

# Benchmarking a Simulated Annealing Approximation Algorithm for the 0-1 Knapsack Problem

Colin Rickert

Department of Computer Science,  
University of Vermont  
Burlington, Vermont, USA  
colinrickert@colinrickert.com

## ABSTRACT

The knapsack problem was analyzed by generating items whose values and costs in the range  $[0,1]$  were drawn from both uniform and beta distributions respectively. A greedy algorithm and a fully polynomial approximation scheme were used for benchmarking against a simulated annealing algorithm that utilizes the greedy algorithm as a pre-processing step.

## General Terms

Algorithms, Measurement, Performance, Design, Experimentation, Theory, Verification.

## Keywords

Combinatorial Optimization, Complexity Theory, Machine Learning.

## 1. INTRODUCTION

A simulated annealing based algorithm for the 0-1 knapsack problem was benchmarked against a greedy algorithm and a fully polynomial approximation scheme. In addition to comparing different algorithmic approaches, the distribution of weights and values in the range  $[0,1]$  were also varied by sampling from either a uniform distribution, a beta( $\alpha=5, \beta=5$ ) distribution or a beta( $\alpha=10, \beta=10$ ) distribution. The beta distribution was chosen primarily because it ranges between zero and one but otherwise is quite similar in shape to a normal distribution provided that  $\alpha=\beta$ . The results of the algorithms were compared by executing them in lisp using items generated from these distributions on knapsack problem sizes ranging from 100 to 1000 in terms of the number of items. The experiment was coded in lisp and the results were gathered in tabular form.

## 2.0 PROBLEM DEFINITION

The 0-1 knapsack problem is known to be NP-Complete and can be outlined as follows:

Given a set of items  $i \in I$ , each of which has a value  $v_i$  and weight  $w_i$  associated with it and a “knapsack” whose maximum weight capacity is  $C$ , find the subset of items  $S'$  that maximizes the net value of these items without violating the weight constraint.

The problem is of great practical value as it can be used to solve a number of real world problems. For instance, consider a ship that has a weight capacity

and whose job is to deliver one set of goods from one port to another. It is in the best interest of the shipping company to ensure that the values of the items on board the ship are maximized without violating the weight constraint of the ship.

There is no known polynomial time algorithm that can solve any NP-complete problem though a relaxed version of the knapsack problem can be solved using a greedy algorithm if fractions of items are allowed<sup>1</sup>. In the fractional valued case, one can simply “stock up” on the item with the greatest value/weight ratio until it is exhausted, then move on to the next valuable item and so forth until the weight capacity is reached which can be done in  $O(n \lg n)$  time.

For the purposes of this experiment, the capacity was always set to be 25% of the size of all of the items in consideration. The reasoning for this is that the average weight of an item would be roughly 0.5 which means that a random problem of size  $N$  would have a total weight of roughly  $0.5N$  on the average. To make the problem maximally hard for a given problem size, one would like the capacity to be violated, on the average, when roughly half of this items are included in the knapsack (i.e. the search space is maximized due to information entropy) and this corresponds to a weight of  $(0.5)(0.5)N = 0.25N$ .

## 2. KNAPSACK APPROXIMATION ALGORITHMS

There are many approaches to solving the knapsack problem including greedy methods, fully polynomial approximation schemes and stochastic methods like simulated annealing and genetic algorithms. These algorithms should be ranked based on their performance on a variety of knapsack problems with variation both in terms of size and the range/distribution of cost/weight values since it is unlikely that there will be a technique that dominates all others on all problem sets.

## 2.1 GREEDY APPROXIMATION TO THE 0-1 KNAPSACK PROBLEM

By far the simplest and most straightforward approximation algorithm for the knapsack problem is a simple greedy strategy where by items are ranked and sorted based on their cost/weight ratios (the higher the better). Items are then taken in order until the capacity has been reached. This approach may seem

naïve but it is in fact surprisingly effective, especially in problems where there is a very large standard deviation of item costs and weights and extreme outliers in the ratios of costs/weights exist as a result. This approach works primarily because it rapidly eliminates items that should not even be considered since their cost/weight ratios are simply too high while always including items that have extremely high cost/weight values. As mentioned, this strategy breaks down as the distribution of cost-weight ratios decreases. However, even though this method is not likely to produce an optimal value, it will often yield a result that is quite close to optimal in most situations. Furthermore, as will be demonstrated, a presorted solution can be “annealed” after sorting to produce even better results than the greedy strategy alone.

## 2.2 FULLY POLYNOMIAL APPROXIMATION TO THE 0-1 KNAPSACK PROBLEM

Dynamic programming can be applied to the integer version of the 0-1 knapsack problem to solve the problem optimally for all integer values. The algorithm works as follows<sup>2</sup>:

1. First represent the knapsack problem as an integer linear programming problem with weight constraint  $C$ :

$$V^* = \text{MAX} \left( \sum_{j=1}^N v_j x_j \right)$$

Subject to  $\sum_{j=1}^N w_j x_j \leq C$  where  $x_j \in \{0,1\}$   
for  $j = 1, \dots, N$

2. Let  $F_j(i)$  represent the smallest knapsack weight that can yield a value of  $i$  involving the variables in the set  $\{1, \dots, j\}$
3. The recursive formula is then:

$$F_j(i) = \min\{F_j(i-v_j) + w_j, F_{j-1}(i)\}$$

4. The table of values is then evaluated in increasing order of objective function values:

$$\begin{aligned} &F_1(1), F_2(1), \dots, F_N(1), \\ &F_1(2), F_2(2), \dots, F_N(2), \\ &\vdots \\ &F_1(i), F_2(i), \dots, F_N(i) \end{aligned}$$

Where  $i$  is the largest knapsack value that can be generated s.t.  $F_N(i) \leq C$ .

Using this method means that there are a total of  $O(NV^*)$  function evaluations<sup>3</sup>. Note that this algorithm can solve the integer version of the knapsack problem optimally however it can still be used to construct a fully polynomial approximation scheme to the real-valued version of the knapsack problem if we scale each coefficient  $v_j$  by the real-valued factor  $k$  and floor the results. In other words  $v_j' = \lfloor v_j/k \rfloor$ . Using this

strategy it can be shown<sup>4</sup> that the running time to solve the scaled problem optimally is  $O(N^2 \lfloor v_{MAX}/k \rfloor)$  where  $v_{MAX} = \max\{v_1, v_2, \dots, v_N\}$ . Unfortunately the running time is not the only constraint one must consider: when using this procedure one must also evaluate memory constraints as the memory required is of  $O(NV_{ALL})$  where

$$V_{ALL} = \sum_{i=1}^N \left( v_i * \text{floor} \left( \frac{v_{\max}}{k} \right) \right)$$

For many practical applications this can be larger than what the main memory constraints of one’s operating system and corresponding hardware will allow.

## 3.3 SIMULATED ANNEALING APPROXIMATION TO THE 0-1 KNAPSACK PROBLEM

Simulated annealing is now a well known technique for combinatorial optimization and has been used to solve a variety of NP-complete problems from Max-Cut to Traveling Salesman. The basic strategy is as follows:

1. For each time step  $t$ , keep track of the total “energy” cost  $E_t$  of the configuration  $c_t$  at time step  $t$  where  $E_t$  is directly proportional to the quality of the solution at that time.
2. Accept a new configuration  $c_{t+1}$ :
  - a. With probability  $e^{-\beta \Delta E}$  if  $\Delta E < 0$
  - b. With probability 1 if  $\Delta E > 0$
where  $\beta = 1/(kt)$  and  $\Delta E = E_{t+1} - E_t$  and  $k$  is a constant
3. Decrease the temperature at each round by some factor  $\epsilon_i$  at each round until the “temperature” is essentially zero.

Outlined below is the proposed version of simulated annealing for the purpose of solving the 0-1 discrete knapsack problem of  $n$  items. The algorithm has the basic features of a general simulated annealing algorithm with the following modifications:

1. The total energy of the system at time step  $t$  is evaluated using a partitioning strategy where by The entire search space is represented as a single list of all the items and a partition point  $j$  that represents the exact position, reading from left to right across the list, where the sum of the weights prior to that point is still less than or equal to the capacity  $C$ :

$$\left( \sum_{i=0}^{i=j} w_i \leq C \right) \wedge \left( \sum_{i=0}^{i=j+1} w_i > C \right) \rightarrow (\text{PARTITION}=j)$$

The formula for energy at time step  $t$  is then

$$E_t = - \sum_{i=0}^{i=j} v_i$$

In other words the energy of the system is equal to the sum of the values of the items up to item  $j$  times negative 1.

2. Upon completing part two above, the algorithm proceeds by randomly choosing items in the list that are left of the partition point (i.e. the “knapsack”) to be swapped with new items not already included in the knapsack with probability  $e^{-\beta \Delta E}$  if  $\Delta E < 0$  and probability 1 if  $\Delta E > 0$ . One item is subject to replacement per time period  $t$ . Once an item has been replaced a new partition point is created using the formula above in the next point in time  $t$  and the energy is recalculated for the new configuration as outlined above for comparison.
3. The temperature starts at 100 degrees and then “cools” by 1/100 of a degree until it reaches absolute zero and no new configurations of higher energy will be accepted. This rate of cooling was found to be satisfactory after some trial and error. Furthermore, experimental trials indicated that  $k=0.0001$  was an optimal value after the results from several trial runs were gathered.

### 2.3 PRE-SORTED SIMULATED ANNEALING

The last strategy used in this experiment was based on pre-sorting the permuted list used in simulated annealing. In this case one starts with the list created by using the greedy strategy and then performs simulated annealing on the presorted list as the initial configuration. As will be shown (see results) the performance of this hybrid algorithm is generally better than either the greedy strategy or unsorted simulated annealing approach. Also, it performed almost as well as the FPAS algorithm with a  $k$  value of 0.01 with the added benefit that simulated annealing does not require the construction of large arrays to be stored in memory and hence can be used on much larger knapsack problems than the standard FPAS algorithm.

### 2.1 STATISTICAL PROPERTIES

One aspect of the knapsack problem that is often overlooked is the distribution of weights and values of the candidate knapsack items themselves. The properties of such distributions can have a significant effect on which strategy to use when optimizing a knapsack problem. If one has some foreknowledge of which distribution these values are being generated from (or a reasonable approximation) they will be able to make a more informed choice about what algorithm to use. At the heart of the matter is the distribution of the cost/weight ratios. This distribution will greatly influence the performance of the greedy approach to the knapsack problem vs. a fully polynomial approximation scheme (FPAS) or Monte Carlo based techniques like simulated annealing.

When the item values and weights are generated from a uniform distribution the result is a very wide

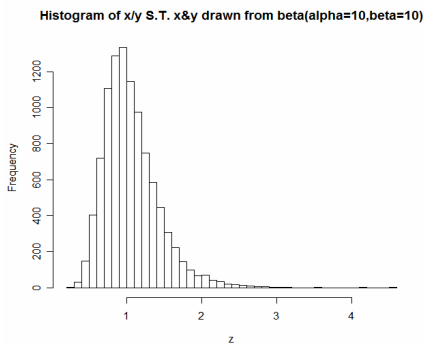
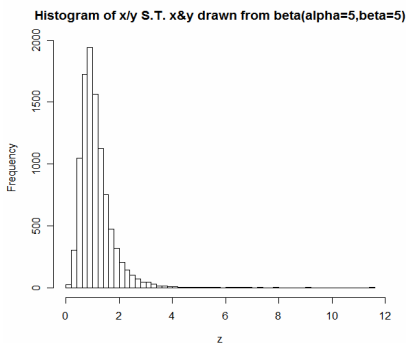
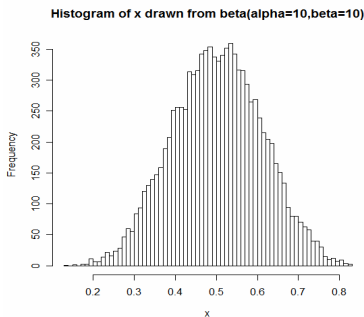
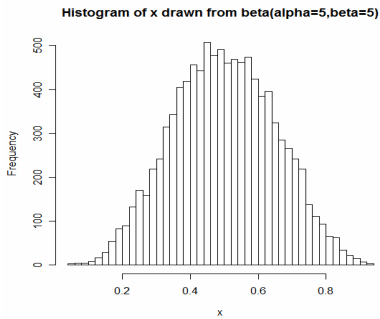
distribution of cost/weight ratios which favors a greedy strategy. For instance, one can generate two vectors of 10000 random numbers between 0 and 1, then divide the resulting vectors to get a new vector equivalent to the set of cost-weight ratios. The resulting vector’s standard deviation will be roughly 595 (quite large) which indicates that several items of extremely low weight and extremely high value (i.e. diamonds if you will) are available. This would be the equivalent of a ship being able to transport everything from diamonds to hay and the low cost items like hay would obviously be excluded. The problem with using this experimental approach (i.e. uniformly distributed weights and values) is that it may not be very realistic. In a real-world scenario both the weights and values will probably be elements of some sort of distribution with standard deviations of cost-weight ratios that are much narrower than in the uniform case.

As the standard deviation of cost-weight ratios shrinks, the greedy strategy becomes less and less optimal and one should consider using more stochastic based methods like simulated annealing (see results). For the purposes of this experiment three different distributions were used for generating the knapsack weights and values. The distributions were 1) uniform in the range [0,1], 2) beta( $\alpha=5, \beta=5$ ), and 3) beta( $\alpha=10, \beta=10$ ). The distributions were used specifically because they generated values between 0 and 1. The probability density function for a beta distribution is:

$$\frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{\alpha-1} (1-x)^{\beta-1}$$

where  $\Gamma(n) = n!$  for all integers  $n$

Keeping the weights and values be constrained between 0 and 1 is desirable from a computational perspective because the quotient of two random numbers between 0 and 1 is very unlikely to result in floating point arithmetic overflow. Allowing the costs and weights to range between the max and min floating point numbers could result in such an overflow should a sufficiently large number greater than 1 be divided by a small number that is less than 1 and greater than 0. Sampling from a beta distribution with  $\alpha=\beta$  will result in a symmetrical distribution of values in the range [0,1] with a shape very similar to a normal distribution and whose standard deviation decreases as  $\alpha$  and  $\beta$  are increased. The following figures show a histogram of vectors whose elements are drawn from beta( $\alpha=5, \beta=5$ ) or beta( $\alpha=10, \beta=10$ ) distributions as well the vectors that represent the quotient of two vectors drawn from either a beta( $\alpha=5, \beta=5$ ) or a beta( $\alpha=10, \beta=10$ ) distribution:



Figures 1-4: Histograms of beta( $\alpha=5,\beta=5$ ) and beta( $\alpha=10,\beta=10$ ) distributions for 10,000 item costs

and weights and the resulting cost/weight ratio distributions.

#### 4.0 RESULTS

3 different experimental trials for each algorithm were coded in lisp and the results were tallied: The greedy method, the fully polynomial approximation scheme (FPAS), simulated annealing and presorted simulated annealing (“sort-anneal”) were all executed for knapsack problems of size ranging from 100 to 600 on uniform, beta(alpha=5,beta=5) and beta(alpha=10,beta=10) distributions. For each of these sizes of knapsack problems, 5 different random knapsack problems (labeled “trial1 through trial5”) were constructed by drawing random samples of item weights and values from these distributions. For knapsack sizes 100, 200, 300, 400 and 500 all 4 methods were compared, however for knapsacks of size 600 only the greedy method and simulated annealing methods were compared. This was due to the fact that the table generated by the FPAS algorithm could not be constructed in lisp for knapsack sizes larger than 500 if  $k=0.01$ . The results indicate that for problem sizes larger than 500, presorted simulated annealing (a.k.a “sort-anneal”) and the greedy strategy were the only competitive/executable algorithms.

Lastly, the columns labeled diff1 through diff5 represent the difference in performance of each algorithm vs. the greedy algorithm on the identical problem of the same size. In other words, the problem generated for trial-1 and size 100 given one of the three distributions was the same for each algorithm used and therefore the differences in performance could be measured. In this case positive values indicate better performance vs. the greedy strategy on the identical problem where as negative values indicate worse performance.

**Table 1-2: Results for uniformly distributed knapsack problem**

Algorithm	Size	trial-1	trial-2	trial-3	trial-4	trial-5
Greedy	100	38.38642	39.361732	39.69098	39.225746	38.733482
Greedy	200	74.88158	82.19867	81.789474	78.881195	82.980385
Greedy	300	125.14536	117.50394	116.342636	112.47009	118.98375
Greedy	400	169.0508	174.76265	164.61156	160.90106	159.38615
Greedy	500	200.53357	211.083	203.6872	207.25027	207.40776
Greedy	600	243.75488	252.52296	241.7148	250.96811	240.79984
FPAS k=0.1	100	38.17304	39.025234	39.642467	39.451702	38.665424
FPAS k=0.1	200	75.01549	82.23075	82.02221	79.247154	83.024826
FPAS k=0.1	300	125.17106	117.51701	116.78359	112.44231	118.65819
FPAS k=0.1	400	168.96954	174.65915	164.60083	161.3846	159.36172
FPAS k=0.1	500	200.13202	210.91208	203.4181	207.17719	207.248
FPAS k=0.1	600	243.17711	251.73665	241.46509	251.04677	240.4072
FPAS k=0.01	100	38.17304	39.361725	39.733913	39.499	38.733486
FPAS k=0.01	200	75.01549	82.36967	82.23123	79.39648	83.15072
FPAS k=0.01	300	125.17106	117.87214	116.914825	112.64989	118.99003
FPAS k=0.01	400	168.96954	174.96	164.76338	161.66505	159.76617
FPAS k=0.01	500	200.13202	211.3945	203.84108	207.4008	207.58916
Sim-Anneal	100	38.00917	39.19754	39.67327	39.41345	38.322765
Sim-Anneal	200	73.85249	82.0181	81.15657	78.5884	82.331375
Sim-Anneal	300	122.61891	116.748695	115.772415	111.1147	117.38103
Sim-Anneal	400	165.5802	172.14902	162.13023	160.30772	156.03258
Sim-Anneal	500	195.28612	206.82991	201.46358	204.53964	200.56416
Sim-Anneal	600	239.61871	247.91724	236.96147	247.08702	235.43036
Sort-Anneal	100	38.38642	39.361732	39.73392	39.498997	38.733482
Sort-Anneal	200	75.22999	82.27146	82.1796	79.28989	83.142586
Sort-Anneal	300	125.47418	117.66611	116.693436	112.65414	118.98375
Sort-Anneal	400	169.07477	174.84479	164.65044	161.58514	159.56891
Sort-Anneal	500	200.64523	211.12425	203.7424	207.33325	207.46553
Sort-Anneal	600	243.81987	252.52296	241.7186	251.23776	240.99855

Algorithm	Size	diff-1	diff-2	diff-3	diff-4	diff-5
FPAS k=0.1	100	-0.21338	-0.336498	-0.048513	0.225956	-0.068058
FPAS k=0.1	200	0.13391	0.03208	0.232736	0.365959	0.044441
FPAS k=0.1	300	0.0257	0.01307	0.440954	-0.02778	-0.32556
FPAS k=0.1	400	-0.08126	-0.1035	-0.01073	0.48354	-0.02443
FPAS k=0.1	500	-0.40155	-0.17092	-0.2691	-0.07308	-0.15976
FPAS k=0.1	600	-0.57777	-0.78631	-0.24971	0.07866	-0.39264
FPAS k=0.01	100	-0.21338	-7E-06	0.042933	0.273254	4E-06
FPAS k=0.01	200	0.13391	0.171	0.441756	0.515285	0.170335
FPAS k=0.01	300	0.0257	0.3682	0.572189	0.1798	0.00628
FPAS k=0.01	400	-0.08126	0.19735	0.15182	0.76399	0.38002
FPAS k=0.01	500	-0.40155	0.3115	0.15388	0.15053	0.1814
Sim-Anneal	100	-0.37725	-0.164192	-0.01771	0.187704	-0.410717
Sim-Anneal	200	-1.02909	-0.18057	-0.632904	-0.292795	-0.64901
Sim-Anneal	300	-2.52645	-0.755245	-0.570221	-1.35539	-1.60272
Sim-Anneal	400	-3.4706	-2.61363	-2.48133	-0.59334	-3.35357
Sim-Anneal	500	-5.24745	-4.25309	-2.22362	-2.71063	-6.8436
Sim-Anneal	600	-4.13617	-4.60572	-4.75333	-3.88109	-5.36948
Sort-Anneal	100	0	0	0.04294	0.273251	0
Sort-Anneal	200	0.34841	0.07279	0.390126	0.408695	0.162201
Sort-Anneal	300	0.32882	0.16217	0.3508	0.18405	0
Sort-Anneal	400	0.02397	0.08214	0.03888	0.68408	0.18276
Sort-Anneal	500	0.11166	0.04125	0.0552	0.08298	0.05777
Sort-Anneal	600	0.06499	0	0.0038	0.26965	0.19871

**Table 2-3: Results for a beta(alpha=5,beta=5) distributed knapsack problem**

Algorithm	Size	trial-1	trial-2	trial-3	trial-4	trial-5
Greedy	100	34.022243	33.15151	33.316498	32.31	31.31975
Greedy	200	63.929493	69.05249	65.99701	68.37825	68.71151
Greedy	300	98.357025	101.21598	99.94799	97.76426	98.192024
Greedy	400	137.18102	134.05849	130.57755	131.37947	130.86055
Greedy	500	163.8118	164.02805	160.76195	164.1903	167.1185
Greedy	600	201.34244	203.01654	202.23344	206.876	196.48657
FPAS k=0.1	100	34.268253	33.430504	33.170494	32.647747	31.490004
FPAS k=0.1	200	64.160995	69.05549	66.11851	68.38249	68.49024
FPAS k=0.1	300	98.04976	101.20328	100.07325	97.572716	98.42651
FPAS k=0.1	400	136.95773	134.11377	130.02199	131.378	130.28151
FPAS k=0.1	500	163.525	163.63828	160.36299	163.88026	166.61026
FPAS k=0.1	600	200.91151	203.00252	201.97006	206.49852	196.2235
FPAS k=0.01	100	34.268253	33.515007	33.4335	32.800995	31.704504
FPAS k=0.01	200	64.160995	69.442245	66.31926	68.52676	68.81525
FPAS k=0.01	300	98.04976	101.36978	100.39574	97.86198	98.607254
FPAS k=0.01	400	136.95773	134.7435	130.61249	131.703	130.95053
FPAS k=0.01	500	163.525	164.1623	161.13574	164.41429	167.11751
Sim-Anneal	100	34.1755	33.352757	33.196747	32.624496	30.969746
Sim-Anneal	200	63.309498	69.04399	65.99052	67.9205	68.06774
Sim-Anneal	300	96.996506	99.776024	98.39674	96.99922	97.489746
Sim-Anneal	400	135.31247	132.88023	128.18704	129.72401	129.26172
Sim-Anneal	500	161.52803	161.37997	158.89226	161.13428	163.96033
Sim-Anneal	600	198.205	199.94797	198.06898	203.58002	192.69495
Sort-Anneal	100	34.28325	33.467506	33.363747	32.616505	31.5915
Sort-Anneal	200	64.16525	69.31799	66.309265	68.473495	68.742004
Sort-Anneal	300	98.357025	101.25272	100.28599	97.835754	98.48128
Sort-Anneal	400	137.46252	134.61649	130.58354	131.64798	130.86055
Sort-Anneal	500	163.9523	164.08128	160.88844	164.37105	167.1185
Sort-Anneal	600	201.38618	203.4613	202.48645	207.031	196.74782

Algorithm	Size	diff-1	diff-2	diff-3	diff-4	diff-5
FPAS k=0.1	100	0.24601	0.278994	-0.146004	0.337747	0.170254
FPAS k=0.1	200	0.231502	0.003	0.1215	0.00424	-0.22127
FPAS k=0.1	300	-0.307265	-0.0127	0.12526	-0.191544	0.234486
FPAS k=0.1	400	-0.22329	0.05528	-0.55556	-0.00147	-0.57904
FPAS k=0.1	500	-0.2868	-0.38977	-0.39896	-0.31004	-0.50824
FPAS k=0.1	600	-0.43093	-0.01402	-0.26338	-0.37748	-0.26307
FPAS k=0.01	100	0.24601	0.363497	0.117002	0.490995	0.384754
FPAS k=0.01	200	0.231502	0.389755	0.32225	0.14851	0.10374
FPAS k=0.01	300	-0.307265	0.1538	0.447754	0.09772	0.41523
FPAS k=0.01	400	-0.22329	0.68501	0.03494	0.32353	0.08998
FPAS k=0.01	500	-0.2868	0.13425	0.37379	0.22399	-0.00099
Sim-Anneal	100	0.153257	0.201247	-0.119751	0.314496	-0.350004
Sim-Anneal	200	-0.619995	-0.0085	-0.00649	-0.45775	-0.64377
Sim-Anneal	300	-1.360519	-1.439956	-1.55125	-0.76504	-0.702278
Sim-Anneal	400	-1.86855	-1.17826	-2.39051	-1.65546	-1.59883
Sim-Anneal	500	-2.28377	-2.64808	-1.86969	-3.05602	-3.15817
Sim-Anneal	600	-3.13744	-3.06857	-4.16446	-3.29598	-3.79162
Sort-Anneal	100	0.261007	0.315996	0.047249	0.306505	0.27175
Sort-Anneal	200	0.235757	0.2655	0.312255	0.095245	0.030494
Sort-Anneal	300	0	0.03674	0.338	0.071494	0.289256
Sort-Anneal	400	0.2815	0.558	0.00599	0.26851	0
Sort-Anneal	500	0.1405	0.05323	0.12649	0.18075	0
Sort-Anneal	600	0.04374	0.44476	0.25301	0.155	0.26125

**Table 3-4 – Results for beta(alpha=10,beta=10) distributed knapsack problem**

Algorithm	Size	trial-1	trial-2	trial-3	trial-4	trial-5
Greedy	100	31.839254	30.26525	32.02775	30.120754	31.860504
Greedy	200	60.48675	62.58549	63.196747	62.132996	62.20949
Greedy	300	96.39425	92.15901	92.90448	93.54573	90.86126
Greedy	400	126.05521	124.06873	125.358	125.95599	125.227
Greedy	500	158.86201	155.49107	153.81602	153.52246	155.4965
Greedy	600	191.77542	186.82175	189.3814	186.07225	188.06474
FPAS k=0.1	100	31.740498	30.325003	32.034748	30.47275	32.232
FPAS k=0.1	200	60.54274	62.708755	63.26299	62.06924	61.961998
FPAS k=0.1	300	96.39176	91.86124	92.65653	93.501724	90.86826
FPAS k=0.1	400	125.68423	123.76225	125.04398	125.81847	124.89149
FPAS k=0.1	500	158.19406	154.94656	153.32106	153.51299	155.29105
FPAS k=0.1	600	190.85553	186.1888	188.61818	185.32245	187.68051
FPAS k=0.01	100	31.740498	30.456753	32.134247	30.570995	32.298744
FPAS k=0.01	200	60.54274	62.958504	63.62374	62.478745	62.269753
FPAS k=0.01	300	96.39176	92.36325	93.17979	93.87746	91.181755
FPAS k=0.01	400	125.68423	124.16876	125.53722	126.21621	125.31374
FPAS k=0.01	500	158.19406	155.61658	153.87904	154.04349	155.99728
Sim-Anneal	100	31.385496	30.189	31.9615	30.186249	31.8395
Sim-Anneal	200	58.726494	62.115498	61.72326	60.740498	60.561985
Sim-Anneal	300	95.111755	89.99398	91.48177	91.75424	90.351
Sim-Anneal	400	124.73502	121.68951	122.98174	124.3665	122.87097
Sim-Anneal	500	155.52023	152.2795	151.36876	149.911	152.47853
Sim-Anneal	600	187.25658	183.50299	185.03049	182.42049	182.38974
Sort-Anneal	100	31.894007	30.396252	32.072998	30.364258	32.1115
Sort-Anneal	200	60.927254	62.861748	63.491753	62.42974	62.23899
Sort-Anneal	300	96.696495	92.30826	93.10724	93.68499	91.06401
Sort-Anneal	400	126.07771	124.13899	125.43475	126.12698	125.28449
Sort-Anneal	500	158.90677	155.60231	153.87352	154.02872	155.84354
Sort-Anneal	600	191.77542	186.90724	189.3814	186.13701	188.39676

Algorithm	Size	diff-1	diff-2	diff-3	diff-4	diff-5
FPAS k=0.1	100	-0.098756	0.059753	0.006998	0.351996	0.371496
FPAS k=0.1	200	35.90501	29.27575	29.459783	31.368728	28.65877
FPAS k=0.1	300	29.28998	31.60324	32.139496	32.27274	34.03023
FPAS k=0.1	400	32.13885	30.87783	27.96306	27.557004	30.06405
FPAS k=0.1	500	31.99352	30.69773	34.80216	31.79999	32.18401
FPAS k=0.1	600	160.03492	-156.365	157.24715	155.50126	-155.766
FPAS k=0.01	100	-0.098756	0.191503	0.106497	0.450241	0.43824
FPAS k=0.01	200	0.05599	0.373014	0.426993	0.345749	0.060263
FPAS k=0.01	300	-0.00249	0.20424	0.27531	0.33173	0.320495
FPAS k=0.01	400	-0.37098	0.10003	0.179224	0.260224	0.086736
FPAS k=0.01	500	-0.66795	0.12551	0.06302	0.52103	0.50078
Sim-Anneal	100	-0.453758	-0.07625	-0.06625	0.065495	-0.021004
Sim-Anneal	200	-1.760256	-0.469992	-1.473487	-1.392498	-1.647505
Sim-Anneal	300	-1.282495	-2.16503	-1.42271	-1.79149	-0.51026
Sim-Anneal	400	-1.32019	-2.37922	-2.37626	-1.589486	-2.35603
Sim-Anneal	500	-3.34178	-3.21157	-2.44726	-3.61146	-3.01797
Sim-Anneal	600	-4.51884	-3.31876	-4.35091	-3.65176	-5.675
Sort-Anneal	100	0.054753	0.131002	0.045248	0.243504	0.250996
Sort-Anneal	200	0.440504	0.276258	0.295006	0.296744	0.0295
Sort-Anneal	300	0.302245	0.14925	0.20276	0.13926	0.20275
Sort-Anneal	400	0.022495	0.07026	0.07675	0.170994	0.05749
Sort-Anneal	500	0.04476	0.11124	0.0575	0.50626	0.34704
Sort-Anneal	600	0	0.08549	0	0.06476	0.33202

## 5.0 CONCLUSIONS

The presorted simulated annealing algorithm performed very well on all sizes and distributions of the knapsack problem. In the uniform case it beat the simple greedy strategy in 25 out of 30 trials. For the  $\beta(\alpha=5, \beta=5)$  distribution the results were improved in 27 out of 30 trials and for the  $\beta(\alpha=10, \beta=10)$  distribution the sort-anneal algorithm virtually dominated the results for the greedy strategy showing improved results in 28 out of 30 trials. Furthermore “sort-anneal” showed no reduction in performance due to problem size vs. the greedy strategy and it produced results that were

roughly comparable to the fully polynomial approximation scheme using a  $k$  value of 0.01 on problem sizes up to 500 (past which the FPAS algorithm could not be executed). Lastly, though in some cases the sort-anneal algorithm performed worse than the greedy strategy one should not conclude that it was incapable of beating the greedy algorithm on the particular problem generated. In fact simulated annealing often does not yield identical results when it is re-run (i.e. it is non-deterministic). This means that re-running and taking the best of several trial runs is a good strategy (and can be parallelized) and may in fact beat the greedy strategy.

---

<sup>1</sup> Cormen T., Leiserson C., Rivest R., Stein C., “Introduction to Algorithms”, McGraw Hill, 2001, pg. 382

<sup>2</sup> Hochbaum D. S., "Approximation Algorithms for NP-Hard Problems," PWS Publishing, 1997. pg. 363

<sup>3</sup> Hochbaum D. S., "Approximation Algorithms for NP-Hard Problems," PWS Publishing, 1997. pg. 363

<sup>4</sup> Hochbaum D. S., "Approximation Algorithms for NP-Hard Problems," PWS Publishing, 1997. pg. 363-364