

# Solving the Subset Sum Problem using Learning Automata

Thomas Jager, Kristian Tveiten, Torbjørn Skagestad

November 8, 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

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Background</b>	<b>3</b>
2.1	Research . . . . .	3
2.2	Project Description . . . . .	4
2.3	Project Goals . . . . .	4
<b>3</b>	<b>Subset sum problem</b>	<b>4</b>
3.1	NP-Complete . . . . .	4
3.2	Approximate Solution . . . . .	4
3.3	Exact Solution . . . . .	5
<b>4</b>	<b>Experiments</b>	<b>5</b>
4.1	Initial proof of concept . . . . .	5
4.1.1	Parameters . . . . .	5
4.1.2	Results . . . . .	5
4.2	Specialized implementation . . . . .	5
4.2.1	Results . . . . .	6
4.3	C Implementation . . . . .	6
4.3.1	<i>Pchanges</i> . . . . .	8
4.3.2	Results . . . . .	8
4.4	Brute force implementation . . . . .	8
4.4.1	Results . . . . .	8

# 1 Introduction

We have chosen the project Subset Sum Problem. In this problem 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 Research

We have based our work on 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 way of making a computers "learn". This is done by giving rewards or punishments to dissociate what is a right and what is a wrong action. 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 is a problem where LA have been used to find a solution. This is very similar to the Subset Sum Problem in many ways. And 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. We have not concentrated on the latter since this would make our project more complicated than necessary. There have been produced reports that solves different problems around the Knapsack Problem using Learning Automata. In February 2007 Morten Goodwin Olsen, Ole-Christoffer Granmo, John Oommen and Svein Arild Myrer published the report "Learning Automata-Based Solutions to the Nonlinear Fractional Knapsack Problem With Applications to Optimal Resource Allocation" [6]. This paper considers a non linear Knapsack problem and demonstrates how it's solution can be used to solve resource allocation problems on the World Wide Web. And the interesting part is that Learning Automata have been used to solve the Knapsack Problem in a more effective manner. And 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 the existing algorithms for that matter. In this paper we are going to try to prove this statement.

## 2.2 Project Description

In this project we want to find out if Learning Automata 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.

## 2.3 Project Goals

We want to implement a Learning Automata solver for both approximate solution and exact solution. 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.

$$A = \{x_1, x_2, \dots, x_n\}$$

$$S \neq \emptyset$$

$$\sum S = 0$$

### 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 quicksort 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 an 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 an approximate solution. If no exact solution exists 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 plain-text 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.

## 4 Experiments

### 4.1 Initial proof of concept

As a proof of concept a subset sum solver was 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 zero 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<sup>1</sup> compared to the sum of the previous run, each Automaton has a  $p$  chance of getting a reward. If the new sum is farther away from the target sum, then each Automaton has  $p$  chance of getting punishment.

#### 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 proof of concept show that a Learning Automata can be used to solve subset sum problem.

### 4.2 Specialized implementation

The proof of concept 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 specialized 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 for

---

<sup>1</sup>The target sum is 0.

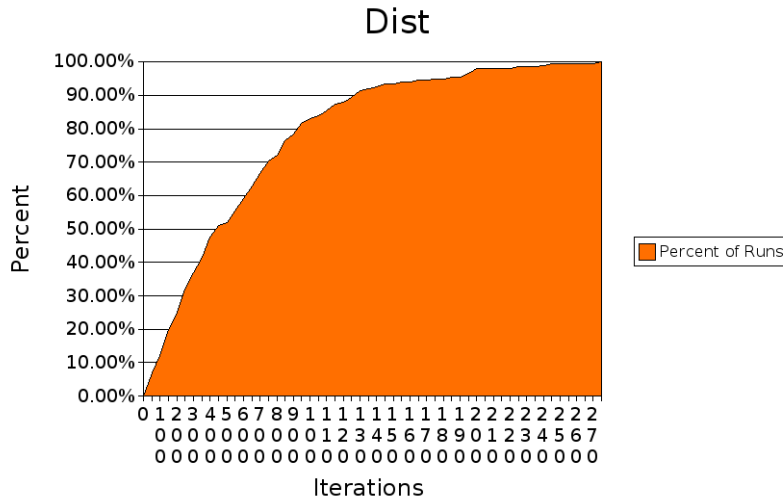


Figure 1: Proof of concept,  $p = 0.4, S_{max} = 4$

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 than 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.

#### 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 specialized implementation finds solutions faster than the proof of concept as shown in figure 1. All the runs find a solution to the same set as in the proof of concept in less than 7000 iterations. This specialized is 3.8 times faster than the proof of concept.

#### 4.3 C Implementation

A C implementation was made of specialized Perl implementation. The real run time of the C implementation is about 35 times faster than 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.

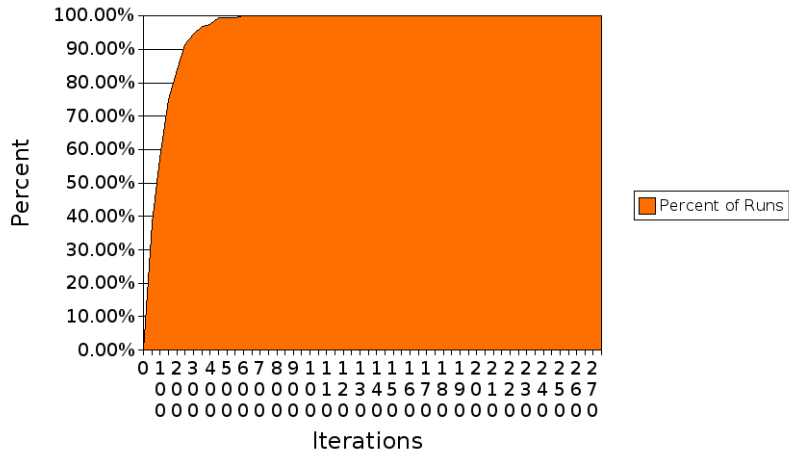


Figure 2: Specialized implementation,  $p = 0.1$

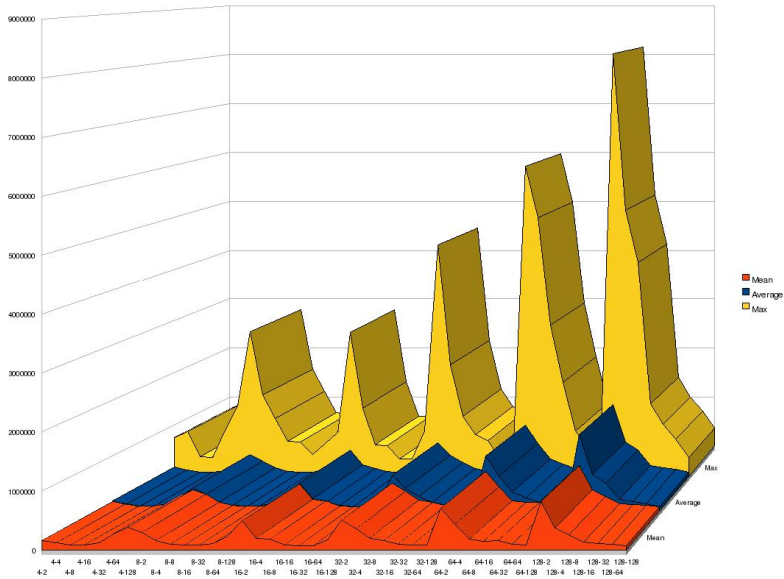


Figure 3: Testing of  $P_{changes}$  and  $S_{max}$  Y-axis is the number of iterations.

#### 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.

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

#### 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  $S_{max} = 16$

### 4.4 Brute force implementation

A brute force implementation was made to compare with the Learning Automata.

#### 4.4.1 Results

The brute force implementation is useful only for very small sets. More testing is needed for small sets.

## References

- [1] [http://en.wikipedia.org/wiki/Subset\\_sum\\_problem](http://en.wikipedia.org/wiki/Subset_sum_problem)
- [2] <http://www.perl.com/>
- [3] [http://en.wikipedia.org/wiki/Knapsack\\_problem](http://en.wikipedia.org/wiki/Knapsack_problem)
- [4] <http://www.cs.cmu.edu/afs/cs/project/jair/pub/volume4/kaelbling96a-html/node10.html>
- [5] [http://en.wikipedia.org/wiki/Machine\\_learning](http://en.wikipedia.org/wiki/Machine_learning)
- [6] Morten Goodwin Olsen, Ole-Christoffer Granmo, John Oommen and Svein Arild Myrer, "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