The Subset Sum Problem

Combinatorial optimization
Northeastern University
Spring 2017

Carlos Bocanegra Guerra
&
Fan Zhou

Introduction

Subset Sum problem is defined as “Given a set of numbers and a target T, is there any subset of them whose sum is equal to T?”. It can be regarded as a special case of Knapsack problem and has important application in complexity theory and cryptography. In this report, we systematically study the subset sum problem and implement four different techniques to solve it: Exhaustive, Greedy, ILP and Local Search.

The structure of this report is the following: Section 1 presents an implementation of the Exhaustive algorithm. It is a baseline algorithm because it can always find the optimal solution with the cost of long running time. Section 2 focus on the complexity analysis of Subset Sum problem. We also discuss the relationship between Subset Sum and some other NP-Complete problems. Section 3 introduces the Greedy algorithm and compare the performance with the Exhaustive solver. As opposed to the Exhaustive algorithm, Greedy algorithm can produce result quickly at the expense of a lower success rate. In Section 4 we introduce how to formulate the Subset Sum problem using Integer Linear Programming (ILP) and implement an ILP solver on AMPL that uses CPLEX for the solver. The results show that this is a promising way of quickly achieving solutions with good quality. In Section 5, we study how to solve Subset Sum with Local Search. We implement a Local Search solver with two solution initialization techniques (Random and Greedy) and two local search algorithms (Gradient Descent and Simulated Annealing). Finally, based on the comprehensive performance evaluation of different algorithms, we provide some insights on how to select the appropriate technique to solve Subset Sum problem.

The sections in this report are self-contained. Each section contains an independent introduction, conclusion and a reference sections. A conclusion that summary the whole report is provided in Section 6.

1. The Exhaustive Algorithm

1.1 Benchmark

The objective of our benchmark is to generate instances of different sizes that contain elements (positive integers) within a specified range. Our system guarantees the three main requirements introduced in the statement of the section:

1. The largest instance should run 20 times slower than the shortest instance.
2. The benchmark should generate at least 100 instances.
3. Some of the instances should run in times bigger than 1 and 10 minutes to provide a complete performance analysis in terms of percentage to completion.

Our benchmark aims to explore as much as possible the instance-space by (1) generating random numbers within a given range, (2) selecting the smallest subset of sizes for which the execution time differs substantially and (3) generating several instances for a given size. After analyzing the execution time for different sizes (see section "*Performance*"), we decided to confine our size scope in 11 sizes, generating 10 instances for each size. Thus, obtaining a total of 110 instances (11*10).

As for the already existing benchmarks for the Subset Sum problem, we did not have much success on finding one that substantially differs from ours, meaning that they all rely on random element generation. In [1], 7 instances with different sizes are provided.

An update of the Benchmark is presented in section 3.1, where the instance space is widely explored by controlling parameters such as the density, the instance size and the target value.

1.2 Exhaustive Algorithm

In this section, we consider the broader definition of the Subset problem, which is: "*Given an array of nonnegative integers S, and a target value T, find out all distinct subsets in array whose sum is equal to given target T. Every integer in the array can be used only once"*. In this subsection, we explore the simplest implementation of a SS solver: The Exhaustive Algorithm.

The exhaustive algorithm analyzes every single possible subset within the set and compares the sum of the elements within it with a target value T. Our implementation makes use of the binary notation, where 0/1 shows if the element is being considered or not in the current subset respectively. To that end, we implemented 2 classes:

BinaryCreator:	Allows us to convert an integer number into its binary form
ExhaustiveSolver:	Contains the method solve that uses the binary array to check what elements should be considered for the subset.

Talking about the algorithm complexity, it follows an exponential distribution $O(2^n)$. Note that, since the algorithm covers all the possibilities, the execution time does not vary depending on the minimum/maximum or the target value T.

1.3 Performance Evaluation

In this section, we analyze the execution time of the exhaustive algorithm as we increase the length of our instance. In order to select the right sizes that provides a wide enough range of results, we run our algorithm

and select the smallest range that contains the wider diversity of the execution time (with reasonable times). We establish that the range from 25 to 35 provides a wide enough spectrum of results (11 sizes in total). Furthermore, we select 10 different seeds and generate 10 instances for each size (making a total of 110 instances).

After measuring the execution time, we wonder how far the algorithm is to optimality after 1 and 10 minutes of execution time. To that end, we generate interruptions in our code and analyze the number of solutions found so far compared to the total expected solutions. Finally, we average the results among the 10 instances for each size. The results are shown in Fig. 1.

Size	1' (%)	10' (%)	Total (s)	
25	100	100	4.962	
26	100	100	10.037	
27	100	100	19.733	
28	100	100	40.405	
29	57.84	100	83.768	(1.3m)
30	27.87	100	174.928	(3m)
31	13.28	100	361.742	(6m)
32	6.27	49.98	691.380	(11.5m)
33	3.09	24.99	1425.163	(23.7m)
34	1.51	12.45	2885.267	(48m)
35	0.78	6.29	5865.265	(1.6h)

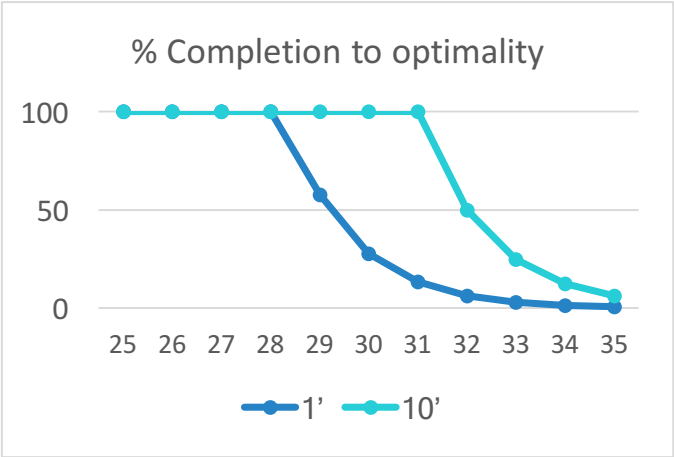

Figure 1. Completion so far after 1 and 10 minutes of execution and total execution time. Graphical representation on the right

1.4 References

[1] "*Data for the Subset Sum Problem*"
https://people.sc.fsu.edu/~jburkardt/datasets/subset_sum/subset_sum.html

2. Complexity Analysis

In Section 2, we study the natural subproblems of our problem (SS) and the problems for which our problem is a natural subproblem. We also provide a comprehensive explanation of the time complexity and algorithms that can be used to solve the problems in P.

The subproblem analysis has a two-side approach. First, it provides a proof that the problem that we are trying to characterize is an NP-Complete problem by finding a subproblem of it that is NP-Complete. Second, by finding subproblem we may come across with a subproblem of our NP-Complete problem that is in NP, hence having a solution that runs in polynomial time.

[1] is Taken as the baseline for our analysis since it contains a broad and complete explanation of subproblems and the inter-connection between problems belonging to different classes. However, not many examples are provided in the book. That is why we extended our horizons by surveying the ACM data base.

Definition of the Decision Version of the Subset Sum (SS) problem [1]
INSTANCE: Given a Finite set A, size $s(a) \in Z^+$ for each $a \in A$; and a positive integer B.
QUESTION: Is there a subset $A' \subseteq A$ such that the sum of the sizes of the elements in A' is exactly B?

2.1 Natural NPC Problems of SS

Definition of Subproblem [1]
A problem $\Pi' = (D', Y')$ is a subproblem of $\Pi = (D, Y)$ if the instances in Π' are contained in Π ($D' \subseteq D$) and the yes-instances in Π' belong to the intersection between the yes-instances in Π (Y) and D' ($Y' = Y \cap D'$).

2-SUM

Problem Definition

INSTANCE: Given a Finite set A, size $s(a) \in Z^+$ for each $a \in A$; and a positive integer B.
QUESTION: Is there a subset $A' \subseteq A$, whose cardinal is 2, sum of the elements in A' is B?

Proof of subproblem

Clearly 2-Sum is a special case of subset sum with target $B = 0$. More concisely, there is a yes instance of 2-Sum problem if and only if it is also a yes-instance for the corresponding subset sum problem (with target $B = 0$). Therefore, 2-Sum is a subproblem of subset sum.

Time complexity

We provide an $O(N)$ algorithm in following section. Thus, 2-Sum problem is in P.

K-SUM

Problem Definition

INSTANCE: Given a Finite set A, size $s(a) \in Z^+$ for each $a \in A$; and a positive integer B and K.
QUESTION: Is there a subset $A' \subseteq A$ such that its cardinal is K and the sum of the elements in A' is B?

Proof of subproblem

K-Sum is a just a generalize version of 2-Sum. In other words, 2-Sum is a subproblem of K-Sum. We can proof K-Sum is a subproblem of subset sum with the similar proof as above.

Time complexity

If K is even, the time complexity is $O(nk/2log(n))$.
If K is odd, the time complexity is $O(n(k+1)/2)$.
The full proof can be seen in [2] and [3]. Thus, K-Sum problem is in P.

Subset Sum with Repetitions Problem (SRP) [4]

Problem Definition

INSTANCE: Given a Finite set A, containing repeated elements, size $s(a) \in Z^+$ for each $a \in A$; and a positive integer B.

QUESTION: Is there a subset $A' \subseteq A$ such that the sum of the sizes of the elements in A' is exactly B?

Proof of subproblem

The SRP can be seen as a restricted problem from SS. In SS the integers can be either repetitive or non-repetitive. SRP is a special case of SS as it only considers the set with repetitive integers. Therefore, SRP is a subproblem of SS.

Time complexity

The complexity of SRP is NPC. First SRP is clearly a NP problem as we can verify the yes-instance in polynomial time. Assuming the input set of SS is A. We can easily reduce SS to SRP by constructing a new set $A' = A \cap x \cap (-x)$, where x is an existing element in A. So, A' must contain at least one repetitive element (x). If there is a yes-instance for A, then there must also be a yes-instance for A' by adding x and $-x$. As SS is a NPC problem, SRP must also be NPC.

2.2 NPC problems for which SS is a natural subproblem

Kth Heaviest Subset Sum Problem (HSP) [4]

Problem Definition

INSTANCE: Given a Finite set A, size $s(a) \in Z^+$ for each $a \in A$; and positive integers B and K.

QUESTION: Are there at least K subsets $A' \subseteq A$ for which the sum of its sizes is at least B?

Proof of subproblem

The SS can be used to solve the HSP problem by employing what we call dynamic programming. All the algorithm needs to do is call SS repeated times for each B and K being considered and keep increasing these values, since the problem states "*at least*".

Time Complexity

The problem is NPC since it is a subproblem of it (SS) is NPC.

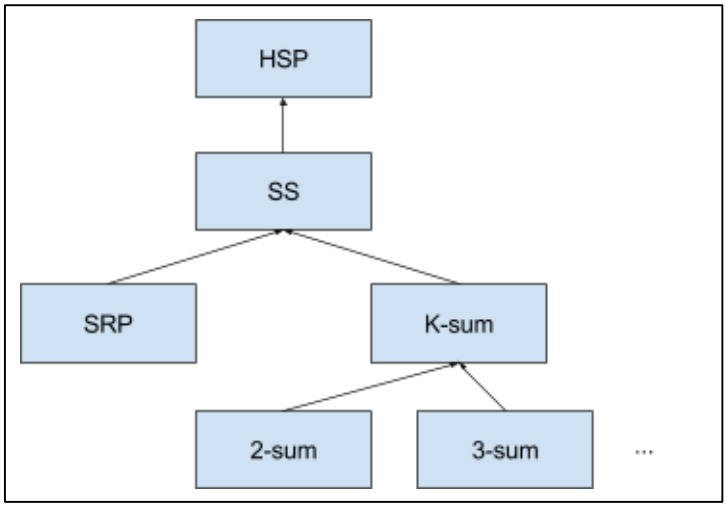

Figure 1: Subproblem relationships between SS and other related problems

2.3 Relationship among subproblems involving SS

The subproblem relationships with SS is shown in Fig. 1. SS is the subproblem of HSP, and both are NPC problems. SRP and K-sum are subproblems of SS, where SRP is NPC and K-sum is P. K-sum also have infinite number of subproblems with certain value of constant K, where all of them are P problems. Since K-sum is P and there appears no known problem that contains K-sum as a subproblem that is in P, we believe K-sum is a maximally polynomially solvable problem. There is no minimally NPC problem in the above figure since there is no NPC problems in the Fig.1 without NPC subproblems.

2.4 Algorithm to solve subproblems of SS that are in P

We provide a O(n) algorithm for solving 2-sum problem (Natural subproblem of SS in P):

```
TWO-SUM(S, B)
      // A is the input set, B is the target number
      h = hashmap
      for s in A
         if (B - s) is in h
            // find a solution
            return true
         else
            h <- s
            return false
```

The idea is to simply iterate over every element in the set. For every newly find element s, we check to see $B - s$ has existed in the hashmap. If so then we find two numbers that added up to be B, and return true. Otherwise we put s in the hashmap and continue iterating. We return false if all element in the set has been checked and no such solution found. Since the finding operation for a hashmap takes $O(1)$, the total time complexity of two sum is $O(n)$, where n is the size of input set.

2.5 Conclusions

In this report, we survey several problems (SRP, 2-SUM, K-SUM and HSP), specifically focus on their relationship with SS. Among these problems, SRP, 2-SUM and K-SUM are subproblems of SS, while SS is the subproblem of HSP. The relationship is depictured in Fig.1. We also introduce their complexity and claim that K-SUM is the maximally polynomially solvable problem and there appears to be no minimally NPC problem. Finally, we provide a simple O(N) algorithm to solve 2-SUM.

2.6 References

[1] Michael R. Garey and David S. Johnson. 1990. Computers and Intractability; a Guide to the Theory of Np-Completeness. W. H. Freeman & Co., New York, NY, USA.
[2] Nir Ailon and Bernard Chazelle. 2004. Lower bounds for linear degeneracy testing. In Proceedings of the thirty-sixth annual ACM symposium on Theory of computing (STOC '04). ACM, New York, NY, USA, 554-560. DOI: http://dx.doi.org/10.1145/1007352.1007436
[3] Jeff Erickson. 1995. Lower bounds for linear satisfiability problems. In Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms (SODA '95). Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 388-395.
[4] J.L. Ramírez Alfonsín, On variations of the subset sum problem, Discrete Applied Mathematics, Vol. 81, Issue 1, 1998, pp 1-7, http://dx.doi.org/10.1016/S0166-218X(96)00105-9. (http://www.sciencedirect.com/science/article/pii/S0166218X96001059)

3. The Greedy Algorithm

The exhaustive algorithm has proven to always find the optimal solution regardless the instance size and the target value. The high success rate is achieved at the expense of the unreasonable execution time. Since the algorithm not only has to find the solution to a problem but also solve it within a reasonable time, we investigate the applicability of the greedy approach for the Subset Sum in this section. The most remarkable features of Greedy algorithms are

1. Always selects the best option available.
2. Never backtracks.

This section is organized as follows: In section *Benchmark update* we introduce some of the modifications to our Benchmark that were suggested in Section 1 by the professor, mainly spanning the instance space by including the instance density as a parameter. In section *Greedy Algorithm*, we go over the Greedy algorithm for the Subset Sum problem. In *Performance Evaluation*, the results in terms of success rate and execution time are provided along with a comprehensive explanation of the algorithm behavior. Finally, the *Conclusion* section summarizes the main findings on the applicability of the Greedy algorithm to the Subset Sum problem.

3.1 Benchmark update

In Section 1 we proposed a Benchmark that tries to expand the instance space as much as possible by exploring different instance size and generating several instances for each size. This two parameters were enough to evaluate the performance of the exhaustive algorithm, whose behavior is not affected by parameters such as the density of the set or the target value.

With the aim to make our Benchmark more robust, we include an extra degree of complexity by tackling the density of the instance. Some definitions of the density are proposed in [1] and [2], where the authors analyze the success rate for solving the Subset Sum problem for different instance density. In all of them, the instance density is defined as a relationship between the cardinality of the set and the maximum value in the set. In our section, we define the instance density as $density = |N|/\max(N)$, where N is the set and $|N|$ is the cardinality of set N. As a result, our Benchmark generates 110 instances in total by analyzing 11 different instance size from 25 to 35 and having 10 different instance density for each size.

3.2 Greedy Algorithm

The idea of greedy algorithm for solving subset sum is very straightforward. We first sort all integers in descending order. Then we pick the integers one by one and check if the sum of picked integers is equal with the target number. If so then we have already successfully found a solution and return true. Otherwise if the sum is larger than the target then it means the last picked integer is too large, we just take that integer out and continue the process. We return false if no solution can be found after we iterate over all integers in the array.

```
Input:   An array A (instance set) and a target number T
Output:  True if solution is found; False if no solution is found

procedure SSGreedy (A,T)
      A.sort()
      for i = A.length-1; i >= 0; i-- do
            if  T == A[i] then
                  return true
            else if  T > A[i] then
                  T -=  A[i]
      return false
```

3.3 Performance evaluation

In this section, we evaluate the impact of certain parameters of the Benchmark and the target value on the performance of the algorithm in terms of success rate and execution time. As opposed to the exhaustive algorithm, the Greedy algorithm will see its performance affected by the target value since it the algorithm stop its execution as soon as it finds a solution. In other words, the lower the target value is, the faster the Greedy algorithm would be able to find a solution (if any).

Impact of the Instance Density

The Table 1 reflects the rate success as a function of the instance density. It is not hard to see that for instances with low density, the performance of the greedy algorithm drops. This is because there are more chances to have larger gaps between the elements in the set and thus, easier for the Greedy algorithm to select the wrong elements in the candidate set and exceed the target value. For instance, the instance $\{15,10,10,33,47\}$ with density 0.1 is not likely to be solved by the Greedy algorithm. On the other hand, the instance $\{3,4,1,3,2\}$ with density 1 will be most likely solved by the Greedy algorithm. Note that both have instance size 5.

Density	Rate of Success
0.1	21
0.2	34
0.3	45
0.4	56
0.5	63
0.6	71
0.7	77
0.8	80
0.9	82
1.0	85

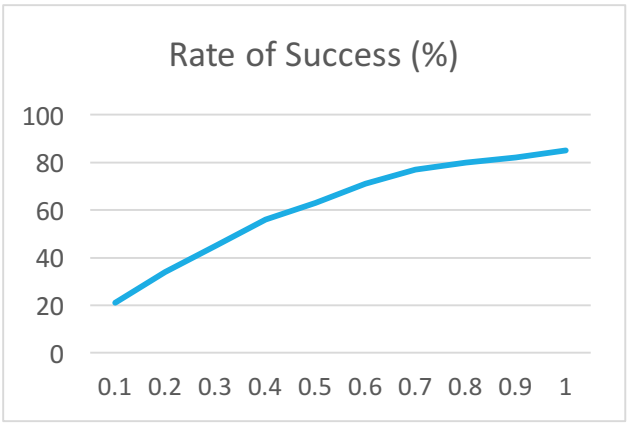

Table 1. Performance of the Greedy Algorithm for different instance densities

Impact of the Target Value

In order to evaluate the impact of the Target value, we define a parameter alpha to control where in the spectrum of possibilities should the target be located. The spectrum of possibilities covers the possible targets from the minimum value (Best-Case-Scenario or BCS) to the sum of all the elements in the set (Worst-Case Scenario or WCS). Therefore, the target is defined as a function of alpha: $Target = \alpha \cdot BCS + (1-\alpha) \cdot WCS$

Table 2 summarizes the performance in terms of success rate of the Greedy algorithm for different values of alpha that determine the target. There is a slight increasing trend as alpha gets close to 1 (minimum value). That can be explain because there are more chances to find lower values in the set than higher. In addition, for alpha equal to 1 the success rate is 100% because the Greedy algorithm will always find the solution for the Worst-Case-Scenario in terms of execution time, which is the sum of all the elements in the instance.

Alpha	Rate of Success	Alpha	Rate of Success
0.0	99	0.50	63
0.05	58	0.55	62
0.10	56	0.60	65
0.15	60	0.65	61
0.20	59	0.70	64
0.25	59	0.75	59
0.30	61	0.80	68
0.35	61	0.85	59
0.40	63	0.90	70
1.45	62	0.95	67

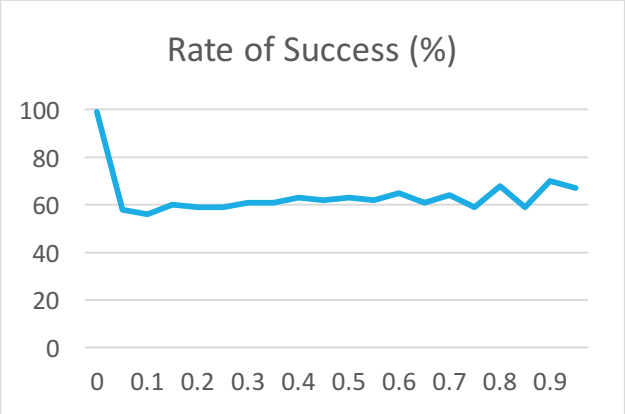

Table 2. Performance of the Greedy Algorithm for different alpha

Impact of the Instance Size

As stated in previous sections, the Greedy algorithm considerably reduces the execution time at the expense of the decrease of the success rate. Table 3 shows the increase of the execution time as a function of the instance size. Note that even for sizes close to 500, the execution time is still in the order of milliseconds. Recall that the exhaustive algorithm would take hours when the instance size is bigger than 35.

Size	Execution Time (ms)
5	1
55	3
105	5
155	7
205	10
255	13
305	15
355	18
405	22
455	24

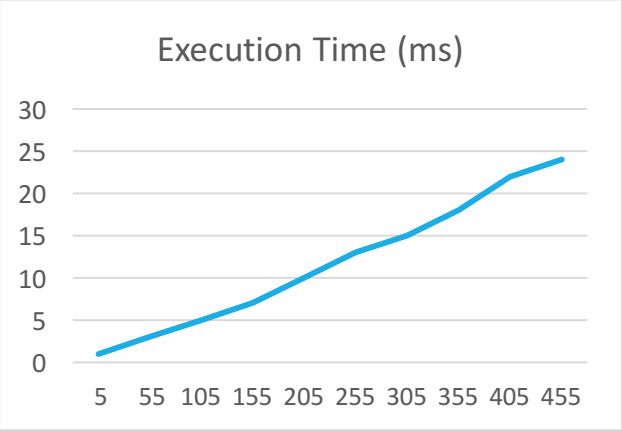

Table 3. Execution time of the Greedy algorithm for different instance sizes

3.4 Conclusion

In Section 3 we implemented and evaluated the performance of the Greedy algorithm for the Subset Sum problem. After including some suggestions made by professor in previous section reports, we widen the instance spectrum of the benchmark by considering several instance densities across our instance set. Subsequently, we evaluated the success rate as a function of the instance density and target value; and the execution time as a function of the instance size. Our findings can be summarized into 3 observations. First, the success rate is higher as the instance density gets higher, meaning that the inter-element gaps in the instance set is smaller. Second, the success rate is not much affected by the target value other than the Worst-Case-Scenario (sum of all the elements), for which the Greedy algorithm always find a solution. Finally, the execution time is considerably smaller compared to the exhaustive algorithm.

3.6 References

[1] J. C. Lagarias and A. M. Odlyzko, "Solving low density subset sum problems," 24th Annual Symposium on Foundations of Computer Science (sfcs 1983), Tucson, AZ, USA, 1983, pp. 1-10.
[2] Changlin Wan and Zhongzhi Shi, “Solving Medium-Density Subset Sum Problems in Expected Polynomial Time: An Enumeration Approach,” CoRR, 2007, http://arxiv.org/abs/0712.3203

4. Integer Linear Programing

In Section 4 we widen the spectrum of possibilities to solve our Subset Sum problem by introducing a new technique: Integer Linear Programming (ILP) [1]. We are asked to implement an ILP solver in AMPL [2], to evaluate the effectiveness of this method and compare it with the exact and greedy algorithm. As to provide consistency among the instance generated in Java, some modifications were made in the core code to generate the AMPL inputs, execute the AMPL script, retrieve the results and parse them in a centralized fashion.

As mentioned in earlier reports, not all the instances generated contain a feasible solution for the specified target. In that case, the objective is to minimize the gap between the optimal solution and the solution found by the solver. A good implementation of a solver is the one that minimizes such gap, the so called tighter gap. In this section, we provide a comprehensive analysis of the gaps between the LP lower bound and the ILP and Greedy solutions as a function of our instance (target value, instance density and instance size).

Finally, the success rate on finding an optimum solution by the AMPL solver is evaluated as a function of the instance size. We restrict the execution time in AMPL and vary the instance size to evaluate the Success Rate.

4.1 ILP formulation for subset sum

For subset sum problem we are given a set of integer numbers W = {w_1, w_2...w_n} and an integer number S, and need to answer there exist a subset of W such that the sum of its elements is equal to S. We define an n-elements vector X = {x_1, x_2...x_n}, where x_i indicates w_i is selected. Now we can formulate the subset sum into following ILP problem:

$$\max \quad Z = \sum_{j=1}^{n} w_j \cdot x_j$$

$$\text{s.t.} \quad \sum_{j=1}^{n} w_j \cdot x_j$$

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

To get LP formulation we only need to remove the last binary constraints. It is worth mentioning that for our problem, the LP lower bound will always be the *Target* value S. The constraints in the ILP formulation can always be relaxed in such a way that the assignation vector does not necessarily have to be binary. This way, our LP will always find a combination whose sum is the target.

4.2 AMPL Code for ILP

The AMPL section is composed by 3 main scripts:

- *Runnable.run*: This script fetches the data for the instance, executes the model, retrieves and outputs the results. Since we are mostly interested in the actual value, the output is the summation of the products between the assignation vector (binary value) and the value in the instance vector.
- *Model.mod*: The model contains the actual formulation of the ILP for the Subset Sum problem:

  $\max \quad Z = \sum_{j=1}^{n} w_j \cdot x_j$

  $\text{s.t.} \quad \sum_{j=1}^{n} w_j \cdot x_j$

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

Where *x* reflects the assignation vector and *w* represents the values in the instance.

- *Inputs.data:* Contains the target value, the instance size and the actual instance (values per index in a vector format). Since several instances are generated, each instance and its configuration are stored in one file. The files are retrieved by the runnable script at the beginning of each execution.

4.3 Java - AMPL Architecture

The core of the code runs in Java, where the instances are generated and the results from the Exhaustive and the Greedy solver are obtained. With the aim to centralize the execution and analysis on one platform, we developed a bridge between Java and AMPL so that these two platforms can communicate and generate compatible Inputs and Outputs together. Thus, automatizing the whole process and guaranteeing a stable comparison between the solutions. In summary, our code takes the following steps:

1. Generate *maxInst* Instances and AMPL input data for each of them.
2. Call the Exhaustive and Greedy Solver and obtain their final value for every instance.
3. Call the ampl code using the Runtime class, specifying the name of the runnable script.
4. Retrieve the output for each execution (instance) and parse it to obtain the final value.
5. Evaluate performance metrics for the three solvers and calculate gaps between them.

4.4 Performance evaluation

In this section, we analyze the results for the ILP and Greedy solver and evaluate the performance in terms of the Rate of Success (ROS), defined as the ratio at which the solvers are able to find the optimal solution. As discussed in the previous report, the main parameters that define the success rate of our solvers are:

1. The instance size: The cardinality of the instance set.
2. The instance density: The density is defined as the instance size over the maximum value within our instance. In our case, we assume that the minimum is always 1. Thus, the density is $|N|/\max(N)$, where *N* is the instance set.
3. The target goal: The goal needs to be selected accordingly to the instance size and density. The target is always lower bounded by the minimum value in the set ($\min(N)$), also referred as Best-Case-Scenario (BCS) in terms of complexity, and upper bounded by the sum of all the elements in the set ($\mathrm{sum}(N)$), also referred as Worst-Case-Scenario (WCS). For simplicity, we define the parameter α as to move within the bounds and select the target value: $Target = \alpha \cdot BCS + (1-\alpha) \cdot WCS$.

The observations of the ROS for the solvers allows us to define the conditions under which our solvers are not able to find the optimal solution. We confine ourselves to that subset of instances and provide some analysis on the gaps between LP bounds and the solutions provided by the solvers.

Impact of the Instance Density

In the previous report, we stated that the ROS for the Greedy algorithm gets better as the instance density increases. We widen the conclusions on this matter by analyzing the behavior for several target values while also provide the results for the ILP formulation.

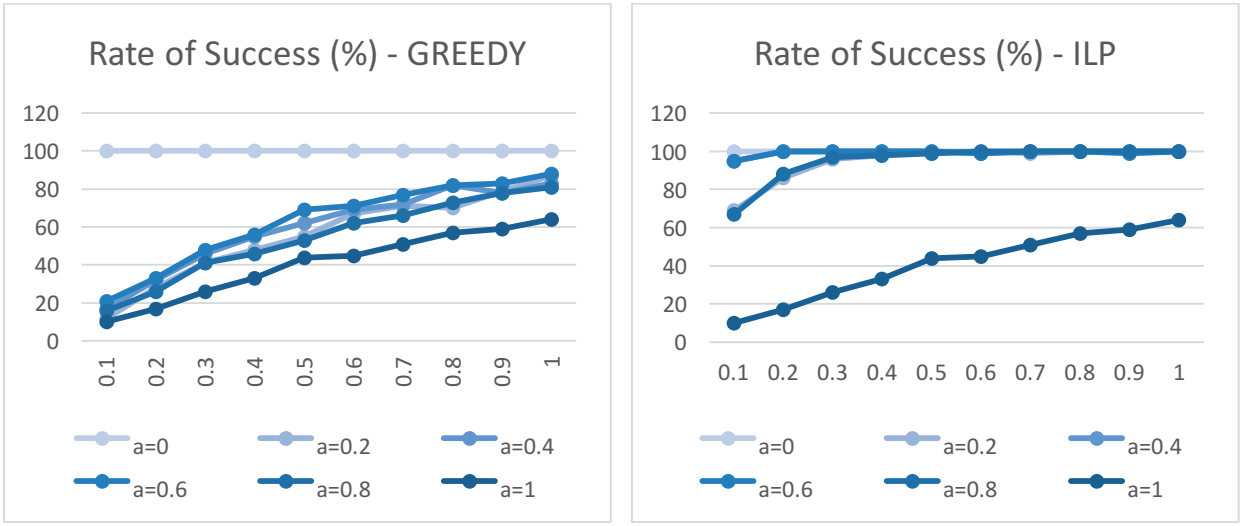

Figure 1. Impact of the density on the ROS for the Greedy and ILP solvers. The results are shown for different values of alpha (low alpha: target close to the minimum; high alpha: target closer to the sum of elements)

Fig. 1. Summarizes the main findings on this regard. We cover the whole spectrum of possibilities densities and evaluated the ROS for some subset of target values. The first observation is that when α is 0, the solvers always find the target. This is obvious since $\alpha = 0$ maps to $Target = Sum\ of\ elements$. The second observation is that the Greedy algorithm barely improves its performance by selecting a lower α (other than 0) whereas the ILP solver finds more solutions as α gets lower. Here, no time limitation is imposed to the ILP solver, meaning that if a feasible solution exists for the instances, the ILP solver will find it. The third observation is that the low ROS for the ILP solver is obtained when the density is low (also for the Greedy algorithm).

By way of example, Fig. 2 shows a comparison between the ROS of the solvers when the instance size is 10 and α is 0.5.

Density	GR	ILP
0.1	18	92
0.2	34	97
0.3	43	99
0.4	50	99
0.5	61	100
0.6	69	99
0.7	73	99
0.8	83	99
0.9	83	99
1	88	100

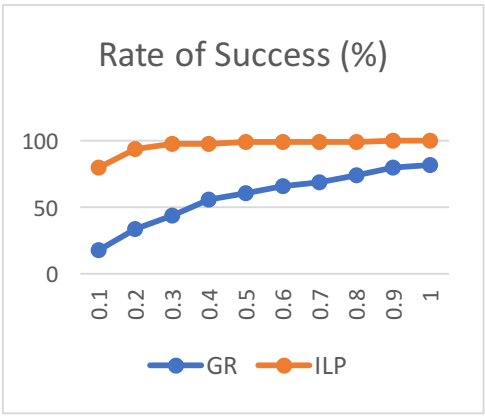

Figure 2. ROS comparison between the Greedy and ILP solvers. Instance size is 10 and α is 0.5

Impact of the Target Value

Recalling the findings included in our previous report, the performance of the Greedy algorithm is not much impacted by the target value. In Fig 3. we extend that analysis by showing the trend for different densities. We

observe that for higher densities, the Greedy algorithm has more chances to find the optimal solution regardless of the target value selected.

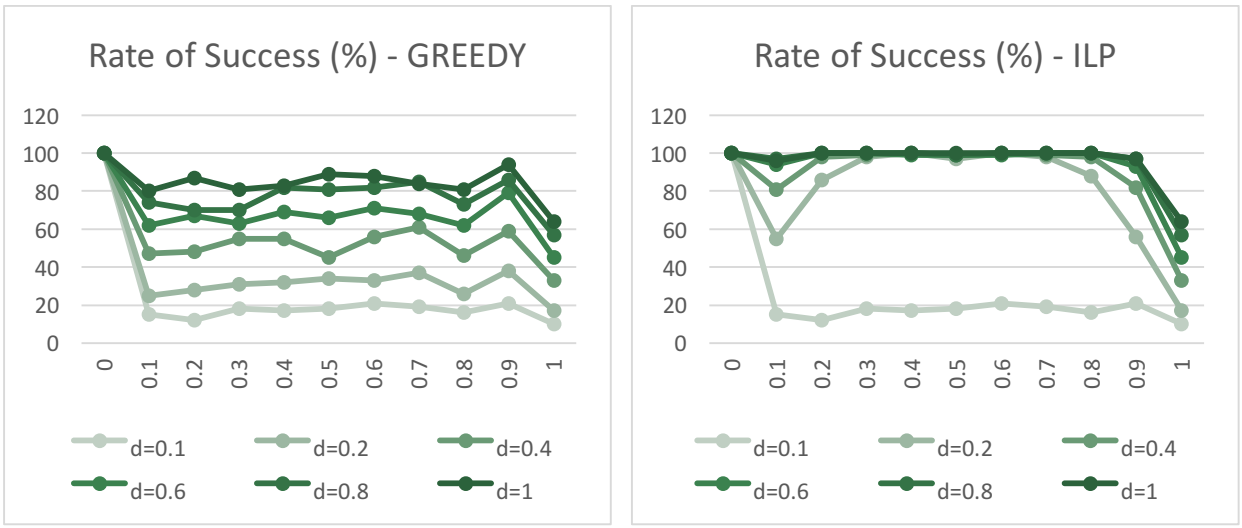

Figure 3. Impact of the target value on the ROS for the Greedy and ILP solvers. The results are shown for different instance sizes

We performed the same analysis on the ILP side. The first observation is that for low densities (0.1 or lower), the ILP performs closely to the Greedy algorithm. This is because few instances contain a feasible solution to the target. As the density gets higher, the chances to reach the target value increases. The second observation is that the lowest ROS is concentrated in α's around *0.1-0.3* and *0.7-1.0*, which means that it is hard to find a solution if the target is closer to either the minimum or the sum of all the elements. The final observation is that the ROS tends to flat out as the density gets higher.

By way of example, Fig. 4 shows a comparison between the ROS of the solvers when the instance size is 10 and the density is *0.5*.

Alpha	GR	ILP
0	100	100
0.1	54	85
0.2	57	99
0.3	59	100
0.4	61	100
0.5	61	100
0.6	65	100
0.7	65	100
0.8	60	99
0.9	64	85

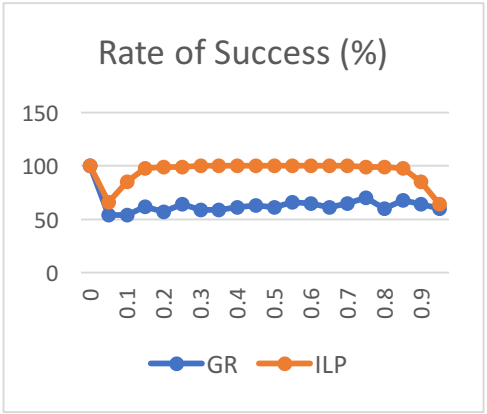

Figure 4. ROS (%) comparison between the Greedy and ILP solvers. Instance size is 10 and density is 0.5

Analysis of the GAPs between solutions (Lower Bound, ILP and Greedy)

It is of our interest to determine how effective is the ILP solver that we implemented. As we have seen in the previous sections, one approach is to analyze the ROS of each solver. A complementary analysis is to analyze

how far the results are from the target value for the instances that no exact solutions were found. In this section, we analyze the gaps between the solvers defined as $gap(\%) = 100 \cdot (Target - solver)/target$.

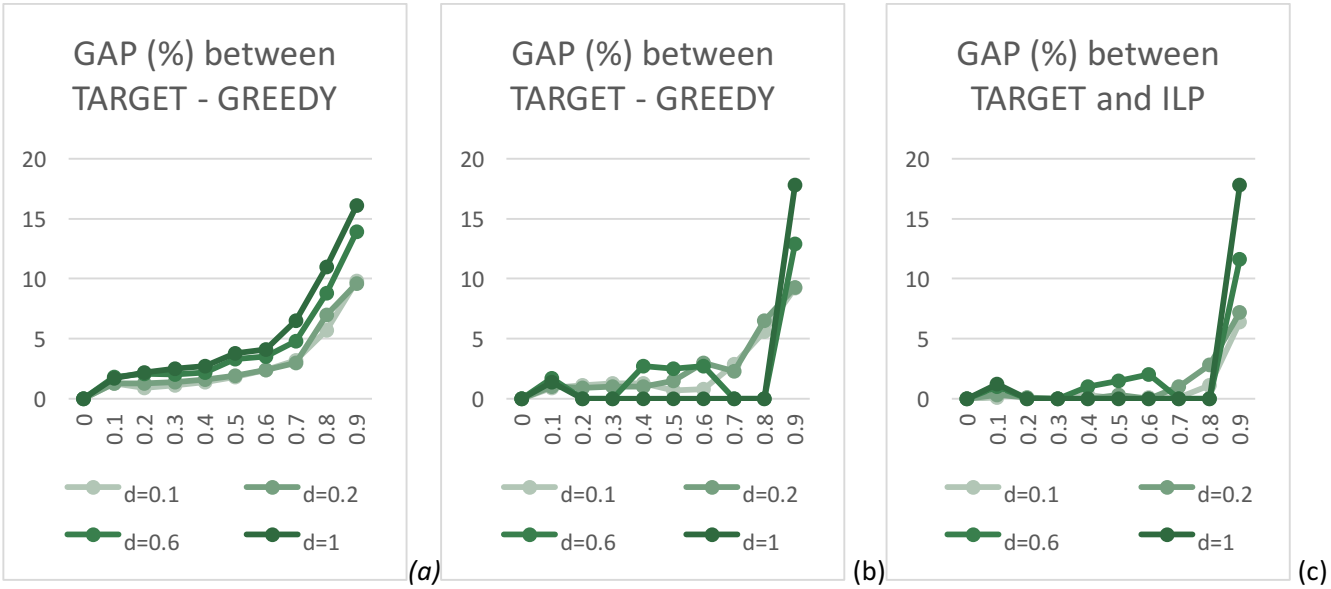

Figure 5. (a) shows the target-greedy gap for those instances that ILP didn't match the target. (b) and (c) show the target-greedy and target-ILP gap respectively for those instances that ILP did match the target

First, we analyze the gaps between the solvers as a function of the target value as shown in Fig. 5. Fig. 5(a) shows the gaps between the Target and the greedy solver for the instances that ILP solver has found the solution but the Greedy hasn't. We observe that as α approaches to 1 (minimum in the instance), the gaps get bigger. In addition, the gaps tend to be bigger as the instance density gets bigger. Fig. 5(b) and (c) shows the gaps between the target-Greedy and target-ILP solutions respectively the for those instances that ILP solver couldn't find the solution. We observe that the results pretty much follow the same trend with slight differences in the absolute values, ILP giving always a lower gap.

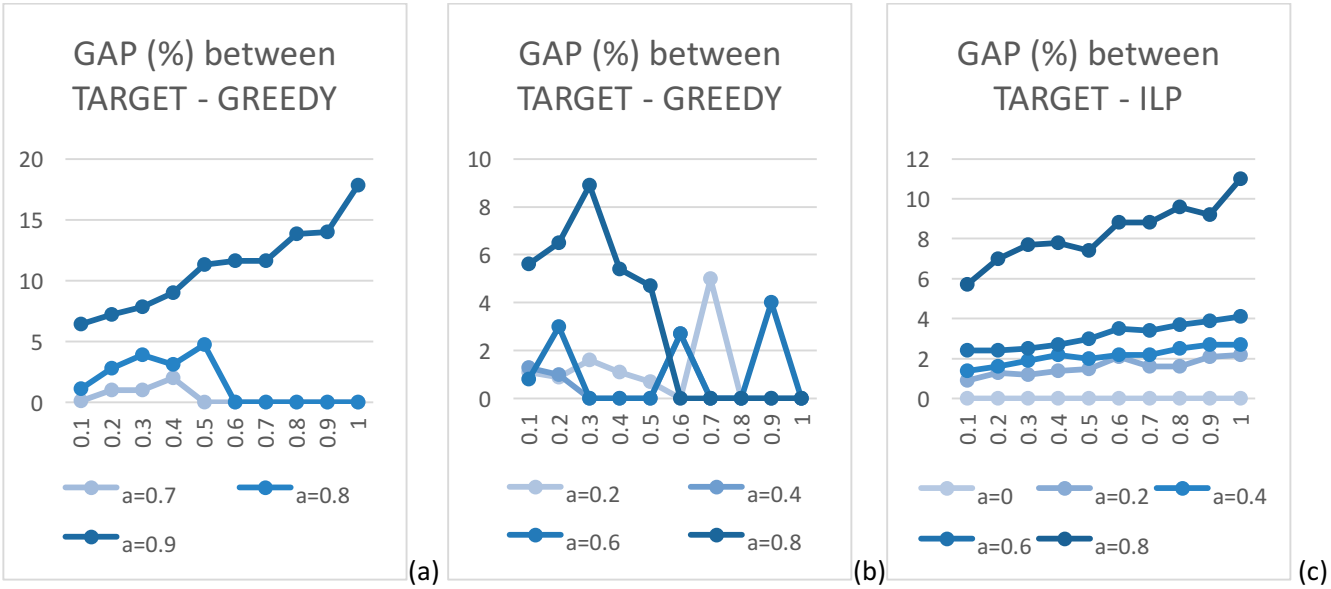

Figure 6. (a) shows the target-greedy gap for those instances that ILP didn't match the target. (b) and (c) show the target-greedy and target-ILP gap respectively for those instances that ILP did match the target

Finally, we analyze the gaps as a function of the instance density. The distribution of Figures in Fig. 6 follows the one shown in Fig. 5. Since the ILP solver had difficulties finding the target values when α is high (Fig. 3), we analyze the gaps using for the greedy solver in those scenarios. Fig. 6(a) shows the target-greedy gaps, revealing that the trend is much correlated with the ROS of the ILP. In other words, for low ROS in the ILP the gap that Greedy gets is higher than for the scenarios where ILP gets a high ROS. Fig. 6(b) reveals that the gap gets higher as the density increases for target-greedy. Regarding the target-ILP gaps, we observe that the trend is monotonous increasing along the *x* axis (density), while also the gap increases as the target value gets closer to the minimum (low ROS for ILP). This result is aligned with Fig 6(a).

Analysis on the Rate of Success for ILP limiting the execution time

We have stated that, when the time execution is not limited in AMPL, the ILP always finds the optimum value. In this section evaluate the ROS when the execution is limited to a fixed time. To that end, we first need to select the right subset of instances guaranteeing that the conclusions are consistent. In other words, we need to create instances for which ILP would find the target value if no execution time is set. As we have seen in Fig 3(b), ILP has a ROS of 100% when the target value lies between the WCS and the BCS ($\alpha = 0.5$) and when the density is high (*0.4* or higher). We use these parameters to configure our environment.

Fig. 7 shows the ROS trend when we fix the execution time to be 50ms and we vary the instance size between 6500 and 7000. The simulations can be replicated using a different value for the execution time, with a proper adjustment on the instance size range.

Size	ROS (%)
6500	100
6600	96
6700	84
6800	48
6900	30
7000	4

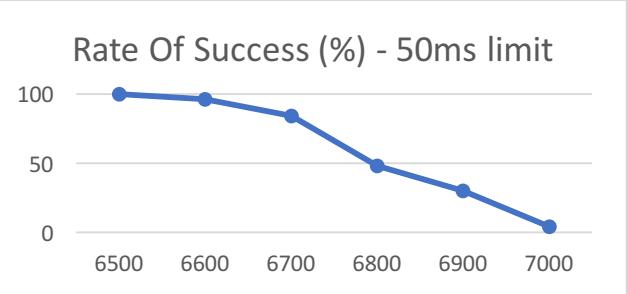

Figure 7. Progression of the Rate of Success (%) when the execution time limit is set to be 50ms

4.5 Conclusion

In this section, we became familiar with a powerful tool to solve problems ILP and LP problems: AMPL. We evaluated the Rate of Success of the Subset Sum problem given different instance configurations such as the instance density, instance size and the target value, and compared the performance with the Greedy solver. We analyzed the configurations for which the target value cannot be reached in our instances, which correspond to a low density (between *0.1* and *0.5*) and a target close to the WCS (between *0.7* and *1.0*) and close to the BCS (between *0.1* and *0.3*). The ILP solver has proven to always find the target value is a feasible combination was possible, as well as to always minimize the gap in case no feasible combination was possible. Our analysis is complemented by an evaluation on the gaps between the Lower Bound (Target for Subset Sum problem) and the Greedy and ILP solvers. We observed that the gaps get bigger as the instance density increases and the target value gets close to the WCS. Finally, we evaluate the performance of our ILP when the execution time is limited.

4.6 References

[1] Nemhauser, G. L. and Wolsey, L. A. Integer and Combinatorial Optimization. New York, 1999.
[2] AMPL ILOG AMPL CPLEX System Version 9.0 User's Guide, 2003

5. Local Search

In this section, we study how to solve subset sum problem with local search techniques. The main idea of local search is heuristically generating neighboring solutions from the initial solution, and probabilistically decides whether moving to a new solution or not. The above process is repeated until the system reaches a state that is good enough for the application, or until a given computation budget has been exhausted. Different local search algorithms have different ways of deciding whether to accept a certain neighbor solution or not. In this experiment, we choose two local search heuristics: gradient (steepest) descent and simulated annealing. The difference between the two algorithms is that the gradient descent always chooses the best from neighbor solutions, while the simulated annealing may accept bad solution with a certain probability.

This section is structured as follows. First, we introduce the common process of local search algorithm. Second, we introduce the definition of neighbor structure and discuss ways of generating initial solutions and the two local search algorithms. Finally, we evaluate the performance of local search algorithms and present the experimental results.

5.1 Local Search Structure

Local search algorithm usually consists following four steps [1]:

1. Initialization: Choose an initial subset S to be the current solution and compute the value of the objective function F(S).
2. Neighbor Generation: Select a neighbor S' of the current solution S and compute F(S').
3. Acceptance Test: Test whether to accept the move from S to S'. If the move is accepted, then S' replaces S as the current solution; otherwise S is retained as the current solution.
4. Termination Test: Test whether the algorithm should terminate. If it terminates, output the best solution generated; otherwise, return to the neighbor generation step.

Different local search algorithms can share step 1 and 2, but the acceptance and termination test are usually different. In the following sections, we present a comprehensive explanation of these steps.

5.2 Initial Solution Generation

We consider two approaches as for the initialization: Random and Greedy. For the Random initialization, we simply generalize a random value within range *[0, 2N-1]*, where *N* is the total instance size. Subsequently, we convert this number into a *N* bit binary, where 1's represent select and 0's do not select. For the Greedy initialization, we reuse the algorithm proposed in Sec. III to generate an initial solution.

5.3 Neighbor Structure

We use the K-Swap neighbor structure, which is also used in [2]. The definition of K-swap structure for a subset sum problem is the following:
"A vector Y =$\{y_1, y_2, \ldots, y_n\}$ *is said to belong the k-swap neighbor of X=*$\{x_1, x_2, \ldots, x_n\}$ *if and only if* $\sum_{j=1}^{n} w_j \cdot y_j \leq T$ *and* $\sum_{j=1}^{n} |x_j - y_j| \leq k$*",*where *X* and *Y* are binary vectors, W = $\{w_1, w_2, \ldots, w_n\}$ is the input instance array and *T* is the target number.
The neighbor solutions can be generated by making at most *K* changes to the current solution. Clearly, the time complexity of generating the neighbor solutions is *O(nK)*. Here we choose *K = 2* to get good balance between the size of neighbor solutions and the algorithm running time. The algorithm we used to generate K-swap neighbor solutions is shown as followings:

```
Input:      Initial solution X={x1,x2,…,xn}, Instance array W = {w1,w2,…,wn} and
            target number T
Output:     Neighbour solutions S
Procedure:  KSwap (X, W, T)
                  S = []
                  n = W.length
                  for (i = 0 … n -2) {
                  flip1Bit = X.clone
                  flip1Bit [i] = ! flip1Bit [i]
                  if (f(flip1Bit) < = T)
                        S.append(flip1Bit)
                  for (int j = i+1; j < n, j ++)
                        flip2Bits = flip1Bit.clone()
                        flip2Bit [i] = ! flip1Bit [j]
                        if (f(flip2Bit) < = T)
                              S.append(flip2Bit)
                  return S
```

In the pseudo-code, f(*) is the evaluate function of a given solution and simply is the sum of the numbers selected. Summarizing the algorithm, we first flip each bit of the given solution to get neighbor solutions after making one change. Then, based on the flip-one-bit solution, we flip another bit to get all the solutions that make two changes to the given solution. Finally, we add all the new solutions to *S* if their sum is less than or equal to the target number.

5.4 Gradient descent and Simulated annealing

Two probabilistic techniques are considered in this report with the aim to approximate the global optimum: Gradient Descent (GD) and Simulated Annealing (SA). GD simply select the best among all the neighbor solutions and terminates when no better solution can be found. The GD algorithm is shown below.

```
Input:      Neighbor solutions S. Current best solution s
Output:     New selected solution s'
Procedure:  SteepestDescent (S, s, W)
                  n = S.length
                  for (int i = 0; i < n; i++)
                  s' = S[i]
                  if (f(s') > f(s))
                        s' = s
                  return s
```

SA represents an improvement from GD in that is does not get stuck at local minima. It does so by accepting worse-than-current solution with probability P and dynamically changing the solutions that are selected with probability 1. This last parameter is called Temperature (T), and it varies as we get closer to the final solution so that the system converges to a stable solution. The SA algorithm is shown below

```
Input:      Neighbour solutions S. Init temperature T0, initial solution
Output:     New selected solution s.
Procedure   SimulatedAnnealing (S, s)
                  while (not terminate)
                  s' = pick random solution from S
                  T = T0
                  if (f(s') > f(s))
```

```
                    break
            else
                    p = e (f (s') – f(s))/T
            if random(0, 1) <= p
                    break
            T = T – 0.1
            return s'
```

It can be seen the probability of accepting bad solution is decided by function e $(f\ (s') - f(s))/T$. For high Temperatures T, the algorithm is encouraged to explore more on neighbor solutions. For low Temperatures T, the algorithm tends to only pick better solutions to speed up convergence. The whole process is repeated until either the time limit exceeds or the final solutions is found.

5.5 Performance Evaluation

In this section, we evaluate the performance of local search algorithms. We focus on evaluation of the quality of the solution with increasing instance size and varying time limitation (10s, 20s, 40s). We also study the different convergence speed of gradient descent and simulate annealing. Finally, we compare the performance between local search with other algorithms (mainly greedy and ILP). Each experiment was repeated at least 20 times to get statistical significant results.

Quality of solution with increasing instance size

We first test the quality of solution found by different local search algorithm with increasing instance size. This allows us to understand the computation boundary of each algorithm. All experiments were done with 20 seconds' limitation. We will test the performance with different time limitations in following section.
We use two metrics to evaluate the quality of solution: accuracy and success rate.

Accuracy: The normalized gap between the sum of solution found and target value. For example, if the sum of solution found is S and the target number is T, then accuracy is computed as (T-S)/S.
Success rate: The probability that the algorithm can found exact solution. For example, if the algorithm X find the exact correct solution n time out of N experiments, then the success rate is n/N.

	GD		SA	
Size	Greedy	Random	Greedy	Random
100	1.00	1.00	1.00	1.00
200	1.00	1.00	1.00	1.00
300	1.00	1.00	1.00	1.00
400	1.00	1.00	1.00	1.00
500	1.00	1.00	1.00	1.00
600	1.00	1.00	1.00	1.00
700	1.00	1.00	1.00	1.00
800	1.00	1.00	1.00	0.98
900	1.00	0.93	1.00	0.91
1000	1.00	0.88	1.00	0.84

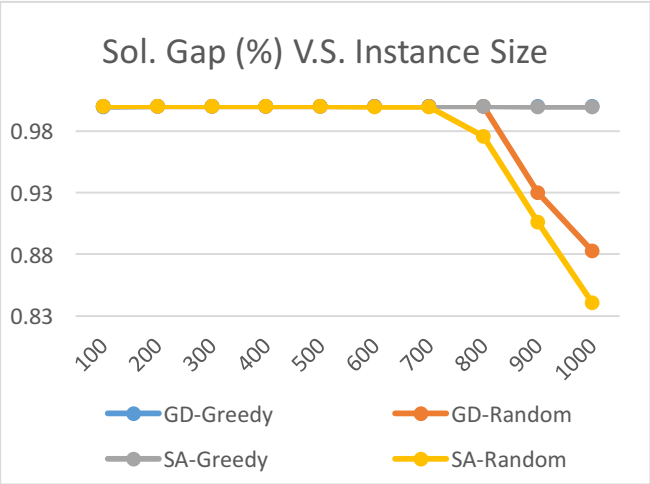

Figure 1. The accuracy of solution decreases after instance size exceeds 600 for simulated annealing and gradient descent with random initialization.

As shown in Fig. 1, all algorithms achieve satisfactory accuracy when instance size is less than 600. However, as instance size keeps increasing, the accuracy of gradient descent and simulated annealing both decrease with random initialization. The main reason is that the randomly generated initial solution can be far away from the target value. Since we can only make at most two changes to the previous selection in each iteration, the improvement of the solution can be slow. The greedy initiation does not have this problem since the initial solution has already been very close to the optimal.

However, the problem of greedy initialization is that it can be easily trapped in local optimal. As shown in Fig.2, the success rate is only around 0.5 to 0.8 if we use greedy initialization and gradient descent local search. Also, we found that the gradient descent usually terminates after only one iteration with greedy initialization. This implies that local search did not actually improve the quality of initial solution at all.

It appears that the gradient descent + random initialization and simulated annealing + greedy initialization can both achieve better performance. The important conclusion is that the choice of the local search algorithm relies on the quality of initial solution. If the initial solution has been good enough, then it is better to use simulated annealing algorithm to avoid converging at local optimal solution. On the other hand, if the initial solution is far away from the optimal, then use gradient descent can allow faster convergence. This also implies that we can combine the advantage of both local search algorithm by first using gradient descent to find a good enough solution, then used simulated annealing trying to further improve it. We leave the evaluation of this method to future research.

	GD		SA	
Size	Greedy	Random	Greedy	Random
100	0.5	1	1	1
200	0.7	1	1	1
300	0.55	1	1	1
400	0.75	1	1	1
500	0.55	1	1	1
600	0.6	1	1	0.75
700	0.85	1	0.95	0.35
800	0.6	1	0.95	0.2
900	0.6	0.6	0.75	0.1
1000	0.5	0.3	0.9	0

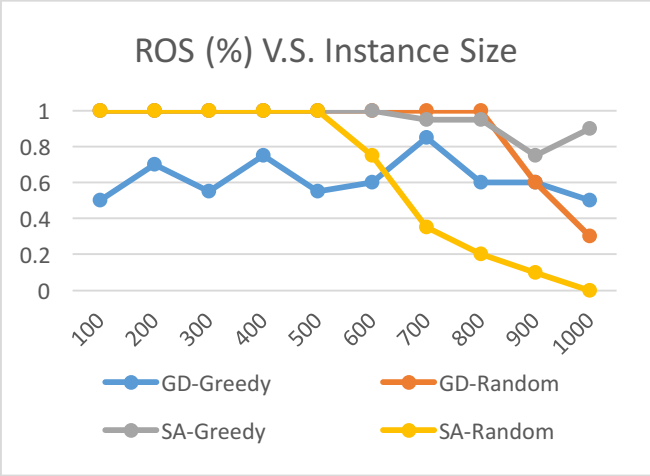

Figure 2. Variation of success rate with increasing instance size. The success rate of Gradient descent + greedy initialization is low even for small size instance array

Analysis on the Execution time

Fig. 3 shows the time took by each algorithm to converge. Note we set the time limitation to 20 seconds. The gradient descent + greedy initialization almost always find the right solution instantly. As we explained before, this is because it only needs very few iterations to converge. Not surprisingly, the random initialization needs much longer time to converge. We also have explained that it is because the local search can only improve solution gradually during each iteration. Also, we found simulated annealing generally takes longer than gradient descent (random initialization). This is understandable as the gradient descent always choose the best from neighbor solutions, thus achieving better convergence speed than simulated annealing (which may accept worse solution).

	GD		SA	
Size	Greedy	Random	Greedy	Random
100	0.00	0.00	0.03	0.08
200	0.00	0.05	0.12	0.51
300	0.01	0.22	0.24	2.18
400	0.01	0.72	0.57	3.76
500	0.03	1.81	1.66	7.20
600	0.03	2.97	2.63	14.58
700	0.03	6.48	2.32	19.00
800	0.12	10.06	3.69	19.31
900	0.15	15.95	5.96	18.78
1000	0.27	16.83	2.33	20.29

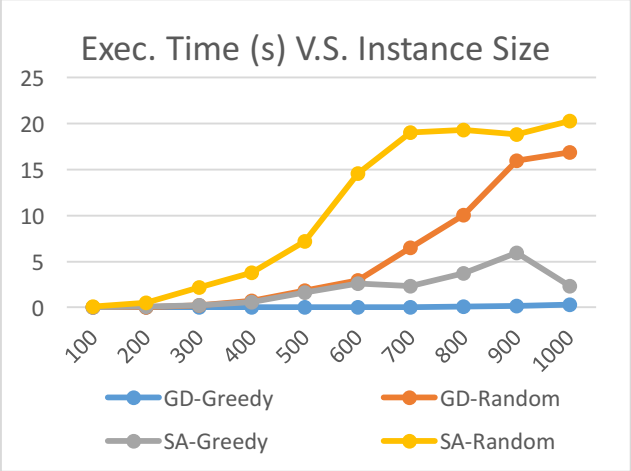

Figure 3. Variation of running time with increasing instance size. Random initialization lead to much longer converge time.

Fig.4 shows the performance of each algorithm under different time limitations (10, 20, 40 seconds). We use random initialization in all experiments as the local search terminates instantly with greedy initialization. As we expected, the solution quality improves as the time limitation increases. Specifically, the accuracy for gradient descent increases from 0.8 to 0.95 when time limitation increases from 10s to 40s for 1000 number instance array. The improvement in solution quality of gradient descent is greater than that of simulated annealing, which only increase the accuracy from 0.8 to 0.9. This is because gradient descent converges faster than the simulated annealing, which can be an advantage when the instance size is large.

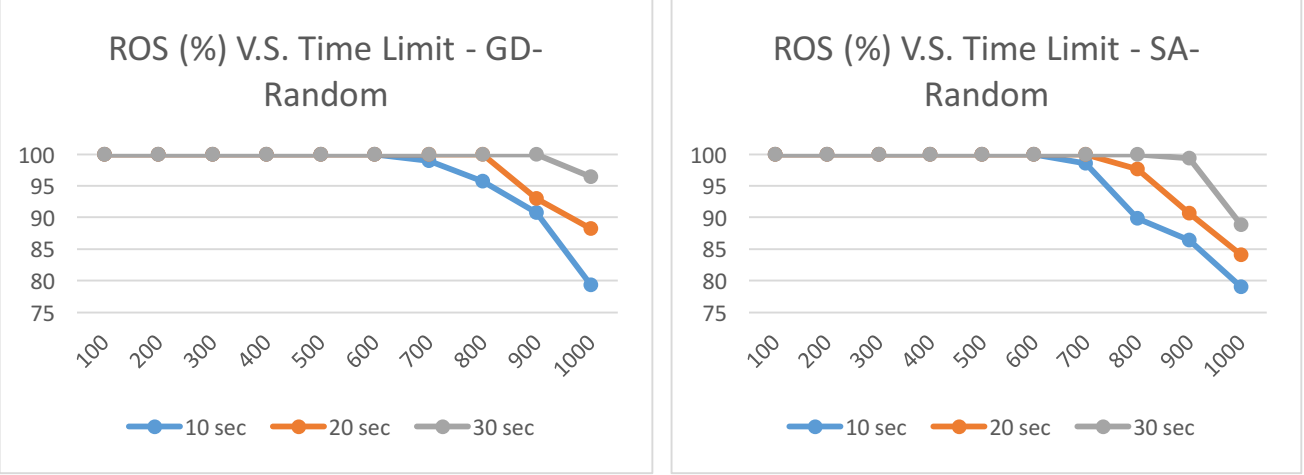

Figure 4. Performance of each algorithm under different time limitations (10, 20, 40 seconds). Larger time limitation leads to better performance

Analysis on the Convergency

Fig.5 and Fig.6 shows the convergence line of gradient descent and simulated annealing with different instance size (200, 400 and 800) and randomly generated initial solution. Clearly gradient descent converges much faster than the simulated annealing. Specifically, simulated annealing takes over 10X iterations to converge when instance size is 800. However, an interesting discovery is that simulated annealing can get close to exact solution very fast – as shown in Fig. 6, the difference between sum of solution and the target number is close to 0 after 70 iterations. However, simulated annealing will not terminate unless the exact solution is found or

the lowest temperature is reached. This property contributes to the long tail of the convergence line as shown in Fig. 6. The advantage of this is that it can increase the probability of finding the optimal solution, with the cost of much longer searching time. In real application, we can control the time limitation to get the good trade-off between the solution quality and the running time as a need-basis.

Iter.	200	400	800
1	-1	-8470	-16407
3	-3	-6889	-13257
5	-5	-5330	-10147
7	-7	-3884	-7062
9	-9	-2493	-4030
11	-11	-1137	-1051
13	-13	0	0
15	-15	0	0
17	-17	0	0
19	-19	0	0

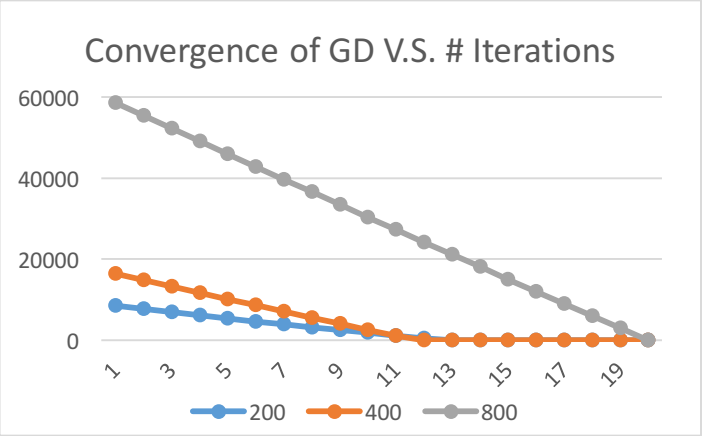

Figure 5. Convergence line for gradient descent + random initialization with different instance size (200, 400, 800)

	200	400	800
Iteration number for Convergence	80	201	245

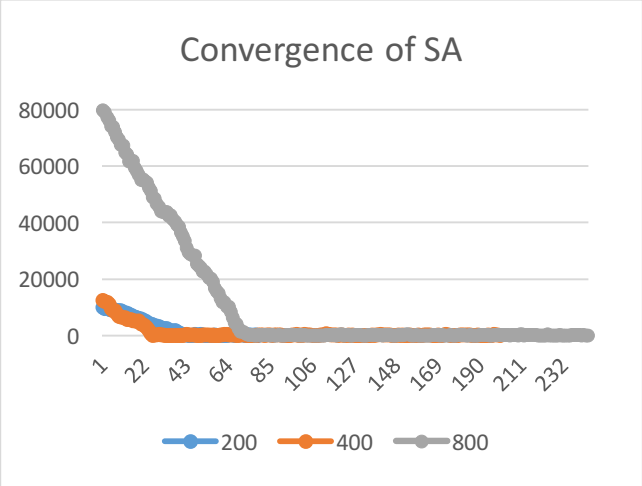

Figure 6. Convergence line for SA with Random initialization for different instance size

Comparison between local search and other algorithms

Finally, we compare the performance of local search with previous studied algorithms. We choose gradient descent + random initialization and simulated annealing + greedy initialization as they have been shown to achieve best performance in evaluation before. We choose instance size from 50 to 500, which we believe is large enough to understand the performance trend of the algorithms. However, we do not include exhaustive algorithm here because as shown in Fig.7, its running time grows exponentially with increasing instance size and soon become unbearable after instance array size exceeds 50.

Iter.	Exhaustive
5	0.0000
10	0.0000
15	0.0002
20	0.0002
25	0.0018
30	0.0106
35	0.0865
40	0.5277
45	4.5297
50	27.4820

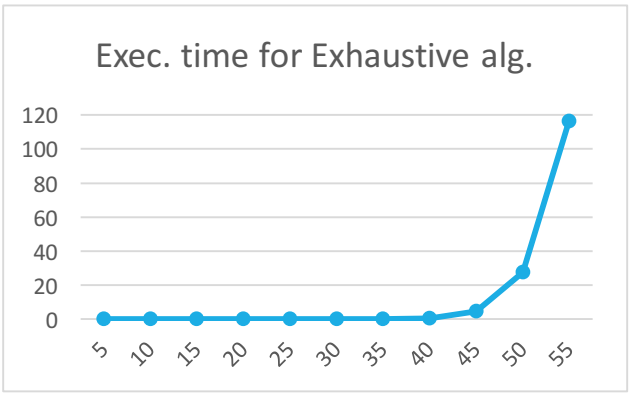

Figure 7. Execution time of the exhaustive algorithm, showing an exponential increase with the instance size.

Fig. 7 shows the success rate of each algorithms. All algorithms can find the optimal solution except greedy algorithm. It shows that both ILP and local search are powerful tools to solve subset sum problem with large instance size that is unsolvable by exhaustive algorithm. We should also mention that while it seems like greedy algorithm can only find exact solution with probability 20% - 40%, the solution it finds is very close to optimal (not shown here). This implies that we can also choose greedy algorithm if some small errors are bearable in the application.

Size	Greedy	ILP
50	0.3	1
100	0.1	1
150	0.05	1
200	0.1	1
250	0.25	1
300	0.35	1
350	0.2	1
400	0.1	1
450	0.35	1
500	0.4	1

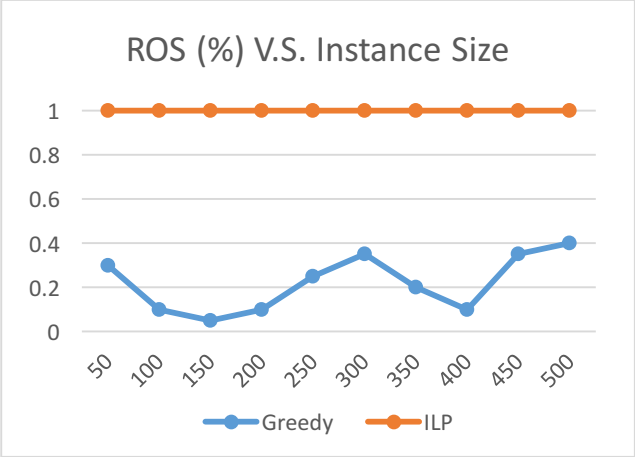

Figure 8. Success rate of different algorithms with increasing instance size. All algorithms achieve 100% success rate except greedy algorithm

Fig. 8 compares the time used by each algorithm. Notice while all experiments are done under 20 seconds constrains, no algorithms exceed this limitation. Specifically, the greedy generally requires <10-6 seconds to converge. ILP finds optimal solution in 0.01s and this is not influenced much by the increasing instance size. The local search algorithm can achieve similar or even faster convergence time for instance array with <150 elements. However, the time cost of local search increases linearly with the increasing size, which makes them much more expensive then ILP for large size instance (e.g., >200).

Size	Greedy	ILP	SA	GD
50	0.000	0.009	0.000	0.004
100	0.000	0.009	0.004	0.037
150	0.000	0.009	0.016	0.085
200	0.000	0.011	0.048	0.128
250	0.000	0.012	0.109	0.189
300	0.000	0.020	0.225	0.148
350	0.000	0.021	0.394	0.734
400	0.000	0.023	0.555	1.215
450	0.000	0.023	0.927	1.318

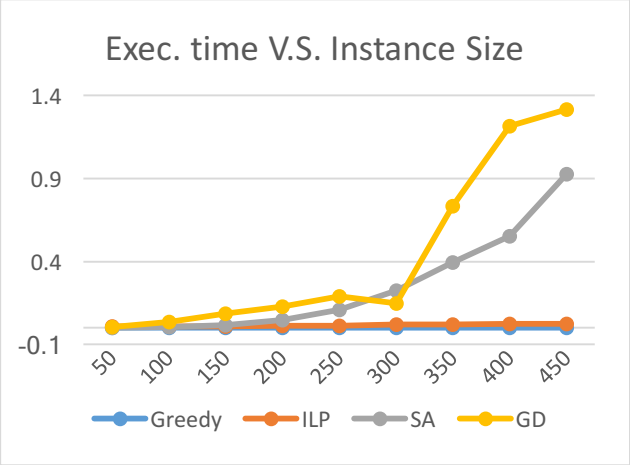

Figure 9. Execution time for different algorithms as a function of the instance size.

5.7 References

[1] Stützle, Thomas, Mauro Birattari, and Holger H. Hoos, eds. Engineering Stochastic Local Search Algorithms. Designing, Implementing and Analyzing Effective Heuristics: International Workshop, SLS 2009, Brussels, Belgium, September 3-5, 2009, Proceedings. Vol. 5752. Springer, 2009.

[2] Ghosh, Diptesh, and Nilotpal Chakravarti. "A competitive local search heuristic for the subset sum problem." Computers & operations research 26.3 (1999): 271-279.

Conclusion

In this report, we comprehensive evaluate five different techniques of solving the sub-set sum problem (exhaustive, greedy, ILP and local search). Our recommendation is that ILP appears to be the most efficient approach in terms of solution quality and running time. However, if AMPL or Cplex are not available, local search (random initialization + gradient descent or greedy initialization + simulated annealing) can also be a good option for reasonable size instance array length (<1000). Greedy algorithm has the fastest running speed, but can only be useful if the local optimal solution is acceptable. We do not recommend using exhaustive solver unless the instance array has very limited size (< 50).

We summarize our findings in terms of performance, execution time and solution quality for the considered solvers.

Exhaustive algorithm: The exhaustive algorithm will iteratively search all combinations of the numbers in given instance array. Therefore, its running time increases exponentially with instance size. This decides that it can only solve problems with very limited size. According to our experiments, the exhaustive algorithm can get the optimal solution for instance array with < 55 elements in reasonable time (few minutes).

Greedy algorithm: Greedy algorithm has linear time complexity since it only iterates over the instance array one time. This decides that greedy algorithm can return solution instantly for very large instance array. However, the success rate of greedy algorithm is not satisfactory – our experiments show that greedy algorithm can only get the optimal solution with less than 40% probability. This means we cannot use greedy algorithm if exact solution of subset sum problem is needed. However, the gap between the greedy solution and the optimal one is very small – especially for large size instance array. The reason is that as the number increase, there is higher probability that greedy algorithm can find solution close to the optimal. In other words, we can use greedy algorithm in applications where small errors are allowed, and the time constrains is very stringent.

ILP: Overall speaking, ILP has the best performance in balancing solution quality and running time. It achieves sub-linearly running time for reasonable size instance array and only needs 10 ms to get the optimal solution for instance array with <500 elements. Furthermore, our evaluation in Sec.IV shows ILP's success rate only starts to drop for instance array with >6500 elements (under 50 ms time limitation). This implies that Cplex has been optimized very well for solving large scale ILP, which makes it an ideal solution for the subset sum problem.

Local search: Different local search algorithms and the quality of initial solution can have huge impact on the running time and the results. Our evaluation shows greedy initialization + gradient descent achieves the fastest converge time, with the cost of low success rate. On the country, greedy initialization + simulated annealing can achieve satisfactory success rate. This is because simulated annealing will probabilistically accept worse solution, which helps it get out of the local optimal. We find random initialization usually requires longer converge time than greedy initialization, especially for simulated annealing. This is because the randomly generated initial solution can be far away from the optimal solution. Limited by the neighbor structure, the converge speed can be slow as the solution can only be improved a little bit during each iteration.