Solving the Subset Sum Problem using Learning Automata

Thomas Jager, Kristian Tveiten, Torbjørn Skagestad

November 2007

Keywords: Learning Automata, LA, Subset Sum Problem, Knapsack Problem, Machine Learning, NP-Complete

Abstract

In this project we want to find out if Learning Automata is suitable for solving the Subset Sum Problem.

Contents

1 Introduction **4**

2 Background **4**
2.1 Contributions to this paper 4
2.2 Project Description 5
2.3 Project Goals 5

3 Subset Sum Problem **6**
3.1 NP-Complete 6
3.2 Approximate Solution 6
3.3 Exact Solution 6
3.4 Super increasing sets 7
3.4.1 Example 7

4 Experiments **8**
4.1 Game of Automata 8
4.1.1 Parameters 8
4.1.2 Results 8
4.2 Game of Automata with Skewed Feedback 8
4.2.1 Results 10
4.2.2 Limitations 10
4.3 C Implementation 10
4.3.1 $P_{changes}$ 11
4.3.2 Results 11
4.3.3 Limitations 11
4.4 Game of Automata with Skewed Feedback 2 12
4.4.1 Results 12
4.4.2 Limitations 12
4.5 Game of Automata with Skewed Feedback 3 12
4.5.1 Results 12
4.6 Brute force implementation 12
4.6.1 Results 13
4.7 Better exponential time algorithm 13
4.8 Branch and bound solver 13
4.8.1 Results 13
4.9 Dynamic Programming solver 14

5 Discussion **15**
5.1 Problems 15

6 Conclusion **16**

List of Figures

1 GoA, $p = 0.4$,$S_{max} = 4$ 9
2 GoASF, $p = 0.1$ 10
3 Testing of $P_{changes}$ and S_{max} Y-axis is the number of iterations. 11

1 Introduction

We have chosen the project Subset Sum Problem. In this project we will be given different sets of integers, and we will need to find out which of these non-empty subsets that equals exactly zero. This should be solved by using different kinds of Learning Automata. We will compare these results and find the most efficient algorithm. A similar kind of problem is also present in the course Discrete Mathematics, and this was one of the reasons that we chose this exercise.

2 Background

2.1 Contributions to this paper

The Subset Sum problems complexity is very well suited for use in public key encryption algorithms. Solving such a problem would in most cases equal to breaking an encryption key. One well known and often used method for breaking an encryption key is the Brute Force Attack [7].

While trying to find the solution for solving our problem we have studied the principles of Learning Automata [4] which is a subcategory under Machine Learning [5]. Using these kind of techniques for solving programming challenges have been more and more popular the last decade. The reason for this is that Learning Automata, LA for short, is a method that uses rewards and punishments to either encourage or discourage an action. After a period of time this should result in encouraged actions being repeated and discouraged actions being discarded. This method is described as very efficient in several papers and web-sites concerning the topic[5]. We were unable to find any other publications that have solved the Subset Sum Problem using LA. Therefore we had to look at other publications that had a close to similar problem. The Knapsack Problem[6] is a problem where LA have been used to find a solution. This is very similar to the Subset Sum Problem in many ways. The most essential similarity is that it contains a set of numbers, and the task is to pick out the numbers that gives the solution wanted. In the Subset Sum Problem this solution is often 0 or 1. But it could also be the problem of finding a set of n distinct positive real numbers with as large a collection as possible of subsets with the same sum. There have been produced reports that solves different problems around the Knapsack Problem using LA. In February 2007 the report "Learning Automata-Based Solutions to the Nonlinear Fractional Knapsack Problem With Applications to Optimal Resource Allocation" [6] was published. This paper considers a non linear Knapsack problem and demonstrates how its solution can be used to solve resource allocation problems on the World Wide Web. The interesting part is that LA have been used to solve the Knapsack Problem in a more effective manner. Since this problem is directly related to the Subset Sum Problem this leads us to believe that a LA based solution is more effective than Brute Force, or the existing algorithms for that matter. In this paper we will

attempt to prove this statement.

2.2 Project Description

In this project we want to find out if LA is suitable for solving the Subset Sum Problem, and compare performance with other ways of solving the problem will be an important part of the project. We want to concentrate on solving problems exactly, because finding an approximate solution could be as easy as just specifying the time the solver should run for and return the best solution found.

2.3 Project Goals

We want to implement a LA solver for both approximate solution and exact solution. But we will concentrate on implementing an exact solver. We will also implement some known algorithms for solving the problem to compare speed. We also want to compare different reward/punishment strategies as well as decision strategies.

3 Subset Sum Problem

Given a set of integers, does the sum of some non-empty subset equal to exactly zero. For example, given the set $\{-7, -3, -2, 5, 8\}$, the answer is yes because the subset $\{-3, -2, 5\}$sums to zero. The exact solution to the subset sum problem is NP-Complete [6], and is perhaps the simplest of the NP-Complete problems to describe.

$S$is a solution.
$A = \{x_1, x_2, ..., x_n\}$
$S \neq \emptyset$
$\sum S = 0$

3.1 NP-Complete

NP-Complete problems are a class of complexity problems that run in polynomial time. Polynomial time refers to the computation time of a problem where the run time, $m(n)$, is no greater than a polynomial function of the problem size, n. Written mathematically using big O notation, this states that $m(n) = O(nk)$ where k is some constant that may depend on the problem. For example, the quick-sort sorting algorithm on n integers performs at most An^2 (Worst case) operations for some constant A. Thus it runs in time $O(n^2)$and is a polynomial time algorithm.

3.2 Approximate Solution

Some problems don't need a exact solution. Approximate solutions are often useful in real life situations like optimizing packing of a container. In these situations it is desirable to get a sum as close to the exact solution as possible. There are many things to consider when searching for a approximate solution. If no exact solution exist in the subset, we will need to set a threshold for how many runs should be done. A simple way to implement this is to keep the sum closest to the exact solution, and discard all other. When the number of runs set as a threshold then is completed, you chose the subset that sums to the solution closest to the exact solution. This type of solution is applicable to the knapsack problem for instance.

3.3 Exact Solution

In cryptography the subset sum problem comes up when breaking encryption keys. When the attacker knows the message and the cipher text (Known plaintext attack), finding the key boils down to a subset sum problem. A key that is not equal to the real key is useless and therefore an exact solution is needed.

3.4 Super increasing sets

It's worth mentioning super increasing sets as they have a interesting property that makes them easy to solve with a greedy algorithm. A super increasing set is a set where each number in the set is the sum of all the previous numbers in the set. If you know your set is super increasing it can be solved in $O(n)$ by sorting numbers from highest to lowest. Iterate trough the set in decrementing order and subtracting numbers that fit within your constraint.

3.4.1 Example

$S = 1, 2, 3, 6, 12, 24$

Constraint = 15, Iterate: 24 does not fit constraint. 12 fits subtract it from your constraint. Constraint = 3. 6 Does not fit. 3 fits and solves the subset.

Solution = $3, 12$

4 Experiments

4.1 Game of Automata

Our first solver[1] used Game of Automata, from now on GoA, is a subset sum solver made in Perl[2]. Each number in the set was connected to a Tsetlin Automaton[4]. The Automaton are initialized with a random state, either 1 or 0 where 0 is no and 1 is yes. If the Automaton at index n answered yes the number at index n is added to a new sum. If this new sum is closer to the target[2] compared to the sum of the previous run, each Automaton has a p chance of getting a reward. If the new sum is further away from the target sum, then each Automaton has p chance of getting punishment.

```
if sum{S} == 0:
  done
if abs(T-oldsum) < abs(T-sum{S}):
 for S_i in S:
  if random() < p:
   S_i.punish()
else:
 for S_i in S:
  if random() < p:
   S_i.reward()
oldsum = sum{S}
```

4.1.1 Parameters

$S_{max}$is the maximum number of states on the yes and no side of the Tsetlin Automaton.

4.1.2 Results

The solver can find solutions. In figure 1 you can see how many iterations it takes the reward/punishment kernel before finding a solution over 400 runs. 100% of the runs solve the subset sum after 27000 iterations. This GoA solver shows that LA can be used to solve subset sum problems.

4.2 Game of Automata with Skewed Feedback

Our second solver[3] uses Game of Automata with Skewed Feedback, from now on GoASF. The GoA Perl implementation was changed to give out reward/punishment based on the relative value of the number in the set and the new sum. The GoASF algorithm works like this: If the new sum is larger then the target, negative numbers included in the sum (Automaton answered yes) gets p chance

[1] set_solver_simple.pl
[2] The target sum is 0.
[3] set_solver_spes.pl

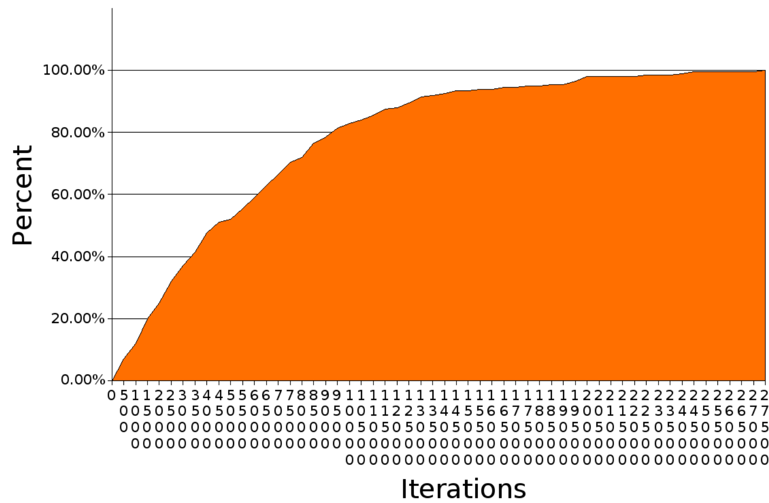

Figure 1: GoA, $p = 0.4$,$S_{max} = 4$

for reward. If the negative number was not included in the sum (Automaton answered no) it gets a p chance of punishment. If the number is positive and was included in the sum it has p chance of punishment, and if it was not included it gets a p chance of reward.

If the new sum is smaller then the target, negative numbers included in the sum get punished. Negative numbers not included in the sum gets rewarded. Positive numbers included in the sum gets rewarded. Positive numbers not included in the sum gets reward.

```
if sum{S} == 0:
  done
if sum{S} > 0:
 for S_i in S:
  if S_i < 0 and random() > {p_changes over n}:
   if S_i.decide():
    S_i.reward()
   else:
    S_i.punish()
else:
 for S_i in S:
  if S_i > 0 and random() > {p_changes over n}:
   if S_i.decide():
    S_i.reward()
   else:
    S_i.punish()
```

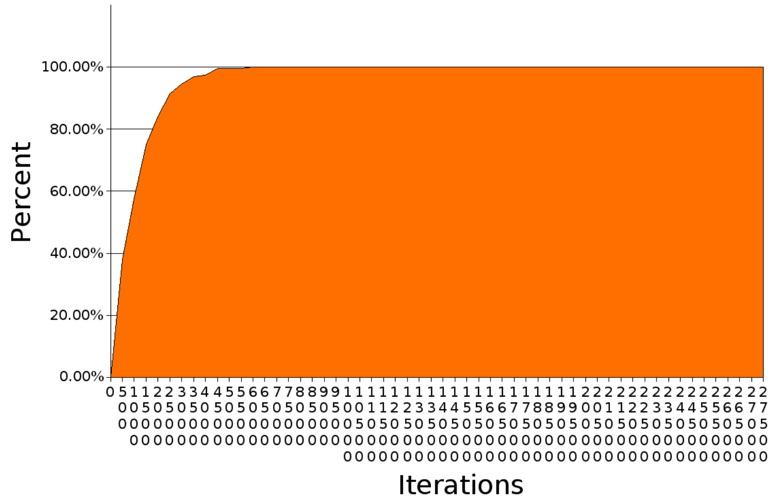

Figure 2: GoASF, $p = 0.1$

4.2.1 Results

Figure 2 shows how many iterations it takes of the reward/punishment kernel before finding a solution over 400 runs. As seen in the figure this GoASF implementation finds solutions faster then the GoA as shown in figure 1. All the runs finds a solution to the same set as in the GoA in less than 7000 iterations. The GoASF implementation is 3.8 times faster then the GoA using the maximum number of iterations as the scale.

4.2.2 Limitations

This punishment reward strategy only works for sets with both positive and negative numbers. Also since this rewards negative or positive numbers exclusively it's not suited to find solutions that are not summed to zero. This strategy can not be used for the knapsack problem. Some experiments show that this strategy is best suited for large uniform sets. And less suited for small sets with few or one solutions.

4.3 C Implementation

A C implementation[4] was made of GoASF. The real run time of the C implementation is about 35 times faster then the Perl implementation for the same number of iterations. This C implementation was used to test different and larger sets. Also different parameters for p and S_{max} was tested.

[4] set_sum_solver.c

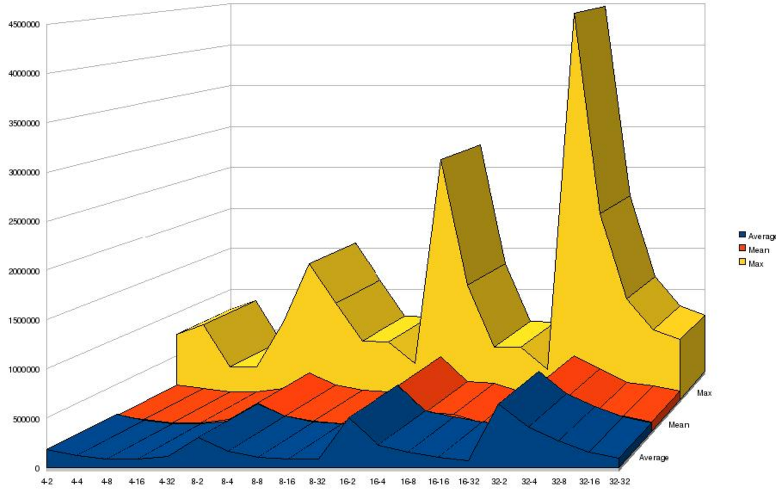

Figure 3: Testing of $P_{changes}$ and S_{max} Y-axis is the number of iterations.

4.3.1 $P_{changes}$

A new parameter was added to this implementation. $P_{changes}$ is used to calculate the probability p of punishment and reward. $P_{changes}$ boils down to the number of probable punishments and rewards for each iteration. p is calculated from $P_{changes}$ like this:

Assuming n is the size of the set.

$p = 1/n/P_{changes}$

4.3.2 Results

Figure 3 shows how the $P_{changes}$ and Max State parameters influence the number of iterations the implementation needs to find a solution. The first number is Max State and the second number is $P_{changes}$. For example 16-32 means Max State = 16 and P Change = 32. The set in this figure is randomly generated, has 1000 numbers and the range of the numbers are between -30000 and 30000. The optimal parameters for this set is $P_{changes} = 32$ and $S_{max} = 16$

4.3.3 Limitations

Since this is just a C-implementation of the same algorithm as 4.2, the same limitations apply in this implementation.

4.4 Game of Automata with Skewed Feedback 2

GoASF 2 [5] is similar to the first. We sort the set and use a pivot where all numbers below have the lowest values, and rewarded if the current sum is over target. We experimented with different pivots. Splitting the set in two equal parts. Every negative number as below pivot and positive numbers above pivot. A side effect of using a pivot is that we cut down the number of comparisons in the reward/punishment kernel. The reason is that instead of checking if the current number is negative we split it into two sets. One for all the numbers below the pivot, and one for all the number above the pivot, making it slightly faster.

The sorting algorithm we use is an in-place quick sort. The complexity of quick sort is $O(n \cdot log \cdot n)$.

4.4.1 Results

The results are about the same as the first specialized implementation. But it is slightly faster in real run time, but the number of iterations it takes to solve a set is about the same. If we have a set with only positive numbers and we set the pivot to half the set size. It can find solutions assuming the target is positive. But if the solution is a set that is either above or below the pivot it may get stuck because it tries to reward lower numbers (below pivot) if the sum is above the target when in fact the solution may be somewhere above the pivot.

4.4.2 Limitations

This Implementation has the same limitations as the first specialized implementation because it may get stuck trying to find a solution above 0.

4.5 Game of Automata with Skewed Feedback 3

GoASF 3 [6] is very similar to GoASF 2, but tries to fix some problems by changing the reward/punishment strategy. This is done by punishing all numbers if it has the same sum over several iterations.

4.5.1 Results

Unfortunately this implementation has the same problems as the GoASF 2 and we quickly gave up trying to fix it.

4.6 Brute force implementation

A brute force implementation[7]was made to compare with the Learning Automata. This naive algorithm tests all possible subsets. And runs in $O(2^n)$

[5] set_sum_solver2.c
[6] set_sum_solver3.c
[7] set_solver_brute.c

exponential time.

4.6.1 Results

Our brute force implementation is only useful for smaller sets. Our implementation can only handle set sizes of less then 32 because of a integer overflow. This algorithm is very slow and has a run time of about 2.5 seconds to find the first solution in test set 8[8]. In comparison the GoASF has average run time of 0.5 seconds.

4.7 Better exponential time algorithm

In was our intention to test as many algorithms as the time span of the project allowed. The Horowitz and Sahni algorithm was one of the alternatives we considered.[9] This algorithm however would not be very different than the naive Brute Force algorithm, when handling bigger sets. Because of this, we decided to not implement it.

4.8 Branch and bound solver

We made a tree searching solver[10] where we sorted the set and used a branch and bound approach with recursion. The problem with such algorithms is that they are unsuited to solve subset to 0. The bound logic sees if the next branch sums with the current sum without going over the target. If the next value goes over the target the algorithm backtracks and tries the values on the neighboring branch. This means that the bounding might be inefficient when negative numbers are introduced. Sorting the set before applying this algorithm seems to fix some of the inefficiencies, but might not be the best solution. It might be possible to interpolate the set to only contain positive numbers. We have however not made any attempts to make such a solution.

4.8.1 Results

The branch and bound algorithm is able to handle much bigger sets than the brute force algorithm. It is also much faster with all sets we tried it with. We tested the algorithm with the same set as with the LA in Figure 4.2. This algorithm could find the first solution within one second. The first solution this algorithm finds in the Test set[11] is $\{-171 - 171 + 12 + 330\}$. This suggests that this algorithm finds smaller subsets first, which fits well with the Branch and Bound theory. After more testing with other sets it looks like this algorithm is faster than the GoASF. The algorithm uses slightly more memory than the GoA though.

[8] test_set8.txt
[9] http://wwwhome.cs.utwente.nl/~woegingergj/papers/exact.pdf
[10] set_solver_bandb.c
[11] test_set.txt

4.9 Dynamic Programming solver

When we first began on this project we wanted to implement the Algorithm describe in the wikipedia article[12] about the subset sum problem. It looks like the algorithm description on Wikipedia is incomplete. Unfortunately this was harder to do then we thought and we never got it to work correctly. We also tried implementing other similar algorithms, but they where all used to solve the knapsack problem, and not the subset sum problem. This means none of the algorithms we tried could handle negative numbers without failing to find a solution. Again we suspect we could make a interpolation algorithm but time was limited and we decided to skip it.

[12] http://en.wikipedia.org/wiki/Subset_sum#Pseudo-polynomial_time_dynamic_programming_solution

5 Discussion

Unfortunately we did not get the Dynamic Programming solver to work. The theory is that this approach could solve it in polynomial time $O(np)$ where n is the number of items in the set and p is the size of the numbers. The only problem is the memory used for the matrix. The memory usage could be optimized by using a bit field instead of integers. It would be really interesting to compare the Dynamic programming approach to the Branch and Bound.

The Branch and Bound approach was surprisingly fast with our test set, but in the worst case scenario it might be just as slow as the Brute Force algorithm. When we tried to solve sets with very few or just one solution the Branch and Bound algorithm was almost always faster then LA. Since the LA is non-deterministic the few runs where the LA was faster could be explained as random aberrancy. We think that the deterministic nature of the Branch and Bound approach is preferable to the LA in this case.

5.1 Problems

When working on this project we encountered a few problems. The biggest problem as implementing alternative algorithms for solving the subset sum problem. The LA just took a few minutes to implement and a few hours to optimize. Also it seems like most sites on this subject concentrate on the knapsack problem witch is similar but different because in the subset sum problem numbers might be negative. In the knapsack problem negative weights wouldn't make any sense, and algorithms in turn does not handle negative numbers. Some kind of interpolating algorithm to convert a set with both positive and negative numbers to a set with only positive numbers. But how would one do this? For example just adding the absolute value of the lowest value to all the numbers in the set and the target wouldn't work because the new target would depend on the number of numbers in the final set. Circular dependency.

$\{-7, -3, 10\}$add 8 to every number in the set $\{1, 5, 18\}$the new target would be 8, but it's wrong since the correct target is 24 or $3 \cdot 8 = 24$or the number of numbers in the solution times what you added.

6 Conclusion

Our goal for the project was to solve the Subset Sum Problem using LA. By implementing our own algorithm, we succeeded in doing this. The algorithm was first quite simple, and straight forward. In this stage it was not very efficient, but already able to solve the problem. Several improvements were made as the project progressed. These improvements greatly decreased the solvers run time. This process of improving the code was very interesting. We learned a lot about the problems complexity, and how to solve it in an effective manner. Even though the solver have been improved to solve the problem much faster than original, it still is not the most efficient solution. Compared to the slower Brute Force however, our algorithm is more efficient. The solver is able to solve both small and large sets, which is a big advantage. If a solutions exist in a set, the solver will eventually find it. The time needed to find a solution depends on the size of the set, and how many possible solutions there is. For further work it would be interesting to implement the Dynamic Programming Solver, as described in wikipedia.

References

[1] Subset Sum Problem: http://en.wikipedia.org/wiki/Subset_sum_problem

[2] Perl programming language: http://www.perl.com/

[3] Knapsack problem: http://en.wikipedia.org/wiki/Knapsack_problem

[4] Learning Automata: http://www.cs.cmu.edu/afs/cs/project/jair/pub/volume4/kaelbling96a-html/node10.html

[5] Machine Learning: http://en.wikipedia.org/wiki/Machine_learning

[6] Ole-Christoffer Granmo, B. John Oommen, Svein Arild Myrer and Morten Goodwin Olsen, "Learning Automata-Based Solutions to the Nonlinear Fractional Knapsack Problem With Applications to Optimal Resource Allocation", IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS—PART B: CYBERNETICS, VOL. 37, NO. 1, FEBRUARY 2007

[7] Brute Force Attack: http://en.wikipedia.org/wiki/Brute_force_attack

Appendices

1. Code and test data - subset.sum.tar.gz