Investigation into Knapsack

Wesley Kerr
Department of Computer Science
University of Wyoming
Laramie, WY 82071
wkerr@cs.uwyo.edu

Abstract

The 0/1 Knapsack problem has been a highly researched problem in computer science dating back to the 1970's when it was first proven to be a NP-complete problem. There have been many different algorithms developed that can solve the knapsack problem, and all of them have different benefits and drawbacks associated with them. We intend to study five different algorithms that can be used to solve the 0/1 Knapsack problem and determine what types of datasets these algorithms will be successful and when they will fail.
Keywords: *Knapsack, Algorithms, Dynamic Programming, Genetic Algorithms*

1 Introduction

Consider the following problem:
A bank robber is inside the vault of a bank. He only has one bag and there is more loot in the vault than can fit into his bag. Each object has a different weight and a different value on the black market. The bag can be filled up to a certain weight without going over, otherwise, the thief will loose everything out the bottom of his bag when it rips. How can he choose objects to put into the bag so that he can maximize his profit?

The problem illustrated above is an example of the 0/1 Knapsack optimization problem. We will be studying this problem from an artificial intelligence and computational complexity perspective in more detail in the following sections, but we assume the reader is familiar with NP-completeness and the analysis of algorithms.

The 0/1 Knapsack decision problem is defined as a set $S = a_1, .., a_n$ of n items with specified weights and values. Let $v(a_i)$ be the value of the a_i object and let $w(a_i)$ be the weight of the a_i object. In the classical integer 0/1 Knapsack problem, $v(a_i) \in \mathbf{Z}^+$ and $w(a_i) \in \mathbf{Z}^+$. The knapsack also contains a maximum capacity W, and for the decision 0/1 Knapsack problem a desired value P.

The task of the 0/1 Knapsack decision problem is to determine if there exists a subset S' of S such that

$$\sum_{i=1}^{n} w(a_i) \leq W \quad \text{and} \quad \sum_{i=1}^{n} v(a_i) \geq P$$

By using this definition of the 0/1 Knapsack problem we are able to prove the NP-completeness of the problem. The 0/1 Knapsack decision problem has been proven to be NP-complete by [3] as well as other authors. Garey and Johnson [3] have shown that if we are able to prove that the decision statement of the problem is NP-complete, the corresponding optimization problem is also known to be NP-hard.

Since typically we are searching for the best possible solution without prior knowledge, we define the 0/1 Knapsack optimization problem as find a subset S' of S such that

$$\sum_{i=1}^{n} w(a_i) \leq W \quad \text{and} \quad \sum_{i=1}^{n} v(a_i) \text{ is maximized}$$

Because of the proofs showing that the optimization form of a NP-complete decision problem is still NP-hard, we know there currently is not polynomial algorithm that is guaranteed to yield the optimal solution to the 0/1 Knapsack optimization problem. If one existed we could use the solution generated by the algorithm to solve the decision problem in polynomial time and we would know that NP=P (currently still an open problem in complexity theory).

Operations researchers have spent a lot of time looking into the 0/1 Knapsack problem for its uses in many industries. For example, manufacturing companies need to maximize the profit of each individual truck sent to the distributor, so they can maximize their profit. Therefore, the knapsack problem can be viewed from the context of manufacturing systems and also from artificial agents since in the future it could be artificial agents loading the trucks.

In this paper we are going to explore different techniques for solving the 0/1 Knapsack optimization problem. We are going to discuss the completeness of the algorithms to see how good the solution they arrive at is, compared to the optimal solution. Since time and space is a commodity, we are going to also investigate the time and space complexity of the provided algorithms. Continuing our investigations, we plan to determine what types of datasets pose problems for each of the different algorithms. The algorithms we are going to investigate are: branch and bound, greedy search, polynomial-time algorithm using dynamic programming, fully polynomial-time approximation scheme, and finally genetic algorithms. We will show that when time is limited artificial agents can benefit from the proofs showing that the fully polynomial approximation scheme can guarantee success.

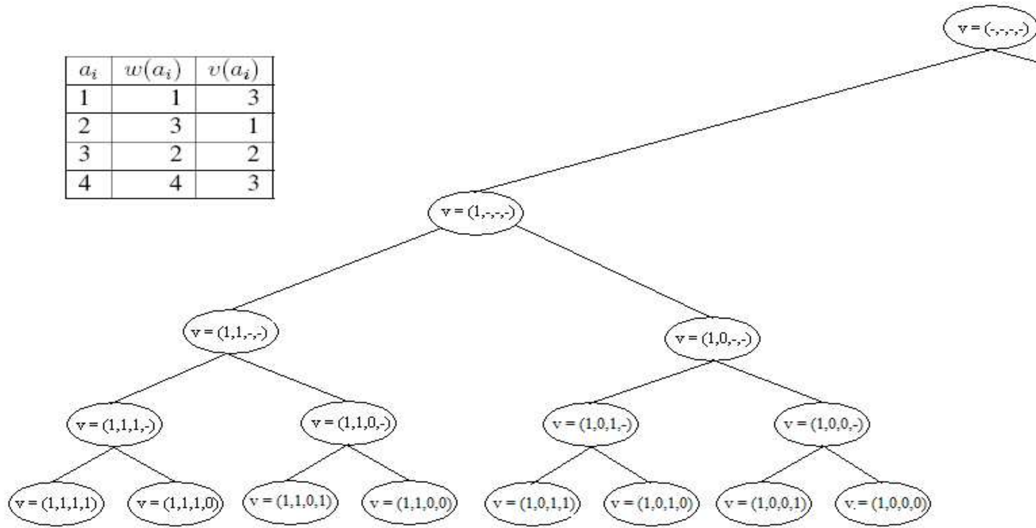
2 Branch and Bound

Branch and bound is a technique used to search the set of all possible candidate solutions to find the optimal solution. Since for each $s \in S$, we can either choose to take or not take the s we have 2 choices for all objects. There are n objects in the set S, therefore we know that there are 2^n possible combinations of solutions.

The algorithm sets up the search through all possible combinations as follows. We maintain one vector of bits corresponding to a possible solution. When the algorithm decides to take an object i (place it into the knapsack) we assign a 1 to the ith position in the vector, otherwise we assign a 0 to the ith position in the vector. The vector can be filled in order starting with the first element. So, for the first element we can either decide to take it or not. If we decide to add the element the vector looks like (1,-,-,...,-), we continue choosing the next elements. Once we have filled the vector with a choice for each element we can calculate the total weight of the solution and its total value. We have to maintain the best so far and as we find solutions that are better we replace the best so far with the new solutions. Using recursion we can traverse all the different possible solutions. A partially developed tree for the dataset can be found in figure 1.

This algorithm is *exhaustive* as it needs to search the entire set of possible solutions before it knows that it has found the maximum profit. This leads to a time complexity of O(2^n). The space complexity of this algorithm is O(n) since we only need to store one possible solution at a time. Since the algorithm is exhaustive it is able to guarantee convergence to the optimal solution. So, regardless of the data set the branch and bound algorithm will find the correct solution.

The branch and bound algorithm's complexity can be improved by pruning invalid solutions before exploring them to completion. For example when we have added an object that causes the current weight to go over the maximum weight W, we no longer need to continue exploring that path since the weight will never be less than



a_i	$w(a_i)$	$v(a_i)$
1	1	3
2	3	1
3	2	2
4	4	3

Figure 1. Partially expanded branch and bound search tree.

the max weight. This can rapidly improve the complexity if you consider different datasets. If you can guarantee that your dataset contains objects such that $\forall a_i$,

$$w(a_i) > \frac{W}{2}$$

The complexity of the algorithm becomes polynomial because of pruning, and at most, you can only put one object into the knapsack. Although the dataset is trivial to solve for any algorithm, it provides a means for determining the complexity of branch and bound for a dataset that has specific weights with respect to the maximum weight W. It also sets a precedent for determining the datasets that even with pruning the branch and bound algorithm complexity is still O(2^n), those datasets such that the weights are distributed around $\frac{W}{n}$.

Because of the high complexity, most branch and bound algorithms can only solve the 0/1 Knapsack problem for small datasets of roughly $n = 40$. This limiting factor has led researchers to search for other algorithms that work on larger datasets.

3 Greedy Algorithm

A naive approach to solving the 0/1 Knapsack problems is to take the ratio of profit to weight for each of the n elements in the set S. Define $ratio$ to be the following:

$$\forall i \quad ratio(a_i) = \frac{v(a_i)}{w(a_i)}$$

Once this ratio has been found, the ratios can be sorted from highest to lowest ratio. When the data is sorted, you can then begin taking elements with the highest ratio until you have reached the capacity of the knapsack W [6], [8]. Table 1(a) shows a pre-generated data set to explore the greedy algorithm on. The algorithm's job is to sort the data in decreasing order and this can be seen in Table 1(b). Finally a solution is chosen for a given knapsack capacity, $W = 8$, in Table 1(c). We can see that with this trivial dataset the greedy algorithm does a good job at determining the optimal solution. Yet as we will find out later, the algorithm can only be guaranteed to find a solution that is half of the optimal and can made to perform arbitrarily badly on any given dataset.

Table 1. Greedy Algorithm in action. (a) shows the initial dataset (b) shows the dataset after sorting. (c) shows the final solution for $W = 8$.

(a)				(b)				(c)		
a_i	$w(a_i)$	$v(a_i)$	ratio	a_i	$w(a_i)$	$v(a_i)$	ratio	a_i	$w(a_i)$	$v(a_i)$
1	1	3	3	1	1	3	3	1	1	3
2	3	1	0.33	3	2	2	1	3	2	2
3	2	2	1	4	4	3	0.75	4	4	3
4	4	3	0.75	2	3	1	0.33			

The complexity of this solution is determined by the sorting algorithm used to sort the n ratios. The best known sorting algorithms still have a complexity of o($n \log n$). The space complexity of this solution only requires O(n) space. Because of these complexities we are able to solve knapsack problems with large numbers of objects regardless of special rules for data sets we saw above.

Unfortunately this algorithm is not complete and can be made to perform arbitrarily bad[1]. Consider the following problem instance of S defined for n items. Let $v(a_i) = w(a_i) = 1$ for all $i = 1, ..., n-1$. Let $v(a_n) = W - 1$, and let $w(a_n) = W = kn$ where k is an arbitrarily large number. Let O be the optimal solution and let S' be the solution found by the greedy solution. Then it is easy to see that $v(O) = W - 1$, while the greedy solution is $v(S') = n - 1$. Therefore we can see that $\frac{v(O)}{v(S')} > k$. Upon examining the solution provided we see that the solution found performs badly because it does not consider the element with the highest profit, while the optimal solution is this element. Using this information we can devise a second algorithm that can be proven to come within $\frac{1}{2}$ of the optimal P. Define the second algorithm to take the $max(v(a_i)_{max}, v(S'))$. Using this second algorithm, one can prove that the solution provided S'' is at least $\frac{1}{2}$ of the optimal[1]. Although this gives us a decent approximation of the actual solution, it is far from optimal from the industries standpoint. We would like a solution that has a tighter bound, so we begin investigating dynamic programming.

4 Dynamic Programming

There is another approach besides branch and bound that is complete and has a better time complexity. Dynamic programming techniques have been used to build the best solution as we go without searching through all possible combinations of solutions. This algorithm is also known as a pseudo-polynomial time algorithm for knapsack. This section borrows heavily from [8] and [4].

To start developing the algorithm we need to find the maximum profit possible by placing objects into the knapsack. Let P be the value of the object with the highest value, then nP is a trivial upper bound on the highest value we can hope to achieve if all objects have a value of P. For each $i \in \{1, ..., n\}$ and $p \in \{1, ..., nP\}$ let $S_{i,p}$ be a subset of S such that the profit of $S_{i,p}$ is exactly p and the weight is minimized. We use this information to our advantage to build a $n \times nP$ table T. We know that for each $i \in 1, ..., n$, $T(i, 0)$ is 0 since we can choose not to take any objects. For all other $p \in \{1, ..., nP\}$, $T(i, p)$ is initialized to ∞. We begin filling the other values in T based on the following recursion:

$$T(i+1, p) = \begin{cases} min\{T(i,p), w(a_{i+1}) + T(i, p - v(a_{i+1}))\} & \text{if } v(a_{i+1}) \leq p \\ T(i,p) & \text{otherwise} \end{cases} \quad (1)$$

Once we have completed the building of the table we can begin determining the best value that can be achieved by looking at the nth column. You begin by working back from nP towards 0 looking for the first weight that is

less than the maximum weight W. Once we have found this weight the current index is the maximum value that can be achieved.

Now that we have determined the maximum value that can be achieved we would like to know what objects to place into our knapsack, so we can then begin working our way back through the table. Let j be the maximum value that can be achieved. The following recursion can be used to backtrack through the table to determine the objects (notice the similarities to the original recursion)

$$T(i+1,p) = \begin{cases} true \text{ and } j = j - v(i+1) & T(j - v(i+1), i) = T(j, i+1) - w(i+1) \\ false & \text{otherwise} \end{cases} \quad (2)$$

Figure 2 contains the pseudo-polynomial time algorithm in action and shows how we can generate the final solution from the complete table. The dataset chosen is the same dataset we have chosen to use for the other algorithms.

(a)

a_i	$w(a_i)$	$v(a_i)$
1	1	3
2	3	1
3	2	2
4	4	3

(b)

	a_1	a_2	a_3	a_4
0	0	0	0	0
1	∞	3	3	3
2	∞	∞	2	2
3	1	1	1	1
4	∞	4	4	4
5	∞	∞	3	3
6	∞	∞	6	5
7	∞	∞	∞	8
8	∞	∞	∞	7
9	∞	∞	∞	10
10	∞	∞	∞	∞
11	∞	∞	∞	∞
12	∞	∞	∞	∞

Figure 2. Pseudo-polynomial time algorithm in action. Table (a) is the dataset for the knapsack problem. Table (b) is equivalent to table T upon completion of the algorithm. The grey shaded areas represent the trace back to find the optimal solution.

The given polynomial algorithm is known to be complete and will always find the optimal solution. The complexity of this algorithm is O(n^2P). This is because we must run through n objects and nP possible values. We can tighten this bound down by using the fact that the optimal value achieved can be no greater than adding all the objects to the knapsack and finding the total value.

$$\sum_{i=1}^{n} v(a_i)$$

Note that this upper bound is not truly polynomial, since nP can be exponentially large to n. Therefore we refer to this algorithm as pseudo-polynomial. We will use this to our advantage when developing a fully polynomial approximation algorithm for the knapsack problem.

5 Fully Polynomial Approximation Scheme

If we look closely at the pseudo-polynomial time algorithm, we can see that if all of the values were small bounded by a polynomial in n then we could say that the pseudo-polynomial time algorithm would in fact be a fully polynomial time algorithm. We use precisely this notion when developing an approximation algorithm for solving the knapsack problem.

The fully polynomial approximation algorithm uses the dynamic programming algorithm, but first it scales all of the profits by some K. The precise value of K is determined by some ε, the error parameter. Given an $\varepsilon > 0$, we define $K = \frac{\varepsilon P}{n}$. For each a_i in S we define a new $v'(a_i) = \lfloor \frac{v(a_i)}{K} \rfloor$. We then run the dynamic programming algorithm using the new values for each object and output the solution that it finds.

This algorithm is known not to be complete, as it is an approximation algorithm. Let S' be the solution produced by the approximation algorithm. Although, we will not output the optimal solution we can prove that the solution produced by the algorithm has a value at least $v(S') \geq (1 - \varepsilon) \cdot OPT$.

Proof 1 *Let O denote the optimal set that can be found for the set S. For each element a_i in O, the value $v'(a_i) \cdot K$ cannot be greater than $v(a_i)$, since we are always rounding down. Therefore it follows that $v(O) - K \cdot v'(O) \leq nK$. The dynamic programming algorithm returns the optimal solution under the new requirements. Therefore*

$$v(S') \geq K \cdot v'(O) \geq v(O) - nK = OPT - \varepsilon P \geq (1 - \varepsilon) \cdot OPT \tag{3}$$

The last conclusion can be drawn because we know that OPT $\geq P$. [8]

We have shown that the solution that we arrive at from the fully polynomial algorithm is at least $(1 - \varepsilon) \cdot$ OPT, all that is left is to show that this algorithm is polynomial.

Proof 2 *The running time of the algorithm is known to be $O(n^2 \lfloor \frac{P}{K} \rfloor) = O(n^2 \lfloor \frac{nP}{\varepsilon P} \rfloor = O(n^2 \lfloor \frac{n}{\varepsilon} \rfloor)$. Which is polynomial in n and $\frac{1}{\varepsilon}$. [8]*

The approximation algorithm can be run on large datasets with arbitrary sized profits since we are scaling all profits anyways. This leads us to our final algorithm pulls from evolution and is builds solutions as such.

6 Genetic Algorithms

Genetic algorithms have been used successfully to solve NP-hard problems and therefore are suited to solve the 0/1 Knapsack problem. Genetic algorithms are a programming technique that uses *survival of the fittest* to converge on a solution to a given problem. The idea behind a genetic algorithm is to have an initial population of individuals. The program then calculates the fitness of each individual. The program then allows those that are the most fit to reproduce and recombine themselves with those other fit individuals in the population, and those that are less fit will die off. Also, since mutations occur in nature, individuals in the population are randomly selected for mutation to occur.

In order to create a genetic algorithm we have to answer several questions. Firstly we need to determine how we are going to represent and individual. Next we need to decide how we are going to determine the fitness of an individual of the function. We are also going to need to know what operations we are going to perform on the individual. And finally, we need to decide how we are going to select the individuals that are capable of producing children for the next population.

Although there are many different techniques for writing genetic algorithms to solve different problems, I chose to only implement one type of these for comparison purposes. The individuals in the population are represented as bit strings of length n (where n is the number of objects). The bit string position i is a 1 if we decide to take

Table 2. Time Complexity of algorithms

Algorithm	Time Complexity	Space Complexity
Greedy	O($n \log n$)	O(n)
Branch and Bound	O(2^n)	O(n)
Genetic Algorithm	O($nPOP$)	O($nPOP$)
PTAS	O(n^2P)	O(n^2P)
FPTAS	O($n^2\lfloor \frac{n}{\varepsilon} \rfloor$)	O($n^2\lfloor \frac{n}{\varepsilon} \rfloor$)

object a_i and a 0 if we decide not to take a_i. The fitness of an individual is defined as the value achieved by the objects selected as long as the sum of the weights of the objects taken doesn't exceed the maximum weight W. If the maximum weight has been exceeded then the fitness of the individual is 0.

The genetic algorithm that was implemented uses stochastic universal sampling for new population selection. So, the individuals with the highest fitness will be more likely to have more children than those less fit. Once we have generated our new population of individuals the algorithm uses 1-point crossover to recombine individuals. One point crossover works as follows, a cross-over point is selected randomly in the bit string. Two individuals are randomly selected and all bits past the cross-over point are swapped between individuals. The only other operation performed on the individuals is mutation, and for each individual we randomly select and index and change the bit of that position.

The complexity of genetic algorithms has yet to be determined. It has been proven that with infinite time the genetic algorithm will converge to the correct solution[2]. Unfortunately, we cannot wait that long to find a solution. Because of the time complexity of finding the optimal solution we typically stop the genetic algorithm at some predetermined time. Therefore, the solution provided genetic algorithm serves more as an approximate solution as opposed to a complete solution. The space complexity of the genetic algorithm is determined by the population size POP and is $O(nPOP)$. The population of the genetic algorithm is typically chosen smaller than n, therefore the space complexity remains polynomial. When working with genetic algorithms the solution generated is sometimes the optimal and on other occasions it is not, as nothing can be guaranteed with genetic algorithms.

7 Experimental Methodology

In order to show the limits of the theoretical predictions on current computer hardware we have devised a set of experiments. Each of the algorithms was run once on these devised datasets. As we are looking at the knapsack problem in a practical sense and not a research sense we only allowed the genetic algorithm to run once. In a research environment we would run the genetic algorithm multiple times and find the best that has been achieved and the average value achieved. We also capped the total running time for any algorithm at 300 seconds, so in the case of the branch and bound solution we see that although it is optimal, it is not allowed to do an entire search of the space and has not encountered the optimal solution when we stop searching. We used these experiments hoping to show the bounds of each algorithm.

The timing complexity for each algorithm is listed in Table 2. We used these complexities to design several experiments to determine the upper limits for completion of each algorithm.

Let n be the number of objects available to place into the knapsack and let P be the maximum profit for the set of available objects. We created several data sets varying n and P for each dataset. Table 3 contains the datasets and the index given to each one.

Table 3. Experiments

Dataset Index	n	P
1	20	50
2	20	75
3	20	10,000
4	20	100,000
5	20	500,000
6	20	1,000,000
7	20	2,000,000
8	25	50
9	25	75
10	30	50
11	30	75
12	100	50
13	500	50
14	1,000	50
15	2,000	50
16	5,000	50
17	10,000	50

8 Experimental Results

We ran all of the different algorithms through the seventeen different datasets. For each algorithm, we found the best value that can be achieved for that dataset. Since we know that the branch and bound and pseudo-polynomial time solutions are optimal solutions we used them as the optimal solution for the datasets they were able to solve. When we reached a $n = 5,000$ we saw that neither the pseudo-polynomial solution nor the branch and bound solution was capable of solving the problem, therefore we took the maximum value found by any algorithm as the optimal solution, although we cannot guarantee that it is in fact optimal. Figure 3 contains the percentages for each of the algorithms solutions compared to the optimal solution found for that dataset, and Figure 4 contains the approximate[1] running time for each of algorithms for each dataset.

The greedy algorithm performs really well on the datasets given, even only having a guarantee of 50% of the best value found. We will continue our research to determine exactly what datasets pose problems for the greedy algorithm, but at this time it is believed that while the branch and bound algorithm suffers from $w(a_i) > \frac{W}{2}$ the greedy algorithm thrives on these datasets. More research needs to be done before recommendations can be made about using the greedy algorithm.

We have proved that the branch and bound algorithm is capable of solving datasets of relatively small n values, since it does a search of the entire solution set. The maximum n that the branch and bound solution was able to solve in a reasonable time (300 seconds) was roughly thirty items. It was capable of finding reasonable solutions for larger problems, although not optimal.

The genetic algorithm was able to find solutions for each problem, although the percent of the best solution found deteriorates rapidly as the number of objects available for the knapsack increases. For each dataset the genetic algorithm ran on a population size, POP, of 100 and the number of generations was 1000. This problem can be remedied in several ways. One way to fix this problem is to increase the number of individuals in the

[1]We call this approximate because we are only measuring the time in seconds.

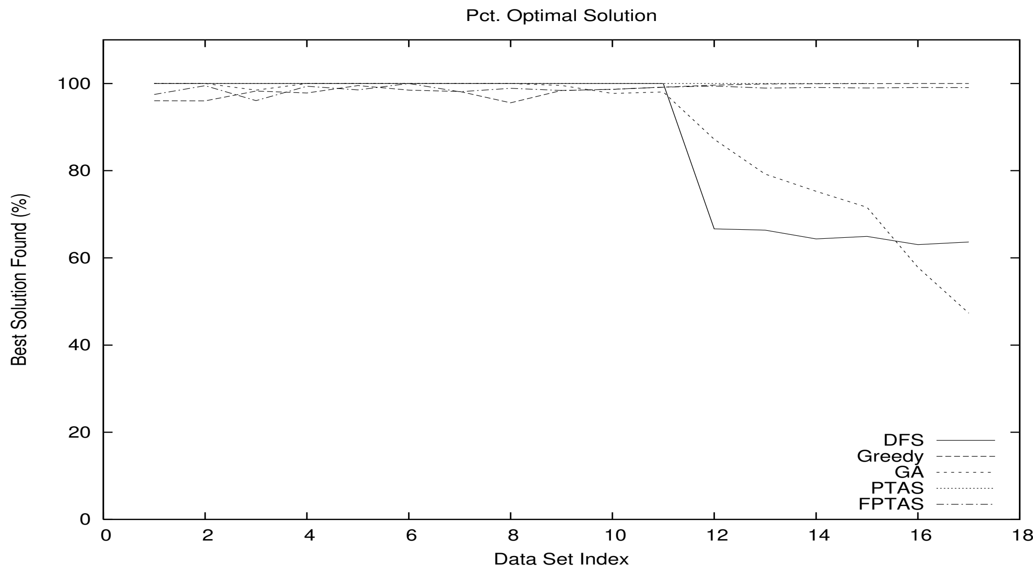


Figure 3. Graph showing the solution found vs. the optimal solution for the given datasets.

population thereby increasing the range that is searched by the genetic algorithm, and the other way to fix the problem is to allow the genetic algorithm to run for more generations. Since this paper is not an investigation in the optimization of a genetic algorithm we will leave that for future research.

The pseudo-polynomial time solution is capable of solving larger datasets than the branch and bound solution, but on really large datasets, even with a small maximum profit, the pseudo-polynomial time solution is unable to solve the problem. The issue here was not a time issue, but instead a space issue. The pseudo-polynomial time solution used more memory than was available to attempt to solve the solution. One remedy to this limitation involves changing the algorithm from storing and entire table to storing just the previous solutions, since they are all that are used to generate the next solutions. The other consideration that has to be taken into account is that along with the current solutions one would also be required to carry along the items in the knapsack that has generated this solution.

Finally, the fully-polynomial time approximation scheme algorithm was able to solve all of the datasets in reasonable time, with very accurate results. These results were generated with a $\varepsilon = 0.5$. Even with this high of an ε we were able to generate results that maintained a percent of the optimal in the 90% range. It appears from the data that the fully-polynomial time approximation scheme is able to generate the best solutions given any of the seventeen datasets. Now that we have our experimental results we are able to generate conclusions based on the figures shown above.

9 Conclusions

There are many different approaches to solving the 0/1 Knapsack problem, even more than the five algorithms presented here. Algorithms have been presented to improve the time complexity of the fully-polynomial approximation scheme as well as guarantee a solution closer to the optimal solution [5], [7]. When choosing an algorithm to solve the 0/1 Knapsack problem, we have shown that there are different factors to consider. If we are dealing with small datasets such that $n \leq 30$, we could use the branch and bound algorithm to find the optimal solution in reasonable time. If our we have a large dataset with small values we could use the polynomial approximation algorithm to find an optimal solution, space permitting. If we have a large dataset and large values for a_i then we are going to need to look into the other three algorithms for determining a solution to the problem. With the genetic

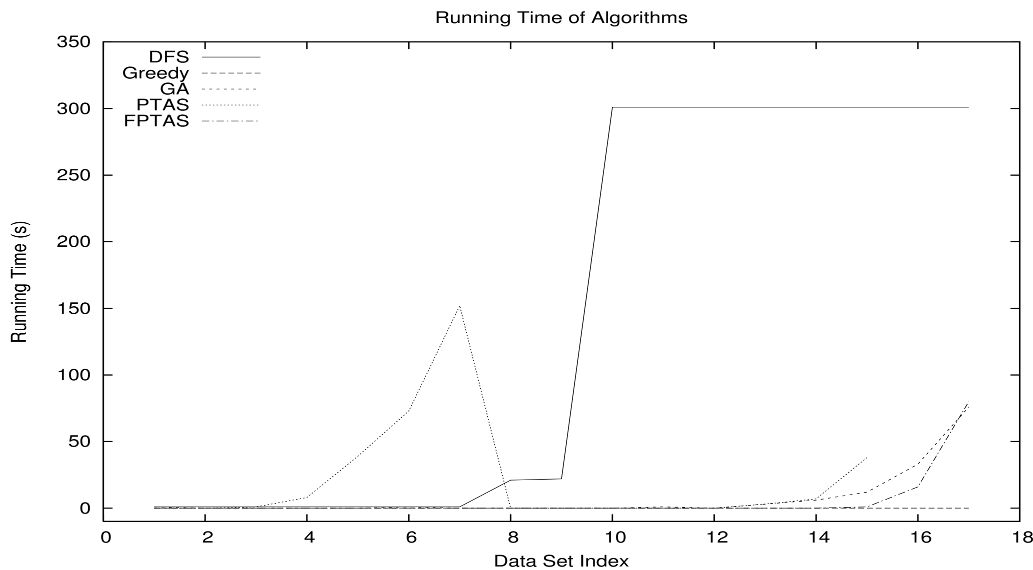


Figure 4. Graph showing the running time for each of the algorithms on each of the datasets.

algorithm we aren't guaranteed anything but we could arrive at the optimal solution, and with the fully-polynomial time algorithm our solution will always be at least as good as $(1 - \varepsilon)\cdot$ OPT. Empirical results have been used to help determine what algorithms will do better when presented with different types of data sets.

References

[1] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation*. Springer, 1999.

[2] T. Back and H. Schwefel. An overview of evolutionary algorithms for parameter optimization. *Evolutionary Computation*, 1(1):1–23, 1993.

[3] M.R. Garey and D.S. Johnson. *Computers and Intractibility: A Guide to the Theory of NP-completeness*. W.H. Freeman and Comp., 1979.

[4] D. Houchbaum. Various notions of approximations: Good, better, best, and more. In D. Houchbaum, editor, *Approximation Algorithms for NP-Hard Problems*, pages 346–398. PWS Publishing Company, 1997.

[5] O.H. Ibarra and C.E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. In *Journal of ACM 22*, pages 463–468, 1975.

[6] M. Lagoudakis. The 0-1 knapsack problem: An introductory survey. Technical report, The Center for And-vanced Computer Studies , University of Southwestern Louisiana, 1996.

[7] E. Lawler. Fast approximation algorithms for knapsack problems. In *Mathematics of Operations Research*, pages 4:339–356, 1979.

[8] V. Vazirani. *Approximation Algorithms*. Springer, second edition, 2000.