EECE-7360 **Combinatorial Optimization** **FALL2013**

Research Project of Subset Sum Problem

Haofu Liao, Ti Wang

December 7, 2013

Department of Electrical and Computer Engineering

Northeastern University

ABSTRACT

In computer science, the Subset-sum Problem is an important problem in complexity theory and cryptography. It is a classic problem that has been studied for many years. In this project, we do the research based on this problem, and try to solve it in different ways. First, we developed a benchmark that consists of instances generated from different test problems for our evaluation. Then, we looked up all complexity results related to the Subset-sum Problem. We got a better understanding of this problem by knowing which subproblems can be efficiently solved and which remain NP-complete. Finally, we studied and implemented exhaustive, greedy, ILP, deepest decent and tabu search algorithms for the Subset-sum Problem. And evaluated all the experimental results from these algorithms. The implementation and evaluation of these algorithms helps us know better of combinatorial optimization solvers and also the Subset-sum Problem.

Contents

I. Introduction

The Subset-sum Problem (SSP) [10] is: given a set of n items and a knapsack, with

$$w_j = weight\ of\ item\ j;$$

$$c = capacity\ of\ the\ knapsack,$$

select a subset of the items whose total weight is closest to, without exceeding, c,
i.e.

$$\text{maximize} \qquad z = \sum_{j=1}^{n} w_j x_j$$

$$\text{subject to} \qquad \sum_{j=1}^{n} w_j x_j \le c,$$

$$x_j = 0 \text{ or } 1, \qquad j \in N = \{1, \dots, n\}$$

where

$$x_j = \begin{cases} 1 & \text{if item } j \text{ is selected;} \\ 0 & \text{otherwise.} \end{cases}$$

The Subset-sum Problem may be considered as a special case of the 0-1 Knapsack Problem, where the weight p_j equals the weight for each item. In spite of the restriction SSP has numerous applications: Diffe and Hellman [8] designed a public cryptography scheme whose security relies on the difficulty of solving the SSP problem, while Dietrich and Escudore [9] developed a procedure for strengthening LP bounds in general integer programming based on SSP problems. According to Martello and Toth [1] SSP can also be applied for solving cargo loading, cutting stock and two-processor scheduling problem.

II. Benchmark and Test Environment

A. Benchmark

Problem $P(E)$

From the optimization version, if we specify w_j uniformly random in $[1, 10^E]$ and $c = n\frac{10^E}{4}$ we can get our new sub-problem. Since we just specify some parameters of our original Subset-

Sum Problem, this sub-problem is still a NP problem.

Problem *EVEN/ODD*

From the optimization version, if we specify w_j even, uniformly random in $[1, 10^3]$ and $c = n\frac{10^3}{4} + 1(odd)$ we can get our new sub-problem. Since we just specify some parameters of our original Subset-Sum Problem, this sub-problem is still a NP problem.

Problem *TODD*

From the optimization version, if we specify $w_j = 2^{k+n+1} + 2^{k+j} + 1$, with $k = \lfloor \log_2 n \rfloor$, and $c = \left\lfloor 0.5 \sum_{j=1}^{n} w_j \right\rfloor = (n+1)2^{k+n} - 2^k + \left\lfloor \frac{n}{2} \right\rfloor$, we can get our new sub-problem. Since we just specify some parameters of our original Subset-Sum Problem, this sub-problem is still a NP problem.

Problem *AVIS*

From the optimization version, if we specify $w_j = n(n+1) + j$, and $c = \left\lfloor \frac{n-1}{2} \right\rfloor n(n+1) + \binom{n}{2}$, we can get our new sub-problem. Since we just specify some parameters of our original Subset-Sum Problem, this sub-problem is still a NP problem.

Choice of instances

Hence, we can generate instances in the following way.

Instance Index	Test Problems	Problem Size (n)
$A_1 - A_{25}$	P(3) : w_j uniformly random in $[1, 10^3]$; $c = n\frac{10^3}{4}$	4, 8, 12, …100
$A_{26} - A_{50}$	P(6) : w_j uniformly random in $[1, 10^6]$; $c = n\frac{10^E}{4}$	4, 8, 12, …100
$A_{51} - A_{75}$	P(12) : w_j uniformly random in $[1, 10^{12}]$;	4, 8, 12, …100

	$c = n\frac{10^{12}}{4}$	
$A_{76} - A_{100}$	EVEN/ODD : specify w_j even, uniformly random in $[1, 10^3]$; $c = n\frac{10^3}{4} + 1(odd)$	4, 8, 12, …100
$A_{101} - A_{125}$	TODD : $w_j = 2^{k+n+1} + 2^{k+j} + 1$, with $k = \lfloor \log_2 n \rfloor$; $c = \left\lfloor 0.5 \sum_{j=1}^{n} w_j \right\rfloor = (n+1)2^{k+n} - 2^k + \left\lfloor \frac{n}{2} \right\rfloor$	4, 5, 6, … 29
$A_{126} - A_{150}$	AVIS : $w_j = n(n+1) + j$; $c = \left\lfloor \frac{n-1}{2} \right\rfloor n(n+1) + \binom{n}{2}$	4, 8, 12, …100

Table 2.1 Instances in benchmark

So, in our benchmark, we have instances from 6 different test problems. Instances generated from the same problem have sizes of 4, 8, 12, …100. The largest instance is 25 times larger than the smallest instance. For each problem, there are 25 instances generated from it. Therefore, there are 150 instances all together.

B. Test Environment

CPU	2.3GHz Intel Core i7
Operating System	OS X 64bit
IDE	Xcode
Compiler	GCC
Programming Language	C++

Table 2.2 test environment

III. Related Problems and Complexity Results

A. Sub-problems

Partition Problem

Definition [2]: Given a finite set A, size $s(a) \in Z^+$ for each $a \in A$, is there a subset $A' \subseteq A$ such that $\sum_{a \in A'} s(a) = \sum_{a \in A - A'} s(a)$?

Is this a sub-problem of subset sum?

Yes. By the decision definition we know that this problem is a special case of Subset-Sum Problem when we let $B = one\ half\ of\ the\ sum\ of\ all\ integers$. Hence, all instances of Partition Problem belong to the set of all instance of Subset-Sum Problem. Since, Partition Problem is a special case, we know that when an instance of this problem is a yes-instance, we know it is also a yes-instance of Subset-Sum Problem. So, the Partition Problem is a sub-problem of Subset-Sum Problem.

Complexity: NP-Complete [2].

3SUM Problem

Definition [3]: Given a set S of n integers, are there $a, b, c \in S$ with $a + b + c = 0$?

Is this a sub-problem of subset sum?

Yes. To make this problem looks like Subset-Sum Problem we define it in this way: given a finite set A, size $s(a) \in Z^+$ for each $a \in A$, a integer $B = 0$. Is there a subset $A' \subseteq A, |A'| = 3$ such that the sum of the sizes of the elements in A' is exactly B? Here, we can find that we specify the $B = 0$ and the size of subset $|A'| = 3$. It is a special case of Subset-Sum Problem. Then, for any instances of this problem, it is also an instance of Subset-Sum Problem. If an instance of 3SUM is a yes-instance, we know that there exists a subset A' whose sum of the sizes of the elements is $B = 0$. Hence, this instance is also a yes-instance of the Subset-Sum Problem. So, it is a sub-problem of subset sum.

Complexity: This problem can be solved in polynomial-time [3].

B. Super-problems

Knapsack Problem

Definition [1]: Given a set of n items and a knapsack, with

$$p_j = profit\ of\ item\ j,$$

$$w_j = weight\ of\ item\ j,$$

$c = weight\ of\ item\ j,$

select a subset of the items so as to

$$\text{maximize} \qquad z = \sum_{j=1}^{n} w_j x_j$$

$$\text{subject to} \qquad \sum_{j=1}^{n} w_j x_j \le c,$$

$$x_j = 0 \text{ or } 1, \qquad j \in N = \{1, \dots, n\}$$

where

$$x_j = \begin{cases} 1 & \text{if item } j \text{ is selected;} \\ 0 & \text{otherwise.} \end{cases}$$

Is the Subset-Sum Problem a sub-problem of this problem?

Yes. From the definition of 0-1 Knapsack Problem, we can know that if we let $p_j = w_j$ this problem is a Subset-Sum problem. Then for any instance of Subset-Sum problem, it is also an instance of 0-1 Knapsack Problem. Once we get a subset that maximized $z = \sum_{j=1}^{n} w_j x_j$ of Subset-Sum problem, it must also maximized $z = \sum_{j=1}^{n} w_j x_j$ of 0-1 Knapsack Problem. So the Subset-Sum problem is a sub-problem of 0-1 Knapsack Problem.

Complexity: NP-Complete [1].

C. Relationships

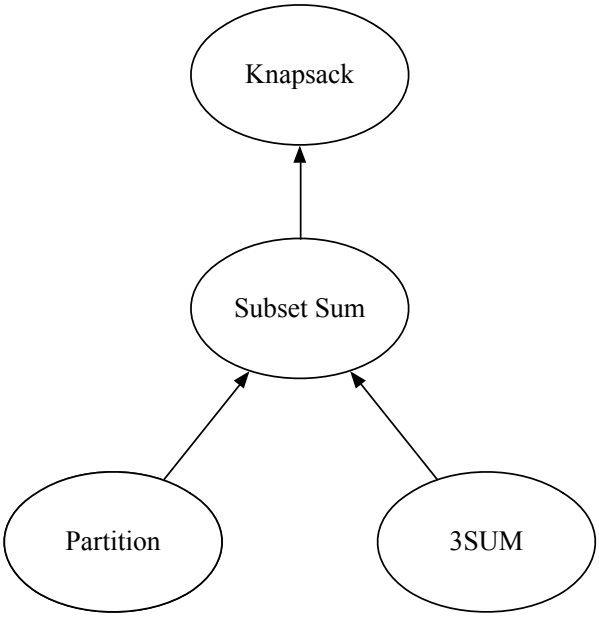


Figure 3.1 relationship between problems

D. Other Related Problems

Kth Largest Subset

Definition [2]: Given a finite set A, size $s(a) \in Z^+$ for each $a \in A$, and two non-negative integers $B \leq \sum_{a \in A} s(a)$ and $K \leq 2^{|A|}$. Are there a subset $A' \subseteq A$ such that satisfy $s(A') \leq B$ (where $s(A')$ is defined to be $\sum_{a \in A'} s(a)$)?

Complexity: NP-hard [2].

Capacity Assignment

Definition [2]: Set C of communication links, set $M \subseteq Z^+$ of capacities, cost function $g: C \times M \to Z^+$, delay penalty function $d: C \times M \to Z^+$such that, for all $c \in C$ and$i < j \in M$, $g(c,i) \leq g(c,j)$ and $d(c,i) \geq d(c,j)$, and positive integers K and J. Is there an assignment $\sigma: C \to M$ such that the total cost $\sum_{c \in C} g\big(c, \sigma(c)\big)$ does not exceed K and such that the total delay penalty $\sum_{c \in C} d\big(c, \sigma(c)\big)$does not exceed J?

Complexity: NP-Complete [2].

Integer Expression Membership

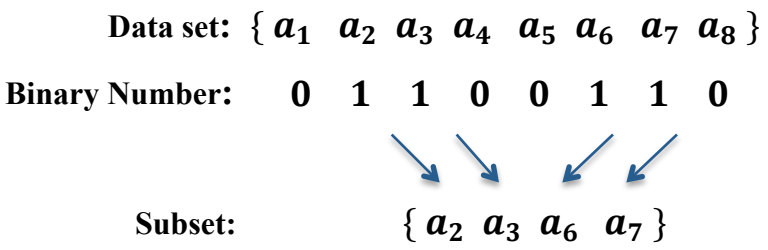
Definition [2]: Integer expression e over the operations $\cup$ and $+$, where if $n \in Z^+$, the binary representation of n is an integer expression representing n, and if f and g are integer expressions representing the sets F and G, then $f \cup g$ is an integer expression representing the set $F \cup G$ and $f + g$ is an integer expression representing the set $\{m + n : m \in F \text{ and } n \in G\}$, and a positive integer K. Is K in the set represented by e?

Complexity: NP-hard [2].

IV. Exhaustive Algorithm

A. Bitwise Algorithm

To make sure we have considered all the subsets in of our instance, we use a binary number to represent each subset. The length of the binary number is equal to the size of the instance, i.e. the number of elements in the data set. Every bit of the binary number stands for a number among the instance. If a bit is one, it means this number is chosen and should be add to this subset. If a bit is zero, it means we won't take this number into our subset. For example, below is an instance of size eight, i.e. there are eight numbers in our data set. Then our length of the binary number should be eight. If we have a binary number 01100110, we can know that its 2nd, 3rd, 6th and 8th bit are one. So we should take a_2, a_3, a_6 and a_7 into our subset.



If we change to another binary number, we can get a new subset which definitely different from our current one. Since the binary numbers is a one to one map to our subsets and the number of subsets is 2^n, which is equal to the amount of numbers of a n length binary number, we can know that in this way we considered all possible subsets and each subset is being considered only once.

Once, we've got all the subsets, all we should do in the next is just sum all the elements of each subset and compare the sum with the given integer to determine whether they are the same. If the answer is yes, we can know we got our solution.

Algorithm 1 Bitwise Exhaustive Algorithm

Input: *dataSet* = an vector that contains all the numbers in the set.

value = the target value.

Output: a bool that indicates whether the subset is found

```
subsetSumBitwise(dataSet, value)
1:  for  i = 0 to total number of subsets – 1
2:       bin = transform i to its corresponding binary number
3:       bits[] = save every bits of bin into bits[]
4:       sum = 0
5:       for  j = 0 to total number of numbers in set
6:            if  current bit bits[j]== 1
7:                 add the corresponding number dataSet[j] to sum
8:            end if
9:       end loop
10:      if  sum== value
11:           return true
12:      end if
13: end loop
14: return false
```

Table 4.1 Bitwise exhaustive algorithm

B. Recursive Algorithm

In the recursive algorithm, we start from every single element of our instance. Assume that the size of instance is n, the data set can be denoted as $\{a_1, a_2, \dots a_n\}$, current level is 1. Then for any element a_i, we add all the elements whose index is larger than the current element's index to it. That is, we can add $a_{i+1}, a_{i+2}, \dots, and\ a_n$ to a_i respectively and get subsets $\{a_i a_{i+1}\}$, $\{a_i a_{i+2}\}, \dots, and\ \{a_i\ a_n\}$. After we did this for every element of the instance, we can get a new bunch of subsets. All those subsets have two elements, and they are different from each other. We say those subsets are on the same level. This level is 2. Based on this new level, we then start

from every subset of this level, i.e. from subset $\{a_1a_2\}\ to\ \{a_{n-1}a_n\}$. Hence, all the two-elements subsets are considered. For each two-elements subset, we add all the elements whose index is larger than the current subset's maximum element index. That is we add $a_{j+1}, a_{j+2}, \dots, and\ a_n$ to $\{a_ia_j\}$. After that, we can also get a new bunch of subsets. At this time, all those subsets will have three elements, and they are different from each other. They are now on level 3. Continue doing this kind of job recursively, we can finally traverse all the subsets. And since all the subsets are different from each other, we can know we consider every solution only once. For all these subsets we will consider the sum and compare it with the given integer K. Once one sum is found equal to K, we stop our recursive and return true. Otherwise, we continue our job until all the subsets are considered. One nice thing should be mentioned is that since every current level subset are based on the last level subset, then when we count the summation, we don't need to sum all current level subset's elements together. All we need to do is just add the new element from current level to the last level's sum results. The following figure is an example of a four-elements data set. We compute the sum results recursively.

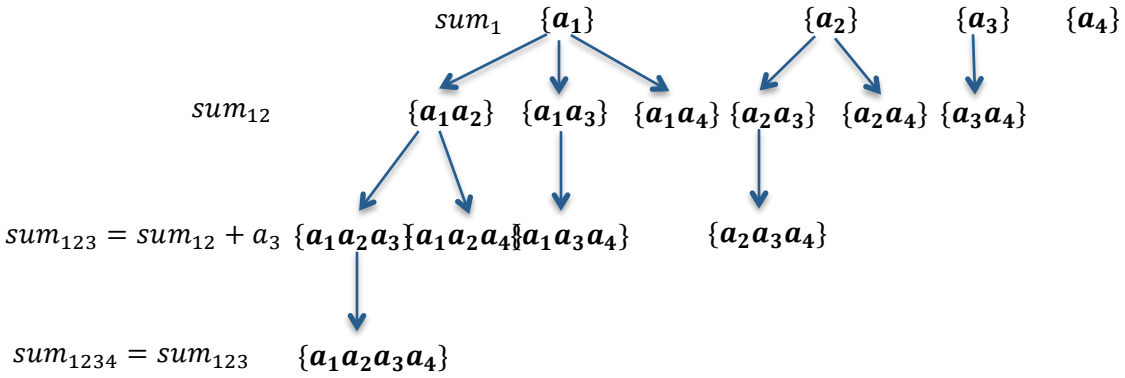


Figure 4.1 recursive algorithm

Algorithm 2 Recursive Exhaustive Algorithm

Input: *dataSet* = an vector that contains all the numbers in the set.

value = the target value.

head = a pointer that indicate the start of current vector

sum = the current sum that counted

Output: a bool that indicates whether the subset is found

---

subSetSumRecursive(*dataSet*, *value*, *head*, *sum*)

```
1:  for  i = head to total number of numbers in the set – 1
2:          currentSum = the sum of sum and dataSet
3:          if  currentSum == value or subSetSumRecursive(dataSet,
4:         value, i+1, currentSum)
5:                 return true
6:          end if
7:  end loop
8:  return false
```

Table 4.2 Recursive exhaustive algorithm

C. Evaluation

Test Steps:

In our test, we pick up instances of different sizes from the benchmark. We start from the instance of size 5, and increase the size of instance gradually. During the test, we record the size of different instance and the elapsed time that it takes to solve this instance. We set two CPU time limits. One is 1 minute, and the other is 10. Once it takes more time than the limits to solve an instance, we record the instance's size and stop the test. All the two exhaustive algorithms we introduced above will be applied to our test and they will be force two solve every instance in the worst condition which means no subset can be found.

Test Results and Analysis:

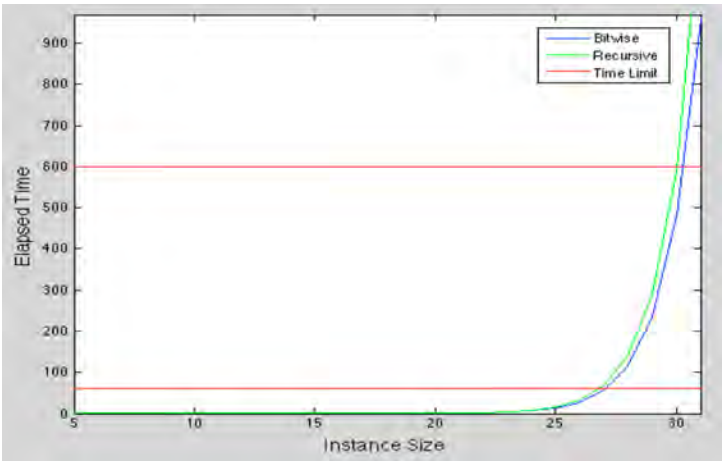


Figure 4.2 test result of exhaustive algorithms

The figure above gives out our test results. The blue curve represents the bitwise algorithm, the green curve represents the recursive algorithm and the red lines indicate our time limits (1 minute and 10 minutes). From the figure we can know both the two algorithms grows very fast, more precisely in the order of exponential. Basically, our bitwise algorithm grows slower than recursive algorithm. It is because the recursive is slower than loops in C++. For bitwise algorithm the largest size that can be solved in 1 minute is 27 and for recursive algorithm the largest size is 26. The largest size that can be solved in 10 minutes is 30 for both algorithms. It means our exhaustive algorithms can only solve about 20% instances among our benchmark.

V. Greedy Algorithm

A. Algorithm

The most natural greedy algorithm of the subset sum problem is

Algorithm Greedy Algorithm

Input: *items* = a vector that contains all the items in the set.

c = the target value.

Output: *opt* = the heaviest subset that greedy algorithm can find, without exceeding the target value.

```
greedy(dataSet, value)
1:    opt = ∅
2:    sort the items into decreasing order by weight
3:    weight = 0
4:    for  i = 0 to total number of items - 1
5:         if  the weight of  opt ∪ items[i] less than c
6:              opt = opt ∪ items[i]
7          end if
8:    end loop
9:    return opt
```

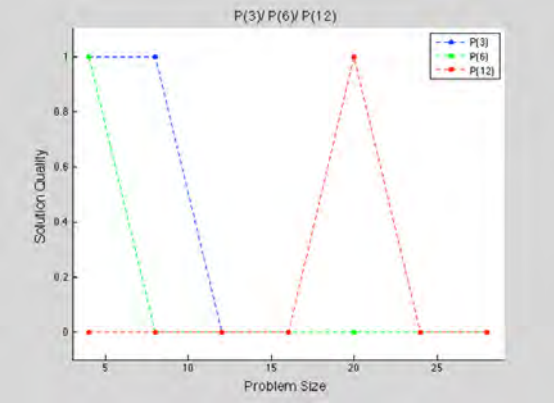
Table 5.1 Greedy algorithm

B. Cases of Failure

The algorithm we developed above does not always find the optimal solution. For example, consider a given set $\{7,6,3,1\}$ and $c = 9$. If we use greedy algorithm, the solution will be $\{7,1\}$, so $weight = 8$, but if we just pick $\{6,3\}$, $weight = 9$ is an optimal solution. In such situation, the greedy algorithm cannot find optimal solution. The reason this situation will happen is because our greedy algorithm always try to contain the heaviest item into the optimal solution. But actually the optimal solution doesn't necessarily contain such item. There may be several light objects that have a total weight that much more close to the target value.

C. Evaluation

Since we can get the optimal results using exhaustive algorithm when our problem size is small, we first test the performance of greedy algorithm for small instances.



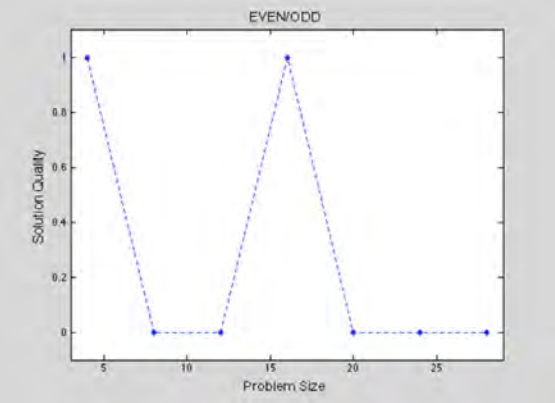


Figure 5.1 Quality of Greedy algorithm for P(E) Figure 5.2 Quality of Greedy algorithm for EVEN/ODD

Figure 5.1 is the solution quality of P(E) problem. Here 1 means the optimal solution is found and 0 means the optimal solution is not found. We can find that for P(3) problem only 2 instance got the optimal solution when we use greedy algorithm. Hence, it means the greedy algorithm performs badly on this kind of problem.

Figure 5.2 shows the solution quality for EVEN/ODD problem. We can also see that only few instances find the optimal solution. Hence, greedy algorithm has also a bad performance on this EVEN/ODD.

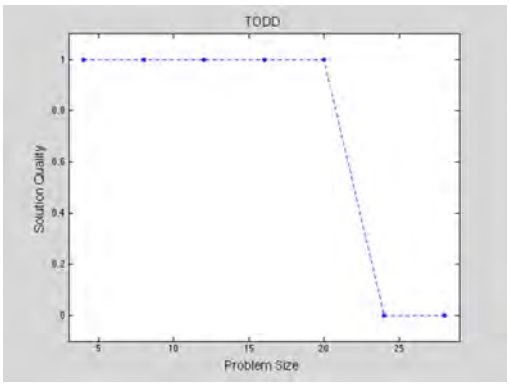


Figure 5.3 Quality of Greedy Algorithm for TODD

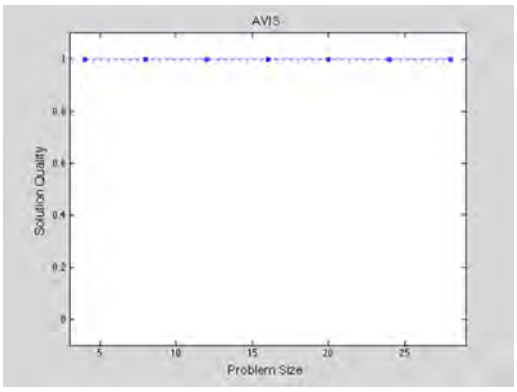


Figure 5.4 Quality of Greedy algorithm for AVIS

Here, we can find that for TODD problem, the greedy algorithm have a good performance. Only 2 instances failed. Actually, the algorithm fails on these 2 instances is no because it can't find the optimal solution, but because the number is so large (on the 2^{24} level) that the computer can't process those numbers.

From figure 5.4, we know the greedy algorithm performance very good on AVIS algorithm. All instances are found the optimal solution. The reason that greedy algorithm performances good on this problem is the difference between numbers of this problem is small. Then, we can't sum some small numbers and get the result that is a little bit large than a large number.

Since all the performance are similar on large size instances. We can get the conclusion. The exhaustive algorithm is a good choice, if we want to solve P(E) and EVEN/ODD problem on small size instances. And we can take greedy algorithm into consideration if the problem size is large. But if we want to solve TODD and AVIS greedy algorithm is always the better choice.

VI. ILP Algorithm

A. ILP Formulation

1. ILP formulation of Knapsack

Since the subset sum problem is a special case of Knapsack problem, it is important to figure out the ILP formulation of Knapsack first.

$$\text{maximize} \qquad z = \sum_{j=1}^{n} p_j x_j$$

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

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

2. ILP formulation of Subset Sum

We can get the ILP formulation of Subset Sum by setting $p_j = w_j$. Hence, we get

$$\text{maximize} \qquad z = \sum_{j=1}^{n} w_j x_j$$

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

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

3. Compute IL lower bound

We calculate the LP lower bound by removing the binary constrains. For our Subset Sum problem, we can always find x_j values, such that the maximum capacity can be reached. Hence, the LP lower bound for Subset Sum problem is

$$Z_{LP} = c$$

where c is the capacity (target) of the subset sum problem.

B. Evaluation

1. Data Representation

How we present our data?

Quality of the greedy solution and LP lower bound relative to the optimal value

Since the data value of different instance in our benchmark varies significantly, it is

inappropriate to use the absolute value to evaluate the gaps. Hence, the relative value is a good choice. Here, we choose the optimal value (ILP solution) as the standard value ($= 1$). Then we calculate the ratio of lower bound and optimal value, and the ratio of greedy solution and optimal value. The formulation is given below:

$$R_{opt} = 1$$

$$R_{greedy} = Z_{greedy}/Z_{opt}$$

$$R_{LP} = Z_{LP}/Z_{opt}$$

How often is the greedy solution optimal?

Usually, we should calculate the number of the greedy solutions that equal to the optimal solution and divide it by the total number of the greedy solution. The formulation is

$$P = \frac{\sum_{i=1}^{N} f\left(Z_{greedy}^{i}, Z_{opt}^{i}\right)}{N}$$

Where N is the number of instances, Z_{greedy}^{i} is the greedy solution of ith instance, Z_{opt}^{i} is the optimal solution of ith instance and

$$f(x, y) = \begin{cases} 1, & if\ x = y \\ 0, & if\ x \neq y \end{cases}$$

But, if we calculate in this way, we won't get the correct answer. Because, the optimal solution we got from AMPL is not precise when the absolute value of data is very large. Hence we need to find a way to avoid this situation. The formulation is given below:

$$P = \frac{\sum_{i=1}^{N} f\left(Z_{greedy}^{i}, Z_{opt}^{i}\right)}{N}$$

Where N is the number of instances, Z_{greedy}^{i} is the greedy solution of ith instance, Z_{opt}^{i} is the optimal solution of ith instance and

$$f(x, y) = \begin{cases} 1, & if\ \frac{x}{y} > 0.99999 \\ 0, & if\ x \neq y \end{cases}$$

Since, Z_{greedy}^{i} is always no larger than Z_{opt}^{i}, we know $\frac{Z_{greedy}^{i}}{Z_{opt}^{i}}$ is between $0.99999 \sim 1$.

How often is the LP lower bound tight?

When we think about the tight LP lower bound, the first question we must answer is what LP lower bound can be defined as tight. For our Subset Sum problem, we observed hundreds of instances and got the following empirical formulation:

$$P = \frac{\sum_{i=1}^{N} f\left(Z_{LP}^{i}, Z_{opt}^{i}\right)}{N}$$

Where N is the number of instances, Z_{LP}^{i} is the greedy solution of ith instance, Z_{opt}^{i} is the optimal solution of ith instance and

$$f(x, y) = \begin{cases} 1, & if \frac{x}{y} < 1.01 \\ 0, & if\ x \neq y \end{cases}$$

Since, Z_{LP}^{i} is always no larger than Z_{opt}^{i}, we know $\frac{Z_{LP}^{i}}{Z_{opt}^{i}}$ is between 1.01~1

2. Graphic Results

Quality of the greedy solution and LP lower bound relative to the optimal value

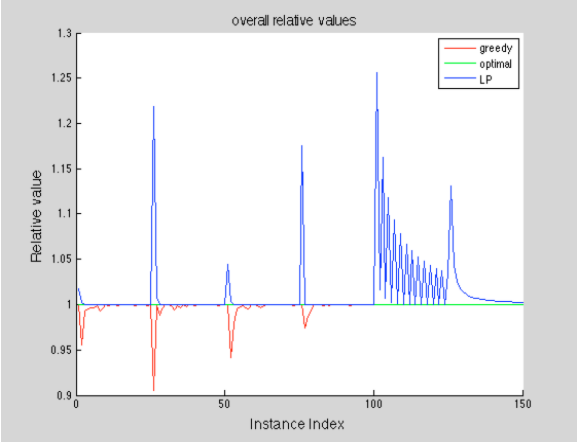


Figure 6.1 overall relative values

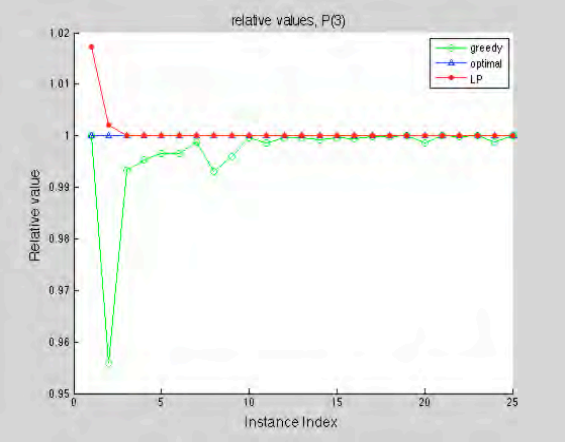


Figure 6.2 P(3) relative values

Figure 6.1 shows the overall quality of greedy solution and LP lower bound relative to the optimal value. In general, LP lower bound finds better solution than greedy solution. For most of the instances, LP lower bound is close to the optimal solution, and LP solution is at most 10% far from the optimal solution. Greedy solution also performs well, but for some instances such as the last two groups, greedy algorithm rarely finds optimal solution.

For P(3) instances shown in figure 6.2, when the sizes of problems are small, differences between optimal solution and greedy or LP solution are big, but as the sizes grow, LP and greedy solutions are very close to optimal solution.

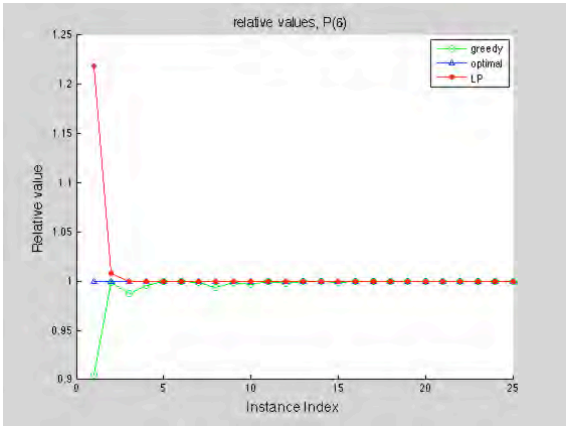


Figure 6.3 P(6) relative values

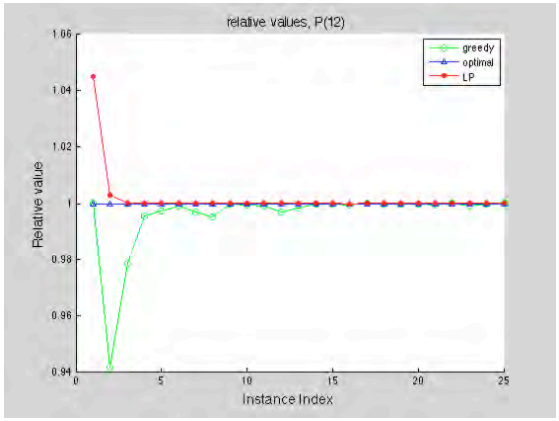


Figure 6.4 P(12) relative values

We can see P(6)'s case in figure 6.3, which is familiar to P(3)'s case, LP and greedy solutions become closer to optimal solutions as the sizes of problems grow.

Figure 6.4 shows P(12)'s relative values. It is same as P(3) and P(6), the bounds become tighter as the sizes become bigger.

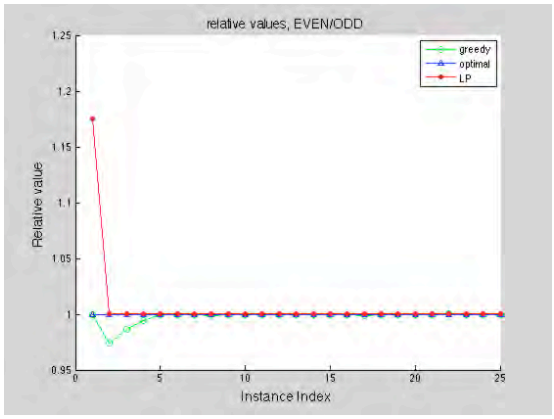


Figure 6.5 Even/Odd relative values

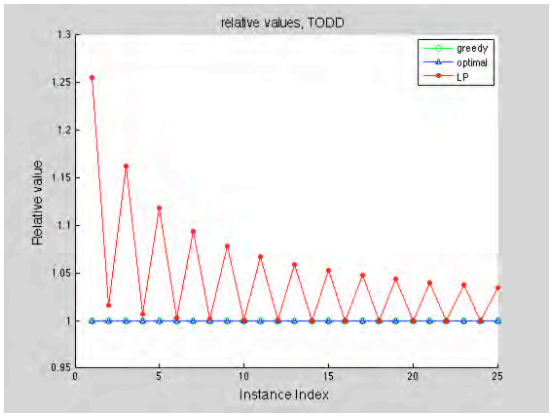


Figure 6.6 TODD relative values

The Even/Odd instances' case is shown in figure 6.5, when the number of elements is bigger than 20, LP finds optimal solution and when the number of elements is bigger than 4, greedy finds optimal solution.

Figure 6.6 shows the TODD's case. The greedy algorithm always finds optimal solution for TODD instances. And the LP bound is much tighter when the numbers of elements are even.

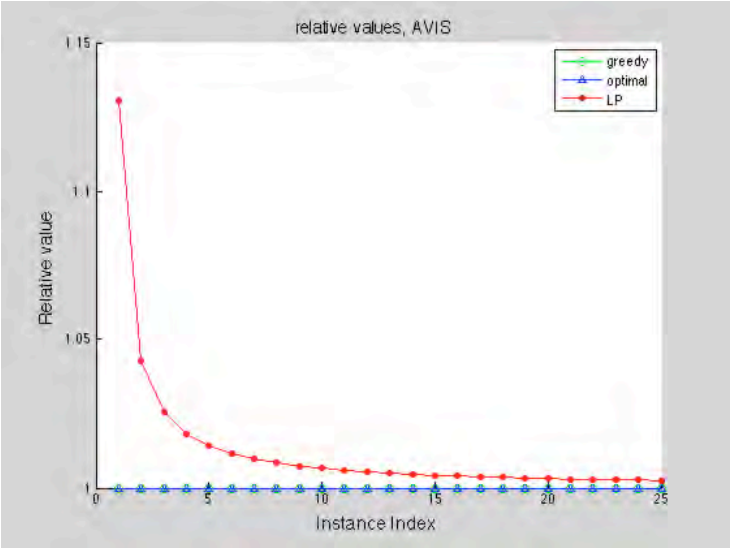


Figure 6.7 AVIS relative values

AVIS ‘s case is shown in figure 6.7. Greedy algorithm always finds optimal solution in this case. And as the problem size become bigger, LP bound become tighter.

	Overall	**P[3]**	**P[6]**	**P[12]**	**EVEN/ODD**	**TOOD**	**AVIS**
Greedy solutions are optimal	49%	12%	20%	24%	36%	100%	100%
Greedy solutions are not optimal	51%	88%	80%	76%	64%	0%	0%

Table 6.1 how often greedy solution is optimal

	Overall	**P[3]**	**P[6]**	**P[12]**	**EVEN/ODD**	**TOOD**	**AVIS**
LP bounds are tight	84%	96%	96%	96%	96%	44%	76%
LP bounds are not tight	16%	4%	4%	4%	4%	56%	24%

Table 6.2 how often LP bound is tight

VII. Local Search Algorithm

A. Initialization and Neighborhood Function

1. Initialization

We two techniques for finding the initial solution: **random** and **greedy**. For random initialization, we uniformly generate a random number from 0 to 2^n-1, where n is the size of our instance. Then, we convert this number to binary that corresponds to a subset of the instance. If the sum of elements in the subset is less than c, we know it is a valid solution, and we take it as the initial solution. For greedy initialization, we use the greedy algorithm we implemented in section V to generate a solution. Since, greedy algorithm have a good performance on our problem, this way should be better than random initialization.

2. Neighborhood Function

The choice of the neighborhood is perhaps the single most important problem-specific decision in local search. The most intuitive one for SSP is perhaps the k-swap [7] neighborhood structure. A vector $X = \{x_1, x_2, \dots, x_n\}$ is said to belong to the k-swap neighborhood of a solution $Y = \{y_1, y_2, \dots, y_n\}$ if and only if $\sum_{j=1}^{n} w_j x_j \leq c$ and $\sum_{j=1}^{n} \left|(x_j - y_j)\right| \leq k$. Notice that as the value of k increases, the neighborhood becomes larger. This results in the possibility of more dramatic improvements in solution value at each iteration at the cost of longer execution time per iteration. It is easy to see that for a k-swap neighborhood, each local search iteration takes $O(n^k)$ time. This makes local search iterations quite expensive for $k > 2$.

B. Steepest Decent

We implement the Steepest Decent algorithm in the following way:

Step 1: Select an initial solution according to the initialization method discussed above.

Step 2: Find the neighborhood of current solution.

Step 3: Choose the best neighbor.

Step 4: If the best neighbor is better than current solution, update current solution with the best

solution and go to Step 2. Terminate if otherwise.

C. Tabu Search

We choose Tabu search as our second algorithm. Tabu search algorithm keeps a list of the last K solutions considered. And makes sure these solutions can't be visited again. This list is our tabu list. We choose K to be large enough to eliminate most small cycles. It is implemented in the following way:

Step 1: Select an initial solution according to the initialization method discussed above. Set $i^* = i$ and $k = 0$.

Step 2: Set $k = k + 1$ and generate a subset V of solution in $N(i, k)$.

Step 3: Choose a best j in V and set $i = j$.

Step 4: If $f(i) < f(i^*)$ then set $i^* = i$.

Step 5: Update Tabu list.

Step 6: If a stopping condition is met then stop. Else go to Step 2

Here, we have two stopping conditions. The first one is $N(i, k + 1) = 0$, which means no feasible solution is found in the neighborhood of solution. The second one is the time limit. If the execution time is larger than the time limit we stop searching.

D. Evaluation

In this subsection, we will only evaluate the quality of solutions between different types of Steepest Decent or Tabu algorithms. The comparison of the exhaustive algorithm, ILP, greedy, local search, and the LP lower bound will be discussed in the last section.

As we have discussed in section V

1. Steepest Decent

For Steepest Decent, the execution time of all the instances in the benchmark didn't exceed the time limit. Hence, we only have two types of Steepest Decent: Steepest Decent with random initialization and Steepest Decent with greedy initialization.

Since we have 6 different types of instances in our benchmark, we will evaluate them

respectively. The following is the evaluation results.

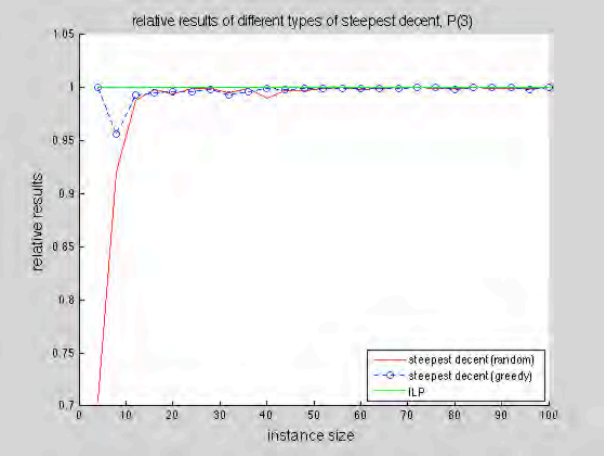


Figure 7.1 relative result of P[3] of steepest decent

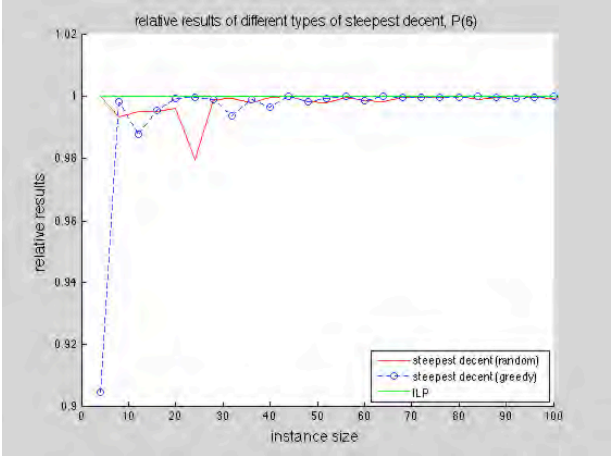


Figure 7.2 relative result of P[6] of steepest decent

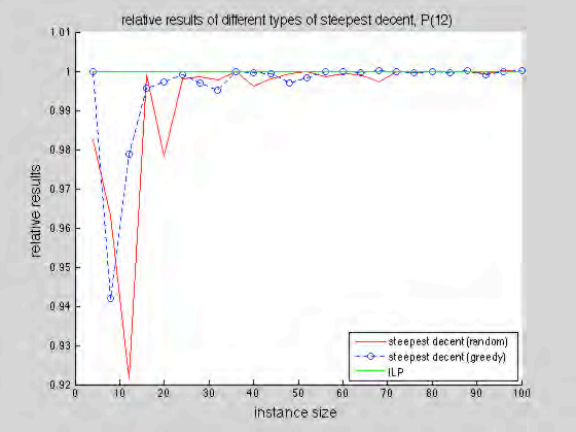


Figure 7.3 relative result of P[12] of steepest decent

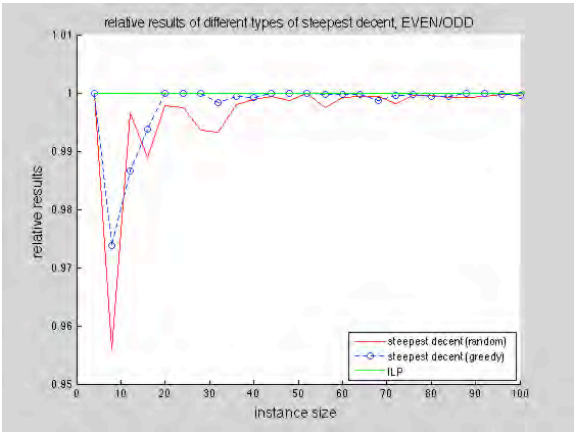


Figure 7.4 relative result of EVEN/ODD steepest decent

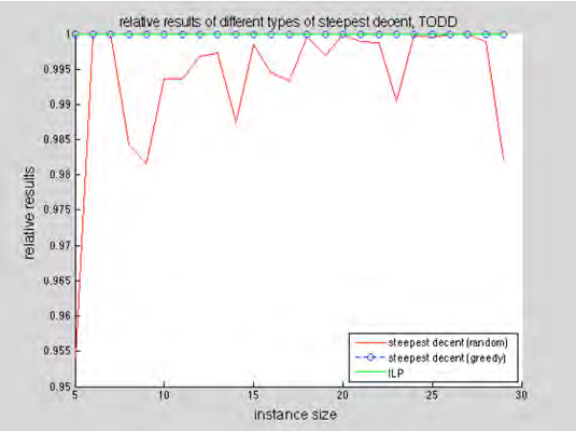


Figure 7.5 relative result of TOOD steepest decent

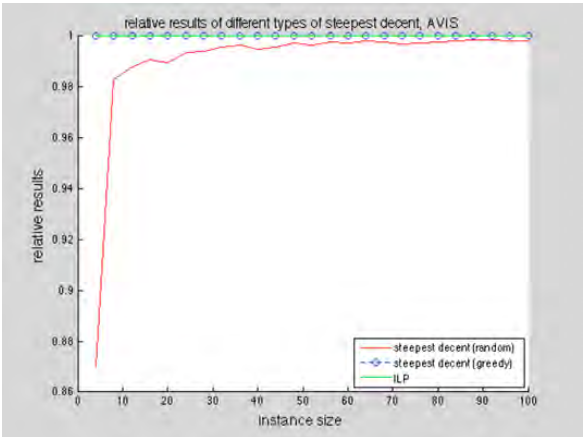


Figure 7.6 relative result of AVIS steepest decent

For instances of problem P(3), P(6), and P(12), random initialization and greedy initialization have a very close performance on the average. Steepest Decent performance better on P(12) than P(6) and P(3). For problem EVEN/ODD, greedy initialization has better performance than random initialization. For problem TODD and AVIS, the greedy initialization is not only better than random initialization, but gives the optimization solutions. We can also see that for problem TODD, the random initialization Steepest Decent algorithm has a random performance with the size of instance growing up. While for other problems, the performance of Steepest Decent algorithm becomes better when the size of instance growing up.

2. Tabu

For Tabu search algorithm, we have 2 ways of choosing initial solution, and 2 time limits. Hence, there are 4 types of Tabu search: Tabu search with random initialization and 1min time limit, Tabu search with random initialization and 5min time limit, Tabu search with greedy initialization and 1min time limit, and Tabu search with greedy initialization and 5min time limit.

Since we have 6 different types of instances in our benchmark, we will evaluate them respectively. The following is the evaluation results.

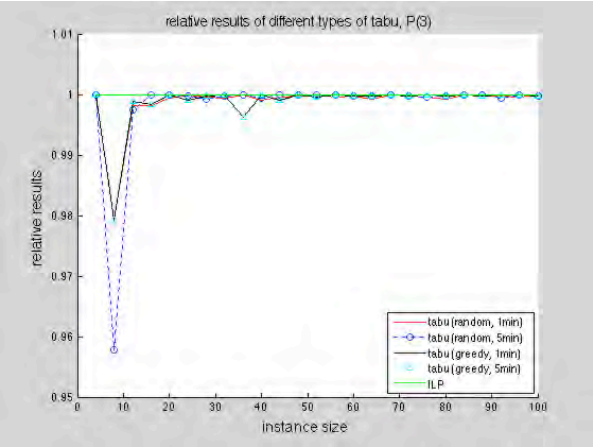


Figure 7.7 relative results of P[3] of tabu search

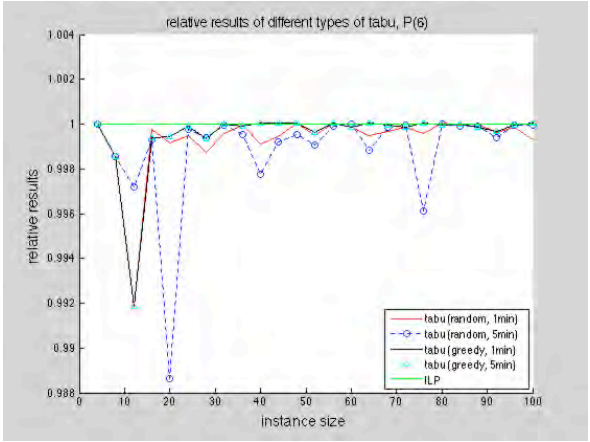


Figure 7.8 relative results of P[6] of tabu search

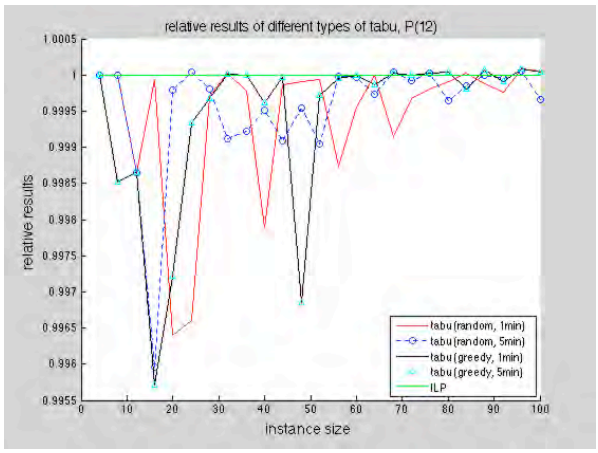


Figure 7.9 relative results of P[12] of tabu search

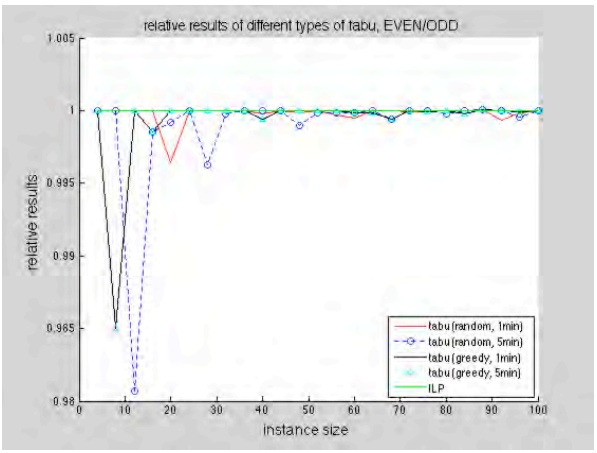


Figure 7.10 relative results of EVEN/ODD tabu search

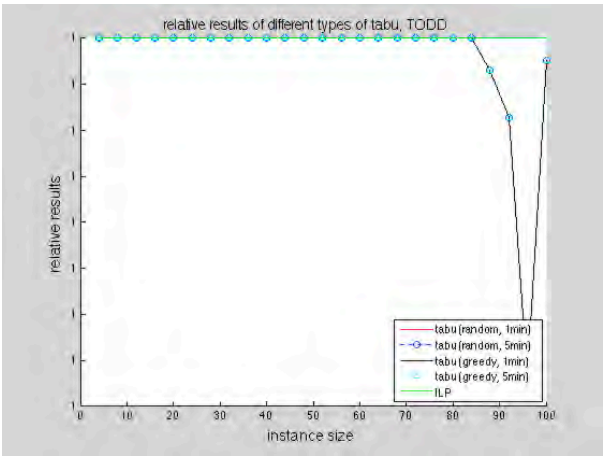


Figure 7.11 relative results of TOOD of tabu search

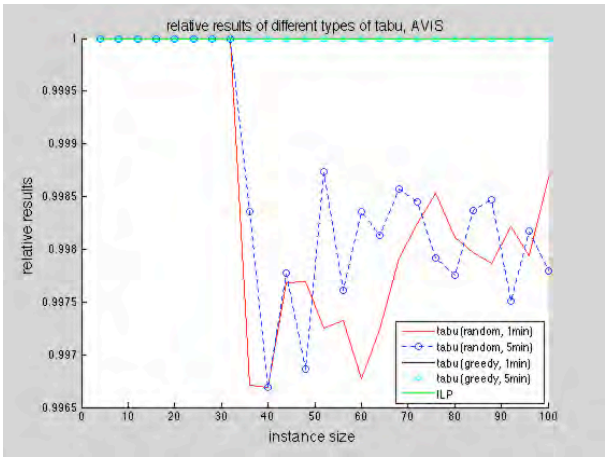


Figure 7.12 relative results of AVIS of tabu search

For instances of problem P(3), P(6) and P(12), the performance of all four types of Tabu search algorithm are close on the average. Tabu search have a better performance on P(12) than P(3) and P(6). For EVEN/ODD problem and AVIS problem, greedy initialization performances better than random initialization on the average. For the TODD problem all four types of Tabu search algorithm almost give the same solutions. We can find that 1min and 5min time limit have little influence on the results for both random initialization and greedy initialization. By comparison between Steepest Decent and Tabu search, we can find Tabu search have a better performance than Steepest Decent algorithm.

VIII. Summary

In this section, we will compare all the algorithms we implemented in this project. Both the quality of solutions and runtime of algorithms will be discussed. Since, our benchmark contains different types of instances, we will discuss them respectively.

A. Quality

Here, we compared the quality of the solutions of Steepest Decent, Tabu, greedy, ILP, and exhaustive algorithms and LP lower bound. Instances from problem P(3), P(6), P(12), EVEN/ODD, TODD, and AVIS are compared separately. Time limit for all the algorithms is 5min.

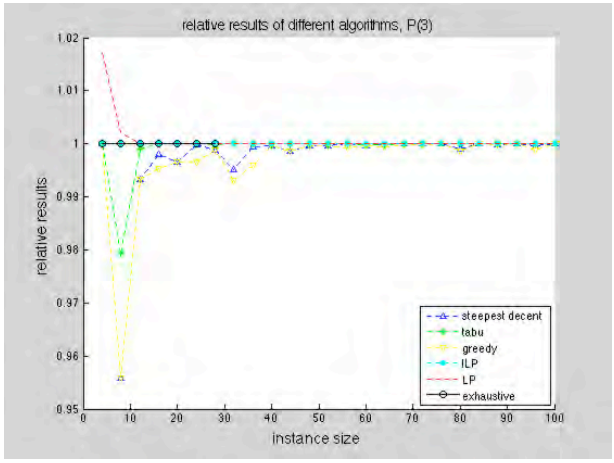


Figure 8.1 relative results of different algorithms P(3)

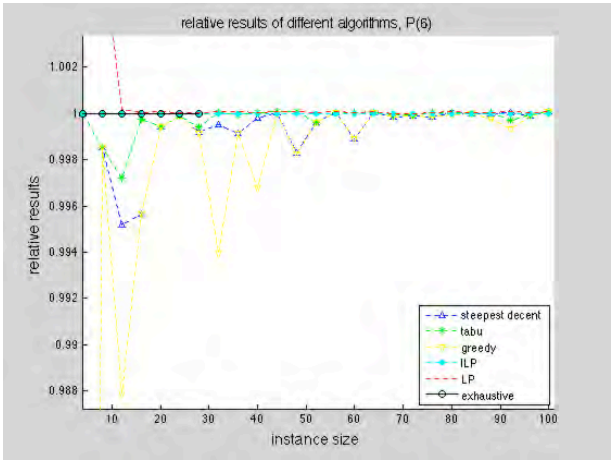


Figure 8.2 relative results of different algorithms, P(6)

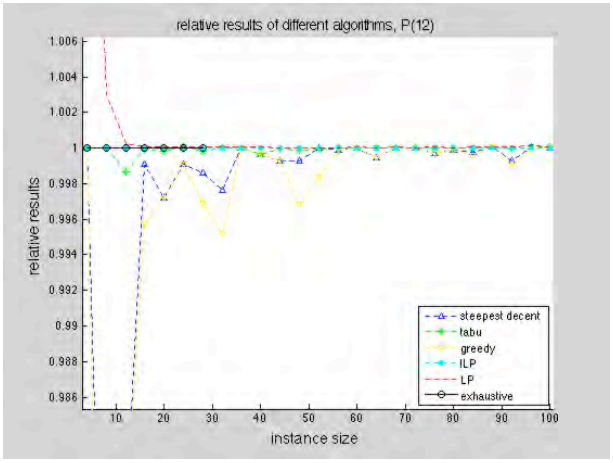


Figure 8.3 relative results of different algorithms, P(12)

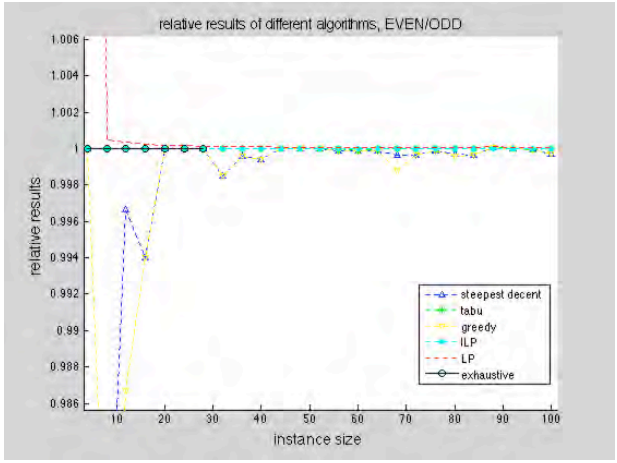


Figure 8.4 relative results of different algorithms, EVEN/ODD

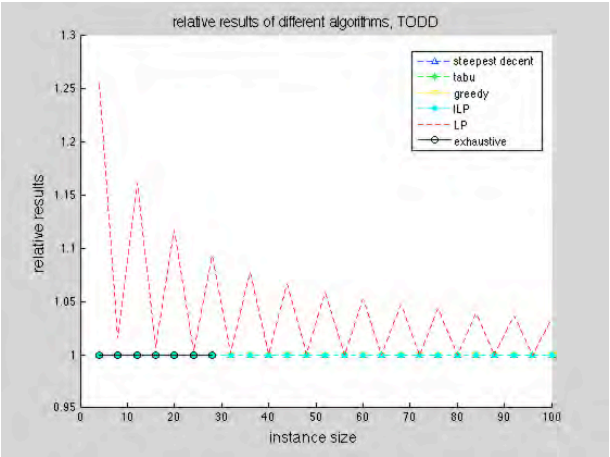


Figure 8.5 relative results of different algorithms, TODD

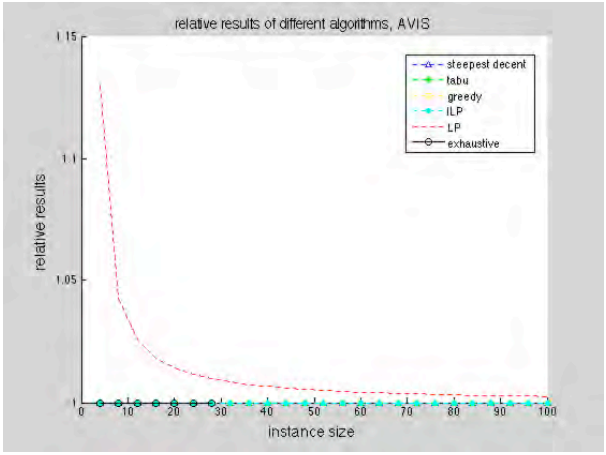


Figure 8.6 relative results of different algorithms, AVIS

From the figures above, we can find that ILP algorithm have a good performance on all problems. Tabu search algorithm also performs very well. Depeest Decent algorithm works very well on TODD and AVIS problem, and works fine with P(3), P(6), P(6) and EVEN/ODD problem. Greedy algorithm also works well on TODD and AVIS problem, but doesn't perform good on other problems. Exhaustive algorithm, as expected, works well when problem size is small but fails when problem size getting larger.

B. Runtimes

Here, we compared the runtimes of all the algorithms on our benchmark. We know that the exhaustive algorithm, Deepest Decent algorithm and greedy algorithm are independent of the instance type. So here we only tested the execution time of one problem. The Tabu search algorithm will only stop after the time limit is reached. So its execution time is also irrelevant with the instance type.

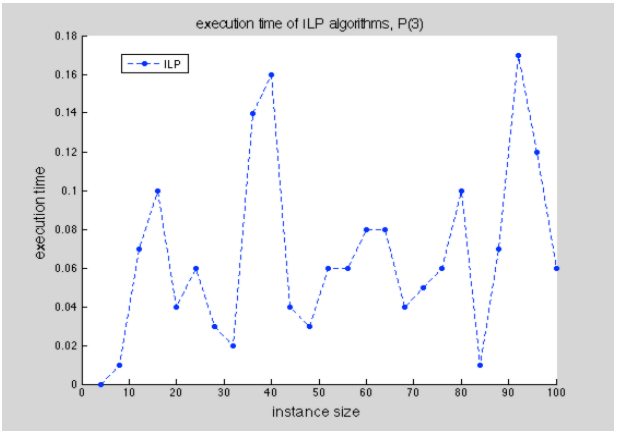


Figure 8.7 execution time of ILP algorithm, P(3)

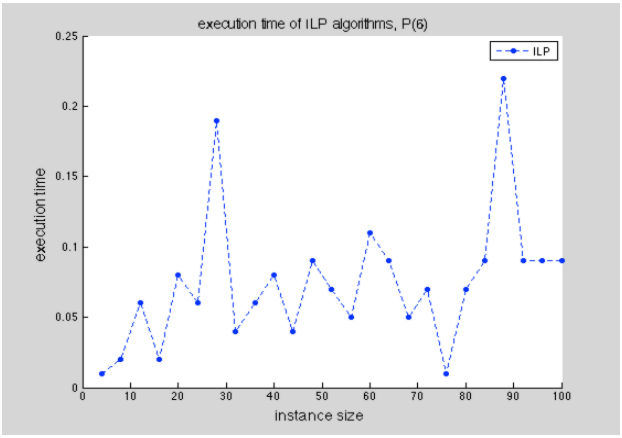


Figure 8.8 execution time of ILP algorithm, P(6)

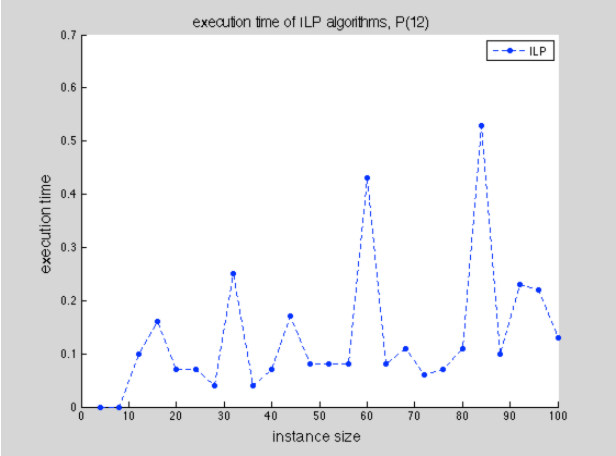


Figure 8.9 execution time of ILP algorithm, P(12)

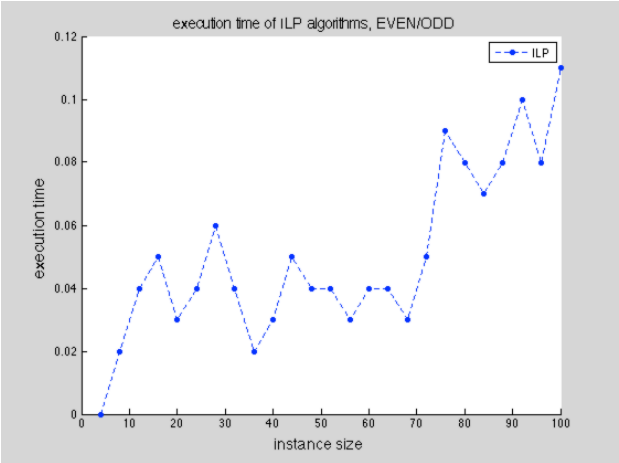


Figure 8.10 execution time of ILP algorithm, EVEN/ODD

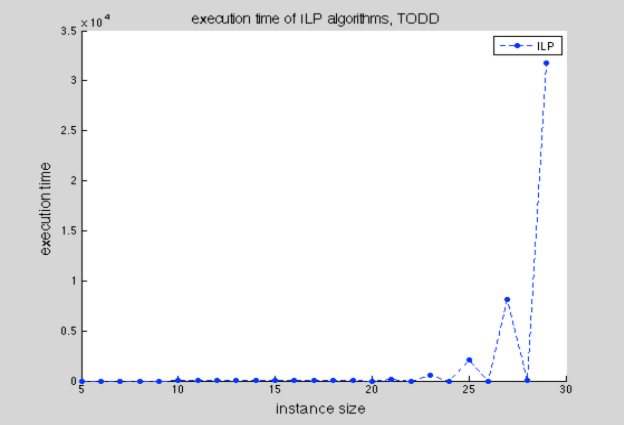


Figure 8.11 execution time of ILP algorithm, TODD

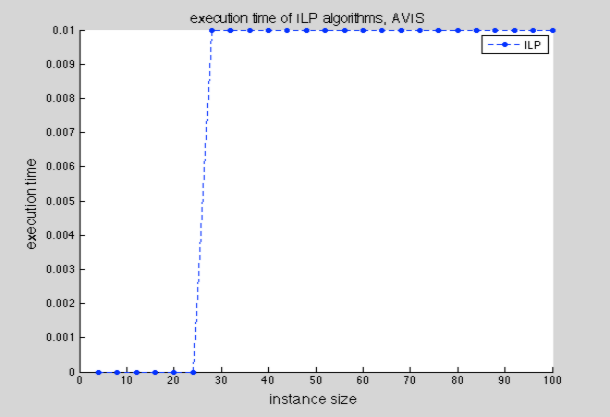


Figure 8.12 execution time of ILP algorithm, EVEN/ODD

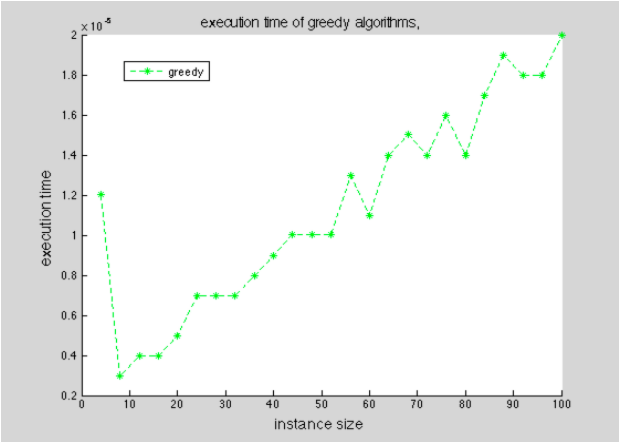


Figure 8.13 execution time of greedy algorithm

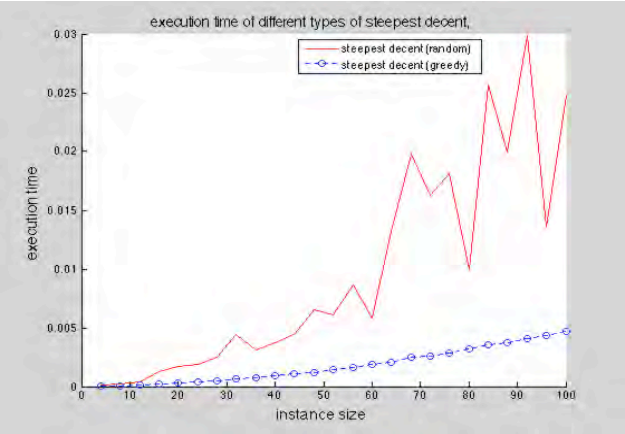


Figure 8.14 execution time of steepest decent

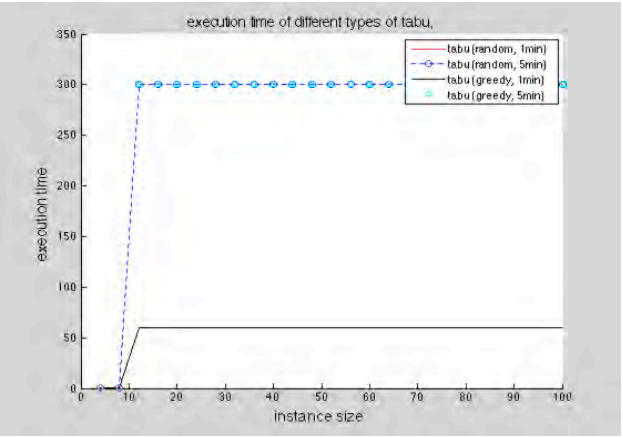


Figure 8.15 execution time of Tabu algorithm

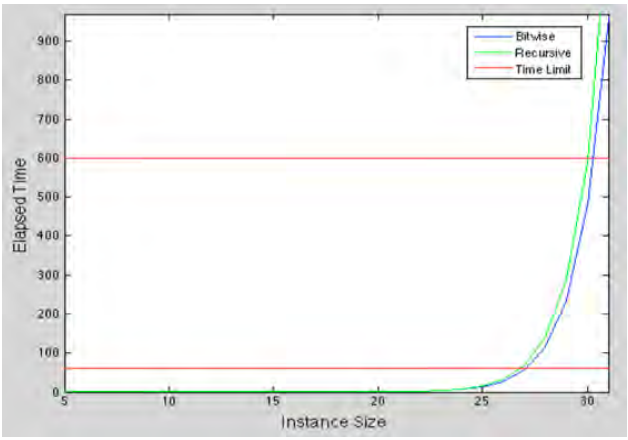


Figure 8.16 execution time of exhaustive algorithm

From the run times we tested above we can know Deepest Decent and greedy algorithm can terminate very quickly. Tabu search can always stop within the time limit, but it stops slowly. For exhaustive algorithm, the execution time grows very fast with the instance size.

The execution time of ILP algorithm depends on the instance type. For P(3), P(6) and P(12) problems, it runs a relative long time than other problems. For TODD problem, ILP algorithm runs fast for small size instance, but when the instance size become large it run a very long time to finish the branch and bound process. For AVIS and EVEN/ODD problem, ILP runs very fast.

C. Conclusion

From the analysis above we can know that ILP algorithm have a very good performance when instances' size is not large. Greedy algorithm runs very fast and has a good performance on TODD and AVIS problem but have poor performance on other problem. Tabu algorithm have a relatively good performance on all problem but will terminate only when the time limit is reached. Steepest Decent algorithm is also good for all the problems and runs fast at the same time.

In conclusion, below is our advise for selecting algorithms to solving instances of our benchmark.

	P(3)	P(6)	P(12)	EVEN/ODD	TODD	AVIS
Small Size	ILP	ILP	ILP	Tabu/ILP	Greedy/ILP	Greedy/ILP
Large Size	Tabu /Steepest Decent	Tabu /Steepest Decent	Tabu /Steepest Decent	Tabu	Greedy	Greedy

Table 8.1 Selection of algorithms for different problems

Reference

[1] Martello Silvano, Paolo Toth, Knapsack problems: Algorithms and computer interpretations. Chichester, England: John Wiley & Sons Ltd, 1990.

[2] M. Garey and D. Johnson, Computers and Intractability: A Guide to the Theory of NP Completeness, Freeman, 1997.

[3] A. Gajentaan and M. H. Overmars, On a class of $O(n^2)$ problems in computational geometry, Comput. Geom. Theory Appl. 5 (1995), 165-185

[4] V. Chvatal. Hard knapsack problems. Operations Research 28 (1980), 402-411.

[5] M. Todd. Theorem 3. In V. Chvatal. Hard knapsack problems. Operations Research 28 (1980), 1408-1409.

[6] D. Avis. Theorem 4. In V. Chvatal. Hard knapsack problems. Operations Research 28 (1980), 1410-1411.

[7] Diptesh Ghosh, Nilotpal Chakravarti, A competitive local search heuristic for subset sum problem. Computers and Operation Research, 26 (1999), 271-279.

[8] W. Diffe and M. E. Hellman, New directions in cryptography, *IEEE Tran. Inf. Theory,* IT-36 (1976), 644-654

[9] B. L. Dietrich and L. F. Escudore, More efficient reduction for knapsack-like constrains in 0-1 programs with variable upper bounds, IBM T.J., Watson Research Center, RC-14389 (1989), Yorktown Heights N.Y.

[10] Pisinger, David. "Algorithms for knapsack problems." (1995).