage, the Covid-19 outbreak means that I am unable to use any of the university's facilities, which has greatly affected the rate at which work is being completed. I expect to have the project completed to a very high standard by the 8th of April.

02/03/2020
Today was the final meeting between me and Thomas. The majority of this meeting was spend discussing very small details of the project. Over the last 3 or 4 meetings I have come with several prepared questions relating to small aspects of the project. This week the questions related to what to include in the appendixes, and the word count of the project. At this stage the project is very close to being completed, and it is just final touches which are being added.

Appendix 3.1 Gantt Chart

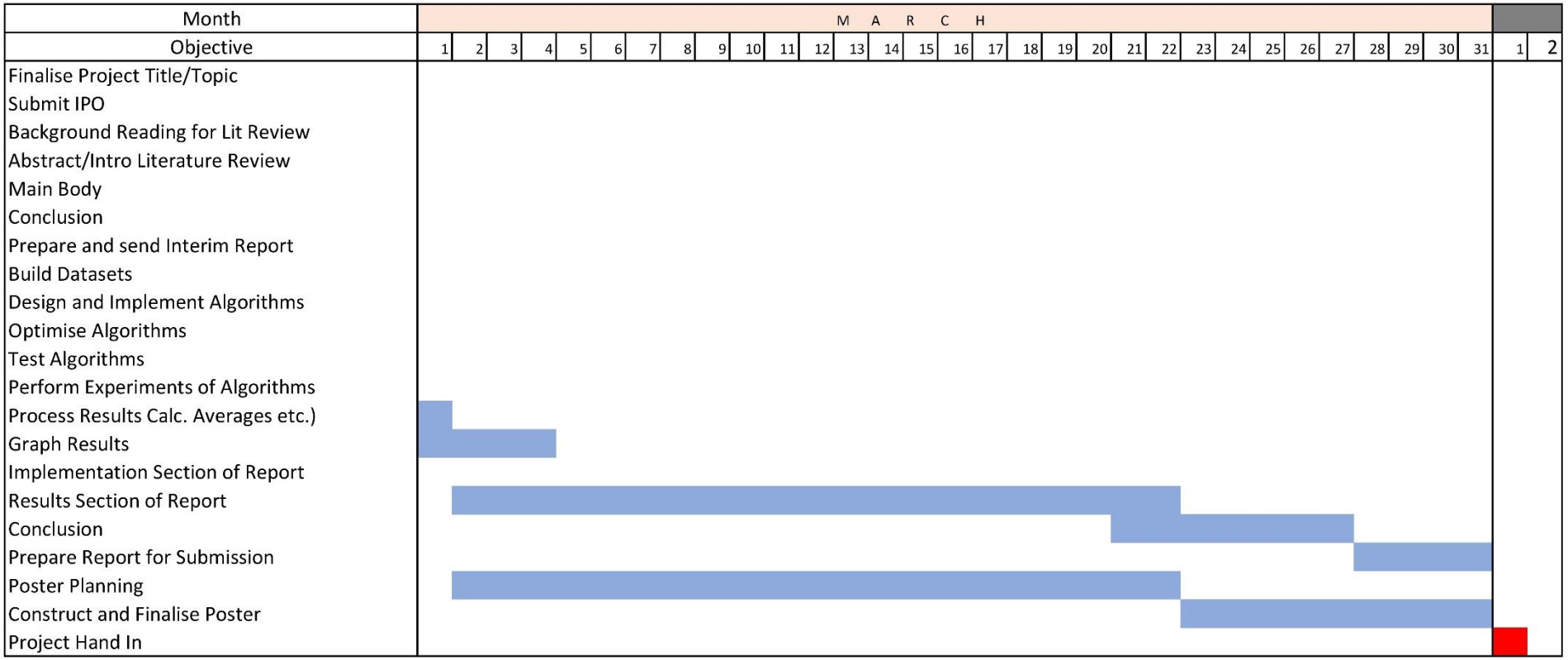


Month	N O V E M B E R																												
Objective	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
Finalise Project Title/Topic																													
Submit IPO																													
Background Reading for Lit Review																													
Abstract/Intro Literature Review																													
Main Body																													
Conclusion																													
Prepare and send Interim Report	H																												
Build Datasets	O																												
Design and Implement Algorithms	L																												
Optimise Algorithms	I																												
Test Algorithms	D																												
Perform Experiments of Algorithms	A																												
Process Results Calc. Averages etc.)	Y																												
Graph Results	S																												
Implementation Section of Report																													
Results Section of Report																													
Conclusion																													
Prepare Report for Submission																													
Poster Planning																													
Construct and Finalise Poster																													
Project Hand In																													

Month	DECEMBER																														
Objective	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Finalise Project Title/Topic																															
Submit IPO																															
Background Reading for Lit Review																															
Abstract/Intro Literature Review			C																												
Main Body			O											E																	
Conclusion			U											X		F															
Prepare and send Interim Report			R											A		O									H						
Build Datasets			S	H										M		R									O						
Design and Implement Algorithms			E	A																					L						
Optimise Algorithms			W	N										R		E									I						
Test Algorithms			O	D										E		X									D						
Perform Experiments of Algorithms			R	-										V		A									A						
Process Results Calc. Averages etc.)			K	I										I		M									Y						
Graph Results				N										S		S									S						
Implementation Section of Report				S										I																	
Results Section of Report														O																	
Conclusion														N																	
Prepare Report for Submission																															
Poster Planning																															
Construct and Finalise Poster																															
Project Hand In																															

Month	J A N U A R Y																												
Objective	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Finalise Project Title/Topic																															
Submit IPO																															
Background Reading for Lit Review																															
Abstract/Intro Literature Review																															
Main Body																															
Conclusion																															
Prepare and send Interim Report																															
Build Datasets																															
Design and Implement Algorithms																															
Optimise Algorithms																															
Test Algorithms																															
Perform Experiments of Algorithms																															
Process Results Calc. Averages etc.)																															
Graph Results																															
Implementation Section of Report																															
Results Section of Report																															
Conclusion																															
Prepare Report for Submission																															
Poster Planning																															
Construct and Finalise Poster																															
Project Hand In																															

Month	FEBRUARY																										
Objective	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
Finalise Project Title/Topic																													
Submit IPO																													
Background Reading for Lit Review																													
Abstract/Intro Literature Review																													
Main Body																													
Conclusion																													
Prepare and send Interim Report																													
Build Datasets																													
Design and Implement Algorithms																													
Optimise Algorithms																													
Test Algorithms																													
Perform Experiments of Algorithms																													
Process Results Calc. Averages etc.)																													
Graph Results																													
Implementation Section of Report																													
Results Section of Report																													
Conclusion																													
Prepare Report for Submission																													
Poster Planning																													
Construct and Finalise Poster																													
Project Hand In																													


Month
M A R C H
Objective
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 1 2
Finalise Project Title/Topic
Submit IPO
Background Reading for Lit Review
Abstract/Intro Literature Review
Main Body
Conclusion
Prepare and send Interim Report
Build Datasets
Design and Implement Algorithms
Optimise Algorithms
Test Algorithms
Perform Experiments of Algorithms
Process Results Calc. Averages etc.)
Graph Results
Implementation Section of Report
Results Section of Report
Conclusion
Prepare Report for Submission
Poster Planning
Construct and Finalise Poster
Project Hand In

Appendix 4.1 Dynamic Programming Results[4]

Coefficients	Capacity	Number of Items	Dataset	DYNAMIC Profit	Weight	Wall Clock Time	Memory
		50		316	49	0.8	2048
		100		467	47	0.9	2048
		500		1102	50	3.23	3072
		750		1244	50	4.43	3447.47
100	50	1000	Dataset 1	1661	50	4.6	4096
		5000		3061	50	7.53	15257.6
		10000		4074	50	11.17	26965.33
		50000		4820	50	27.7	21613.03
		100000		4916	50	44.53	143401.93
		50		651	100	1.17	2048
		100		969	95	1.83	2048
		500		1859	100	5.2	3072
		750		2330	100	6.17	4096
100	100	1000	Dataset 2	2617	100	6.23	4096
		5000		4493	100	12.2	16179.2
		10000		6105	100	17.77	28910.93
		50000		8826	100	36.83	32225.57
		100000		9546	100	67.87	164826.57
		50		1104	497	4.53	2048
		100		1625	498	4.83	2048
		500		4382	500	8.4	4096
		750		5247	500	10.37	5120
100	500	1000	Dataset 3	6031	500	12.93	6144
		5000		12052	500	35.43	23552
		10000		15794	500	44.33	44987.73
		50000		29917	500	115.9	109941.27
		100000		37129	500	272.27	321338.83
		50		1743	1000	4.8	2048
		100		2639	1000	5.13	2048
		500		5896	1000	11.97	5120
		750		7217	1000	16.6	6144
100	1000	1000	Dataset 4	8178	1000	19.6	8192
		5000		16259	1000	46.9	33792
		10000		22977	1000	67.83	64512
		50000		45973	1000	229.63	207976.37
		100000		60197	1000	496.53	515615.77
		50		2367	2497	5.8	3072
		100		5090	4993	10.17	4096
		500		12750	5000	30.97	11264
		750		15357	5000	39.03	18432
100	5000	1000	Dataset 5	17903	5000	43.37	22528
		5000		38642	5000	153.93	99454.1
		10000		55162	5000	251.73	198025.37
		50000		118033	5000	1101.47	995174.17
		100000		160200	5000	2443.97	1972429.57
		50		2597	2570	8.67	3072
		100		4986	5382	15.33	5120
		500		18149	10000	44.63	22528
		750		22588	10000	55.83	32768
100	10000	1000	Dataset 6	25900	10000	66.13	43008
		5000		57029	10000	256.87	197786.93
		10000		81193	10000	450.7	395222.93
		50000		171520	10000	2437.63	1971703.2
		100000		238999	10000	4497.77	3942588.63
		50		2377	2267	46.83	11264
		100		4976	5138	55.37	21504
		500		24587	24612	162.67	101994.6
		750		36999	36423	203.5	153659.33
100	50000	1000	Dataset 7	49292	49082	260.87	205400.57
		5000		129491	50000	1098.2	1024608.73
		10000		185156	50000	2194.13	2049042.67
		50000					
		100000		JAVA	HEAP	SPACE	ERROR
		50		2703	2632	55.27	26624
		100		5050	5278	76.7	52224
		500		25046	25052	236.97	255471
		750		37069	37922	339.5	383471
100	100000	1000	Dataset 8	49288	50272	443.07	511627
		5000		178208	100000	2126.67	2561793.67
		10000					
		50000		JAVA	HEAP	SPACE	ERROR
		100000					

[4] green highlight = optimum value, red highlight = problem unsolved

Appendix 4.2 Genetic Algorithm Results

Coefficients	Capacity	Number of Items	Dataset	GENETIC Profit	Weight	Wall Clock Time	Memory	Generations
		50		314	45	1541.1	97541.87	4400
		100		467	47	1921.13	17588.13	4000
		500		1005	50	9290	104293.8	7400
		750		1111	50	14746.2	110885.8	8467
100	50	1000	Dataset 1	1434	50	22522.83	150917.13	10200
		5000		2057	50	162190.9	595865.43	17000
		10000		2819	50	479706.23	861887.93	25698
		50000		1820	50	600077	602892.1	6828
		100000		1514	49	600070.97	218448.07	3381
		50		633	96	1414.7	129589.6	4400
		100		969	95	1707.03	60323.43	4000
		500		1761	100	9929	113214.73	8667
		750		2174	100	20306.13	136355.3	13000
100	100	1000	Dataset 2	2400	100	29939.63	126497.07	14533
		5000		3329	100	241149.57	467875.9	25800
		10000		4151	100	513205.23	822985	28415
		50000		2648	99	600061.9	1001288.17	6811
		100000		2229	99	600098.47	244497.87	3417
		50		1017	496	2206.27	64205.67	7267
		100		1502	498	3301.57	97050.67	8333
		500		4373	499	4996.97	138250.27	4400
		750		5080	500	15322.17	130850.07	9667
100	500	1000	Dataset 3	5757	500	30971.43	146063.37	15200
		5000		9902	500	422700.47	520591.53	43295
		10000		10567	498	600015.33	798028.97	31001
		50000		6954	498	600056	625834.93	6460
		100000		6145	498	600136.03	400678.23	3345
		50		1694	996	3030.73	70833.87	9600
		100		2513	998	4403.1	78930.03	10800
		500		5675	999	12987.7	115977.53	11333
		750		7152	1000	8526.27	84548.03	5267
100	1000	1000	Dataset 4	8060	999	10083.43	120578.2	4733
		5000		14031	999	576880.97	729238.7	55370
		10000		11487	999	520949.4	574280.13	29243
		50000		11258	999	600044.23	658073.43	9573
		100000		10537	998	600096.53	2128280.77	4615
		50		2367	2497	3700.63	81593.4	9533
		100		5063	4995	6870.97	79629.63	12933
		500		12415	4998	35808.33	79195.67	27933
		750		14892	5000	62321.37	95353.83	36533
100	5000	1000	Dataset 5	17396	4999	71758.03	150180.73	33067
		5000		35777	4999	582529.27	744654.3	57389
		10000		39041	4998	600021.2	865900.13	28392
		50000		35022	4997	600076.7	1224830.67	6639
		100000		34197	4997	600123.8	2346238.73	4004
		50		2597	2570	2354.3	108161.33	8533
		100		4986	5382	4527.97	81056.17	11600
		500		17853	9998	32798.67	74124.43	33733
		750		22108	9999	53861.9	107260.67	41867
100	10000	1000	Dataset 6	25369	9999	81792.83	121501.6	52400
		5000		56737	10000	41795.63	543233	5400
		10000		69437	9998	600021.67	631552.23	36713
		50000		65459	9996	600080.57	1108613.67	6605
		100000		64771	9996	600129.57	2269690.73	3608
		50		2377	2267	3390.13	87825.97	9733
		100		4976	5132	6637.50	41793.63	13600
		500		24587	24612	50812.50	402122.57	35200
		750		36999	36415	98295.66	696123.69	46552
100	50000	1000	Dataset 7	49291	49065	154983.60	1107180.37	57800
		5000		111882	49991	600024.80	871123.40	45654
		10000		153914	49989	600036.30	776131.03	36256
		50000		320275	49997	595260.43	2014913.63	5325
		100000		303453	49997	600171.93	2209774.40	2904
		50		2703	2632	2266.8	94270.13	8600
		100		5050	5278	4500.7	89715.6	12200
		500		25045	25044	31548	76114.77	28467
		750		37069	37910	61060.33	94412.17	38467
100	100000	1000	Dataset 8	49287	50256	100088.37	115806.2	48133
		5000		144381	99966	600030.4	384185.53	52334
		10000		167560	79438	600047.9	726469.13	20115
		50000		556259	99993	600194.57	676194.1	3327
		100000		593740	99996	600235.73	2109715.53	2331

Appendix 4.3 Branch and Bound Results

Coefficients	Capacity	Number of Items	Dataset	BRANCH Profit	Weight	Wall Clock Time	Memory
100	50	50	Dataset 1	316	49	11.2	89866.8
		100		467	47	21.63	90316.43
		500		1102	50	954.87	43981.43
		750		1244	50	10875.7	124054.53
		1000		NO SOLUTION FOUND			
		5000					
		10000					
		50000					
		100000					
100	100	50	Dataset 2	504	99	15.9	90651.73
		100		689	100	34.3	92324.3
		500		1627	100	4590.7	57016.6
		750		1794	100	77146.1	110251.27
		1000		NO SOLUTION FOUND			
		5000					
		10000					
		50000					
		100000					
100	500	50	Dataset 3	1056	499	61.23	97205.77
		100		1619	499	502.4	48361.87
		500		NO SOLUTION FOUND			
		750					
		1000					
		5000					
		10000					
		50000					
		100000					
100	1000	50	Dataset 4	1689	999	121.2	107416.43
		100		2366	1000	3598.8	23336.03
		500		NO SOLUTION FOUND			
		750					
		1000					
		5000					
		10000					
		50000					
		100000					
100	5000	50	Dataset 5	NO SOLUTION FOUND			
		100					
		500					
		750					
		1000					
		5000					
		10000					
		50000					
		100000					
100	10000	50	Dataset 6	NO SOLUTION FOUND			
		100					
		500					
		750					
		1000					
		5000					
		10000					
		50000					
		100000					
100	50000	50	Dataset 7	NO SOLUTION FOUND			
		100					
		500					
		750					
		1000					
		5000					
		10000					
		50000					
		100000					
100	100000	50	Dataset 8	NO SOLUTION FOUND			
		100					
		500					
		750					
		1000					
		5000					
		10000					
		50000					
		100000					

Appendix 4.4 Greedy Approximation Algorithm Results

Coefficients	Capacity	Number of Items	Dataset	GREEDY Profit	Weight	Wall Clock Time	Memory
		50		311	45	48.5	2048
		100		467	47	50.83	2048
		500		1083	49	66.43	3072
		750		1244	50	71.1	4096
100	50	1000	Dataset 1	1661	50	79.1	4096
		5000		3061	50	0	14336
		10000		4074	50	143.83	3035.53
		50000		4820	50	292.73	94362.5
		100000		4916	50	435.57	53498.83
		50		621	76	47.1	2048
		100		969	95	52.37	2048
		500		1859	100	68.17	3072
		750		2330	100	71.93	4096
100	100	1000	Dataset 2	2614	100	78.17	4096
		5000		4493	100	122.13	14336
		10000		6104	100	151.43	3001.03
		50000		8826	100	285.87	94657.57
		100000		9546	100	425.83	53873.1
		50		1092	484	47.27	2048
		100		1618	495	50.57	2048
		500		4370	496	66.63	3072
		750		5234	499	74.73	4096
100	500	1000	Dataset 3	6026	499	82.37	4096
		5000		12051	500	122.1	14336
		10000		15793	500	152.77	3070.07
		50000		29917	500	285.43	94991.03
		100000		37129	500	432.3	53430.1
		50		1742	998	48.13	2048
		100		2623	990	50.87	2048
		500		5887	998	67	3072
		750		7211	1000	74.37	4096
100	1000	1000	Dataset 4	8175	1000	79.2	4096
		5000		16259	1000	123.3	14336
		10000		22977	1000	155.57	3071
		50000		45973	1000	313	94716.33
		100000		60197	1000	497.37	53781.23
		50		2367	2497	50.63	2048
		100		5090	4993	54.47	2048
		500		12745	4999	78.4	3072
		750		15357	5000	73.93	4096
100	5000	1000	Dataset 5	17903	5000	84.1	4096
		5000		38641	5000	139.7	14336
		10000		55162	5000	155.43	3002.47
		50000		118033	5000	317.67	94581.8
		100000		160200	5000	463.7	54428.27
		50		2597	2570	49.4	2048
		100		4986	5382	53.27	2048
		500		18149	10000	69.93	3072
		750		22588	10000	77.8	4096
100	10000	1000	Dataset 6	25900	10000	86.3	5120
		5000		57029	10000	134.13	14609.07
		10000		81193	10000	168.6	3001.5
		50000		171520	10000	333.17	94650.4
		100000		238999	10000	483.9	53691.5
		50		2377	2267	49.37	2048
		100		4976	5138	53.87	2048
		500		24587	24612	73.77	3072
		750		36999	36423	81.33	4096
100	50000	1000	Dataset 7	49292	49082	88.27	5120
		5000		129490	50000	159.07	14677.33
		10000		185156	50000	176	3070.07
		50000		401745	50000	391.87	94742
		100000		557726	50000	702.27	54733.17
		50		2703	2632	56.47	2048
		100		5050	5278	75	2048
		500		25046	25052	87.63	3072
		750		37069	37922	91.23	4096
100	100000	1000	Dataset 8	49288	50272	106.87	5120
		5000		178208	100000	163.7	14950.4
		10000		255192	100000	192.8	3069.93
		50000		568294	100000	460.77	94915.83
		100000		794360	100000	956.33	54549.57

Appendix 5

DynamicProg - DynamicProgramming/src/Main.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
DynamicProgramming
*Main.java
File file = new File("D:\\", "DynamicProgrammingResult.txt");
if (!file.isFile() && !file.createNewFile())
{
throw new IOException("Error creating new file: " + file.getAbsolutePath());
}
BufferedReader reader = new BufferedReader(new FileReader(file));
int lines = 0;
while (reader.readLine() != null) lines++;
reader.close();
int testCase = 71;
System.out.println(testCase);
System.exit(0);
switch(testCase) {
case 0: Constants.setDataSize(50); Constants.setKSCapacity(50); Constants.setName("D:\\test1.in");
break;
case 1: Constants.setDataSize(100); Constants.setKSCapacity(50); Constants.setName("D:\\test1.in");
break;
case 2: Constants.setDataSize(500); Constants.setKSCapacity(50); Constants.setName("D:\\test1.in");
break;
case 3: Constants.setDataSize(750); Constants.setKSCapacity(50); Constants.setName("D:\\test1.in");
break;
case 4: Constants.setDataSize(1000); Constants.setKSCapacity(50); Constants.setName("D:\\test1.in");
break;
case 5: Constants.setDataSize(5000); Constants.setKSCapacity(50); Constants.setName("D:\\test1.in");
break;
case 6: Constants.setDataSize(10000); Constants.setKSCapacity(50); Constants.setName("D:\\test1.in");
break;
case 7: Constants.setDataSize(50000); Constants.setKSCapacity(50); Constants.setName("D:\\test1.in");
break;
Problems
Javadoc
Declaration
Console
<terminated> Main [Java Application] C:\Program Files\Java\jdk-12.0.2\bin\javaw.exe (6 Apr 2020, 19:58:45)
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
at Main.knapsackDP(Main.java:20)
at Main.main(Main.java:264)
Outline
Main
profit : int
weightK : int
knapsackDP(int[], int[], int, int) : void
main(String[]) : void
Task List
Find
All
Activate...
Quick Access
Writable
Smart Insert
71 : 64 : 1776