Adaptive Algorithms for solving the Knapsack Problem

Bachelor's Project Thesis

Sharif Hamed, s2562677, s.s.Hamed@student.rug.nl,
Supervisors: Dr M.A. Wiering

Abstract: This thesis describes and compares three algorithms for solving the 0-1 knapsack problem. The latter is a combinatorial optimization problem in which the aim is to maximize value with some constraint. The knapsack problem is NP-complete which means there is no algorithm that can solve every knapsack problem in polynomial time, furthermore it is also NP-hard since there is no algorithm that can verify in polynomial time if a solution is optimal. Due to their strong correlations and high coefficients of the elements, the knapsack problems that are used for this thesis are hard to solve. One of the three algorithms to be tested and compared is an evolutionary algorithm named PBIL. This is an already existing algorithm adjusted for this thesis. PBIL is a genetic algorithm that creates a population of knapsack solutions by using a probability distribution. Each iteration this probability distribution is updated and mutated to generate a new knapsack population. The second algorithm that will tested is based on the Boltzmann exploration function. This is a reinforcement/bandit algorithm that uses reward and punishment to learn which solutions are promising to try. As a third algorithm, this thesis presents the Tsetlin Machine. This is a finite state machine/bandit algorithm that learns by updating states. The results show that both Boltzmann and Tsetlin perform significantly better than PBIL in all conditions. The changing independent variables are the knapsack length, the coefficient range and the height of the constraint. Tsetlin performs significantly better than Boltzmann in eight of the nine conditions. Boltzmann performed significantly better than Tsetlin in the condition of high constraint.

1 Introduction

The knapsack problem is a combinatorial optimization problem that is NP-hard (Bernhard and Vygen, 2018). The fact that it is NP-hard means that it is believed that there can exist no algorithm that can solve the problem in polynomial time. If there was such an algorithm then all NP-hard problems can be solved in polynomial time, which is unlikely (Bernhard and Vygen, 2018). The knapsack problem is widely used for testing the performance of new algorithms in their ability to maximise a function. The knapsack problem is already a well established problem that was already researched in the 80s when computers had relatively low computational power (Pirkul, 1987). Today it is less used because it is seen as a relatively easy NP-hard problem. More complex NP-hard problems, such as the traveling salesman problem (TSP) are now more commonly used as test-beds to evaluate the performance of new genetic and other optimization algorithms (Zhou et al., 2019). However, the knapsack is a problem that is much easier to implement and more fundamental because of its simplicity. The reason that the knapsack problem has lost popularity could be due to the very good performance of dynamic programming algorithms. The field of dynamic programming was developed by Richard Bellmann in the 1950s (Bellman, 1952). In essence dynamic programming recursively breaks down some problem in sub problems. To be applicable, there must be sub problems nested in the main problem. Solving these sub problems dramatically reduces the time it takes to solve a problem like the knapsack problem (Sniedovich, 2010). Dynamic programming can solve the knapsack problem in so called pseudopolynomial time. This means that the time complexity is in the order of some polynomial of the input length, but also in the order of some polynomial of one or several of the input values (Kellerer et al., 2004). Because of the existence of these pseudopolynomial algorithms the knapsack problem is called weakly NP-hard. It may seem because of this that the knapsack problem is already solved but there are still hard problems even for dynamic programming algorithms. By increasing the correlation between the elements in the knapsack and increasing the coefficient range, the knapsack problem can be made much harder (Pisinger, 2005).

A knapsack problem has a set of N elements. Each element can be present in the knapsack (denoted by $k_i = 1$) or can be absent ($k_i = 0$). Every element also has some weight $w_i \in \mathbb{N}$ and a value $v_i \in \mathbb{N}$ (in literature also denoted as profit). The global fitness and weight values are:

$$F(k) = \sum_{i=1}^{N} k_i v_i \qquad W(k) = \sum_{i=1}^{N} k_i w_i \tag{1.1}$$

The goal is to maximize the fitness function. The trivial solution would be $k_i = 1 \quad \forall i$. However, there is a constraint $c \in \mathbb{N}$ and the global weight must be smaller than this constraint. Now we are ready to define the problem precisely. We want to find a k such that the following equation holds:

$$F(k) \geq F(k') \quad \forall k' : W(k), W(k') \leq c \tag{1.2}$$

In this thesis three algorithms will be presented and tested on the knapsack problem. The goal of this thesis is to rank the performance of all three algorithms in their ability to maximize the knapsack problem. Furthermore the thesis will try to discuss the mechanics of the algorithms which will make it work better or worse. All three algorithms can be placed somewhere in the field of reinforcement leaning (RL) or in evolutionary computation.

RL is an area of machine learning where the agent or algorithm must try to choose the right actions by learning from its environment. In the literature, learning algorithms are labeled as supervised learning or unsupervised learning. However, when it comes to RL it can not be labeled as supervised learning because we are not getting instructive feedback like: 'x is the right answer', (Sutton and Barto, 2018). Unsupervised learning does not give instructive feedback but tries to find relations and correlations within the data (Chinnamgari, 2019). However, RL is dependant on some reward definition that is provided from the start. The reward and punishment are the only feedback it uses to reinforce some actions over others (Wiering, 1999).

One of the three algorithms is based on the Boltzmann distribution which is a probability measure that is also known as the Gibbs distribution. For that reason the algorithm is called 'Boltzmann'. The Boltzmann algorithm uses the Boltzmann distribution to choose between elements in the knapsack. Furthermore it uses a reward and update function to learn from its actions. The knapsack problem is a simplified environment so that mostly the evaluation function needs to be used. When an algorithm uses RL but its state transitions and transition probabilities are not monitored or used and mostly the evaluation function is important, an algorithm can be labelled as a bandit algorithm (Sutton and Barto, 2018).

The next algorithm is named Tsetlin and is based of the Tsetlin Machine from (Granmo, 2018). The Tsetlin machine is inherently a finite state machine (Tsetlin, 1963) but in this thesis only some of its fundamental properties are used. It is made partly stochastic so that it will inherit the stability from finite state machines but explores more like a stochastic bandit algorithm. The algorithm gives states to the elements of the knapsack. The states partly determine the action. Beside the states, Boltzmann exploration influences the choice of action.

Finally the algorithm named PBIL (Population Based Incremental Learning) will be tested (Baluja, 1994). PBIL is a well established algorithm and is adjusted for the purpose of this thesis to try and compete with Boltzmann and Tsetlin. PBIL is a evolutionary algorithm (EA). An EA is a biologically motivated and inspired adaptive system that is widely used for optimization problems. Regarding PBIL the motivation mostly comes from gene optimization through mutation and natural selection (Liu et al., 2019). PBIL will have an initial population of knapsacks. Every iteration it will update this population by using a probability vector. This probability vector is also updated and mutated.

All three algorithms will be tested on the knapsack problem. First, this thesis will explore whether the algorithms based on RL perform better on the knapsack problem than EA. In other words, whether Boltzmann and Tsetlin perform better than PBIL. Second, this thesis will compare Boltzmann, which is a fully stochastic algorithm, with Tsetlin, which is partly a finite state machine.

2 Methods

All three algorithms have been implemented in C/C++. First PBIL will be explained, then Boltzmann and at last Tsetlin. All three algorithms use a technique where it can only provide knapsack solutions that satisfy the constraint. The internal learning mechanisms of the algorithms are not responsible for keeping the solution within the constraint bound. Alternatively, when there is no action left without making the total weight ($W(k)$ in equation 1.1) overstep the constraint, the algorithm stops adding elements.

2.1 PBIL

PBIL maintains a population of potential solutions, which in this case is a population of assignments k where the initial population is determined randomly.

Let V_t be a probability vector that has N probability entries, one for each element 1 to N. PBIL uses this probability vector to make a new population of knapsacks. Algorithm 2.2 shows how the new population is created. V_t also represents what the algorithm has learned. An entry of V_t stands for the probability that it is chosen individually, so not normalized over the other entries. The way PBIL learns is as follows: It keeps a copy of the best solution found thus far denoted as k^*. Then it uses this solution to change the entries of V_t as shown in equation 2.1.

$$V_{t+1}(i) = V_t(i)(1-\alpha) + k_i^*\alpha \tag{2.1}$$

Equation 2.1 uses some learning parameter $\alpha \leq 1$. When $k_i^* = 0$ then $V_t(i)$ will be decreased but when $k_i^* = 1$ then $V_t(i)$ will be increased. In this way PBIL will learn from the best solution it has made thus far (natural selection). Algorithms 2.1, 2.2, 2.3 and 2.4 are together the PBIL algorithm for the knapsack problem.

Algorithm 2.1 PBIL

```
1: $knapsack \leftarrow$ array length N filled with zeros
2: $P \leftarrow$ empty two dimensional array
3: $V \leftarrow$ probability vector initially all 0.5
4: $I \leftarrow$ Total amount of iterations of the algorithm

5: for $i = 1$ to $I$ do
6:     $P \leftarrow$ createPopulation(V)
7:     $knapsack \leftarrow$ getBestKnapsack(P)
8:     $V \leftarrow$ updateProbVector(knapsack, V)
9:     $V \leftarrow$ mutate(V)
10: end for
11: return $knapsack$
```

Algorithm 2.2 createPopulation(V)

```
1: $N \leftarrow$ Length of the knapsack
2: $P \leftarrow$ empty two dimensional array
3: $Rows \leftarrow$ how many solutions in population
4: $j \leftarrow$ random index between 0 and N
5: $j \leftarrow j + 1$
6: for $i = 1$ to $Rows$ do
7:     for $j = 1$ to $N$ do
8:         $r \leftarrow$ random float between 0 and 1
9:         if $V(j) > r$ then
10:            $P(i, j) \leftarrow 1$
11:        else
12:            $P(i, j) \leftarrow 0$
13:        end if
14:        if $Weight(P(i, \cdot)) > constraint$ then
15:            $j = 0$
16:        end if
17:    end for
18: end for
19: return $P$
```

There is one aspect of the algorithm that has yet to be addressed. Algorithm 2.4 shows that after updating the probability vector there is some mutation. This means that the probability vector entries are randomly changed by some value. This is to ensure that the algorithm will keep exploring and does not make too many solutions that are the same. The pseudo code for the mutation of PBIL has been implemented as in algorithm 2.4. By some mutation probability an element will be decreased by some mutation value. When mutation takes place then with 50% chance the element will be increased by the mutation value (Baluja, 1994).

Algorithm 2.3 updateProbVector(knapsack, V)

```
1:  $N \leftarrow$ Length of the knapsack
2:  $\alpha_1 \leftarrow$ learning rate
3:  $\alpha_2 \leftarrow$ learning rate greater than $\alpha_1$
4:  $\alpha \leftarrow$ float
5:  $F \leftarrow$ fitness evaluation function
6:  $Bknapsack \leftarrow$ best knapsack so far
7:  if $F(Bknapsack) \geq F(knapsack)$ then
8:      $\alpha \leftarrow \alpha_1$
9:  else
10:     $\alpha \leftarrow \alpha_2$
11:     $Bknapsack \leftarrow knapsack$
12: end if
13: for $i = 1$ to $N$ do
14:     $V(i) \leftarrow V(i)(1 - \alpha) + \alpha * Bknapsack(i)$
15: end for
16: return $V$
```

Algorithm 2.4 mutate(V)

```
1:  $N \leftarrow$ Length of the knapsack
2:  $m_v \leftarrow$ mutation value
3:  $m_p \leftarrow$ mutation probability
4:  for $i = 1$ to $N$ do
5:      $r_1 \leftarrow$ random float between 0 and 1
6:      $r_2 \leftarrow$ random float between 0 and 1
7:      if $m_p > r_1$ then
8:          $V(i) \leftarrow V(i)(1 - m_v)$
9:          if $r_2 > 0.5$ and $V(i) + m_v \leq 1$ then
10:             $V(i) \leftarrow V(i) + m_v$
11:         end if
12:     end if
13: end for
14: return $V$
```

2.2 Boltzmann

In general, for bandit algorithms it is challenging to find the best trade off between greedy (exploiting) and exploring behavior. A greedy action is taking the action that has the highest expected reward. An exploring action is one that explores a part of the action space that is not yet explored or explored less than other parts. In essence what the Boltzmann algorithm does is exploring the action space and updating the expected reward (expected value) as seen in algorithm 2.5.

For both exploring and greedy actions some expected value is needed. Just like the probability vector in PBIL, the expected values given by $E_t(i)$ can be seen as the learned values of the Boltzmann algorithm. Some element i having an expected reward of $E_t(i)$ does not have any meaning. What gives meaning is the relative value, so, if some element j has an expected reward such that $E_t(i) > E_t(j)$, then element i is preferred over element j. $E_t(i)$ is updated as follows:

$$E_{t+1}(i) = E_t(i) + \alpha(F_t(k) - \bar{R}_{t-1})(1 - b_i) \qquad \forall i : k_i = 1 \quad (2.2)$$

What is most important when updating $E_t(i)$ is evaluating if we want to reward or punish the knapsack solution. Equation 2.2 uses $(F_t(k) - \bar{R}_{t-1})$ to evaluate if the solution is good or bad. $F_t(k)$ is the fitness/reward received at time t from knapsack k (calculated as in equation 1.1) and $\bar{R}_t$ is the average reward at time t. Equation 2.3 explains how $\bar{R}_t$ is derived where $\bar{R}_0 = 0$.

$$\bar{R}_t = \frac{1}{t}\sum_{i=1}^{t} F_i(x) \quad (2.3)$$

When $k_i = 0$ then updating $E_t(i)$ will be somewhat different as can be seen in equation 2.4. The difference is due to the fact that if k_i switches from one value to another then the operations must be reverted. For example, if $F_t(k) > \bar{R}_{t-1}$ then $E_t(i)$ must be increased for the elements that have $k_i = 1$. However, $E_t(i)$ must be decreased for elements that have $k_i = 0$.

$$E_{t+1}(i) = E_t(i) - \alpha(F_t(k) - \bar{R}_{t-1})b_i \qquad \forall i : k_i = 0 \quad (2.4)$$

As mentioned, each iteration the Boltzmann algorithm (algorithm 2.5) makes use of exploration. There is however still a trade off to be solved between exploration and exploitation. The greedy (exploiting) way of using these values is by choosing the element with the highest expected reward and then the element with the second highest expected reward and so on until the knapsack is filled. When such an algorithm is converging it must overcome local maxima, which is some value that is the maximum that can be reached when the same direction is followed. To overcome these local maxima it must explore more at first instead of being greedy right away. This is solved by using the Boltzmann distribution. This technique of exploring is often referred to as Softmax exploration (Vamplew et al., 2017).

$$b_i = P(i) = \frac{e^{E_t(i)/T}}{\sum_{n=1}^{N} e^{E_t(n)/T}} \quad \forall i : p_i = 1 \qquad (2.5)$$

Equation 2.5 shows that b_i is a probability of some element i to be chosen (set to $k_i = 1$), where p_i denotes if an element can be added. The probability is computed by using the expected value. When some element has a relatively high expected value then the probability of this element b_i will be relatively higher. This makes the algorithm stochastic which in itself already is a way to introduce more exploration. The parameter T in equation 2.5 solves the trade-off. If this parameter increases the probabilities b_i become more random. Per iteration, T will be decreased, inspired by the process of annealing (Rutenbar, 1989) making the Boltzmann algorithm more exploiting per iteration.

In equation 2.5 the value p_i is mentioned. p_i is equal to 1 if it is possible for k_i to be 1. So only elements that are possible to be chosen will be included in the Boltzmann distribution. For example: when the weight of the knapsack is below the constraint and adding some element i will not make it exceed the constraint, then this element is possible.

$$p_i = 1 \quad \forall i : n \neq i, \quad \left(\sum_{n}^{N} k_n w_n \right) + w_i < c \qquad (2.6)$$

Algorithm 2.6 uses binomial properties to select between elements using the Boltzmann distribution. We see two functions that are not further explained, *updateBoltzmannD* which is the procedure of calculating b_i as in equation 2.5 and *updatePos* which updates the array P which gives for each element the value 1 if this element can be added without violating the constraint (equation 2.6).

Algorithm 2.5 Boltzmann

```
1: $N \leftarrow$ Length of knapsack
2: $E \leftarrow$ Expected values elements initially 0
3: $knapsack \leftarrow$ array length N filled with zeros
4: for $i = 1$ to $I$ do
5:     $knapsack \leftarrow$ expl($N$,knapsack,constraint)
6:     $E \leftarrow$ updateExpValue(knapsack)
7: end for
8: return $knapsack$
```

Algorithm 2.6 expl(N, knapsack)

```
1: $constraint \leftarrow$ weight constraint
2: $P \leftarrow$ array with possible elements
3: $B \leftarrow$ Boltzmann distribution
4: for $j = 1$ to $N$ do
5:     $s \leftarrow 0$, (cumulative probability)
6:     $r \leftarrow$ random integer
7:     for $i = 1$ to $N$ do
8:         if $B(i) + s > r$ and $P(i) = 1$ then
9:             $knapsack(i) \leftarrow 1$
10:            $P \leftarrow$ updatePos(constraint)
11:            $B \leftarrow$ updateBoltzmannD(P)
12:            break
13:        else
14:            $s \leftarrow s + B(i)$
15:        end if
16:    end for
17: end for
18: return $knapsack$
```

Algorithm 2.7 updateExpValue(knapsack)

```
1: $N \leftarrow$ Length of knapsack
2: $E \leftarrow$ Expected values per element
3: $F \leftarrow$ fitness/reward of solution
4: $\bar{R} \leftarrow$ average reward
5: $B \leftarrow$ Boltzmann distribution
6: $\alpha \leftarrow$ learning rate
7: for $i$ to $N$ do
8:     if $knapsack(i) = 1$ then
9:         $E(i) \leftarrow E(i) + \alpha * (F - \bar{R}) * (1 - B(i))$
10:    else
11:        $E(i) \leftarrow E(i) - \alpha * (F - \bar{R}) * B(i)$
12:    end if
13: end for
14: return $E$
```

2.3 Tsetlin

The Tsetlin machine is a finite state machine and was first developed by M.L Tsetlin in the Soviet Union in the 1960s (Tsetlin, 1963). The idea is argued to be even more fundamental than the artificial neuron. Ole-Christoffer Granmo found good results in using the Tsetlin machine in pattern recognition. In his paper (Granmo, 2018) he describes the Tsetlin machine in much detail. This paper gives two fundamental properties of the Tsetlin machine that for a big part also lie at the core of the Tsetlin algorithm that was implemented in this thesis. Here we have them quoted:

1. "The current state of the automaton decides which action to perform. The automaton has $2Q$ states. Action 1 is performed in the states with index 1 to Q, while Action 2 is performed in states with index $Q + 1$ to $2Q$."

2. "The state transitions of the automaton govern learning. One set of state transitions is activated on reward, and one set of state transitions is activated on penalty. As seen, rewards and penalties trigger specific transitions from one state to another, designed to reinforce successful actions (those eliciting rewards)."

When we mention the Tsetlin algorithm we refer to the algorithm developed for this thesis which must not be confused with the Tsetlin machine. The proposed Tsetlin algorithm does follow rules that the Tsetlin machine is based on for a big part but not completely. Our Tsetlin algorithm uses the mentioned states, $S_t(i)$, that will give the state of some element i. $S_t(i) \in (1, 2Q)$. Secondly $k_i = 1$ is the same as an 'action' of selecting an item or $k_i = 0$ not selecting an item.

While in our Tsetlin algorithm it is true that:

$$k_i = 0 \quad \forall i : S_t(i) \leq Q \tag{2.7}$$

it is not true that:

$$k_i = 1 \quad \forall i : S_t(i) > Q \tag{2.8}$$

This has to do with the fact that we are working under constraints and we do not let the knapsack exceed this constraint. Instead of only relying on the states, the algorithm uses a small "heuristic" which is knowing when no more elements can be added. This is why equation 2.8 can not be held true. This means that there is some extra technique needed to choose between all the elements $i : S_t(i) > Q$. Our Tsetlin algorithm uses the Boltzmann exploration to choose between these elements and instead of using expected value as done in equation 2.5 it takes the states $S_t(i)$ in order to get the probabilities. We see in equation 2.9 how the Boltzmann distribution is calculated our Tsetlin algorithm.

$$t_i = P(i) = \frac{e^{S_t(i)/T}}{\sum_{n=1}^{N} e^{S_t(n)/T}} \quad \forall i : p'_i = 1 \tag{2.9}$$

In our Tsetlin algorithm the possible values are calculated as follows:

$$p'_i = 1 \quad \forall i : p_i = 1 \text{ and } S_t(i) > Q \tag{2.10}$$

The only difference with equation 2.6 is that in equation 2.10 the elements need to have a state higher than Q.

In the Tsetlin algorithm there are two sets of state transitions where one is activated on reward and the other on punishment. Equation 2.11 and 2.12 are the state transitions when the algorithm is rewarded.

$$S_{t+1}(i) = S_t(i) + 1, \forall i : S(i) > Q \tag{2.11}$$

$$S_{t+1}(i) = S_t(i) - 1, \forall i : S(i) \leq Q \tag{2.12}$$

Equation 2.13 and 2.14 are the state transitions when the algorithm is punished.

$$S_{t+1}(i) = S_t(i) - 1, \forall i : S(i) > Q \tag{2.13}$$

$$S_{t+1}(i) = S_t(i) + 1, \forall i : S(i) \leq Q \tag{2.14}$$

When an element's state is on one side of Q ($i \leq Q$ or $i > Q$) then a reward would mean that it is moved further away from the boundary state value (Q). Equations 2.11 and 2.12 move the element state further away from Q while equations 2.13 and 2.14 move the element states closer to Q (punishment).

In order to choose if the algorithm should be punished or rewarded it makes use of a so called stochastic fitness $\tilde{F}$ and the average reward $\bar{R}$. The average reward $\bar{R}$ is calculated with a parameter α to make sure that later findings will influence the algorithm more:

$$\bar{R}_t = \bar{R}_{t-1} + \alpha(\tilde{F}_t - \bar{R}_{t-1}) \tag{2.15}$$

The stochastic fitness is a solution to the problem that finite state machines can be dependant on the initial settings and may therefore be trapped in a local maximum. In order to overcome this we use the average reward in equation 2.15, but this is not enough. To make the algorithm less dependant on the initial settings and more exploring as a whole we introduce a stochastic evaluation function of the fitness (Granmo et al., 2007). Previously the fitness F was calculated as in equation 1.1. Now the stochastic fitness is calculated as follows:

$$\tilde{F}_t = F_t \pm r \quad : r \in (0, \tau F) \tag{2.16}$$

Each iteration the fitness is by a 50/50 chance increased or decreased by some random number r, which is a number between zero and a ratio of the actual fitness (making it stochastic). The ratio is dependant on a parameter $\tau < 1$. When this parameter is high the stochastic fitness can differ much from the actual fitness. When the parameter gets lower the stochastic fitness comes closer to the actual fitness. Our Tsetlin algorithm uses annealing for this parameter to make the algorithm initially explore more and to be independent from the initial settings.

The main algorithm 2.8 shows that Tsetlin uses exploration and updating of states. The exploration is similar to the exploration in Boltzmann algorithm 2.6 but uses an extra condition when calculating the possible values which was explained in equation 2.10. In algorithm 2.9 first the adjusted Boltzmann distribution is computed. After this there is an extra part where the algorithm is punished for under-packing. What this means is that due to the strict boundary Q there are situations that the knapsack solution will neglect elements that can normally be added without violating the constraint. To solve this the algorithm checks if there is an element that could have been added to the knapsack solution by using equation 2.6. When there is an element that is wrongly neglected then this state is punished.

In the our Tsetlin algorithm the main goal is to combine the strength of stochastic optimization and finite state machines. A stochastic method has more possibilities of exploring the action space and there are many possible ways to avoid local maxima of which annealing is a popular one. Finite state machines are in contrast much more deterministic.

Algorithm 2.8 Tsetlin

```
1: $N \leftarrow$ Length of knapsack
2: $S \leftarrow$ states initially random between 1 and 2Q

3: $knapsack \leftarrow$ array length N filled with zeros
4: $constraint \leftarrow$ weight constraint
5: for $i = 1$ to $I$ do
6:     $knapsack \leftarrow$ ExplT($N$,knapsack,constraint)
7:     $S \leftarrow$ UpdateStates(knapsack)
8: end for
9: return $knapsack$
```

Algorithm 2.9 ExplT(N, knapsack, constraint)

```
1: $P' \leftarrow$ array with possible elements, equation 2.10

2: $T \leftarrow$ Tsetlin distribution as in equation 2.9
3: for $j = 1$ to $N$ do
4:     $s \leftarrow 0$, (cumulative probability)
5:     $r \leftarrow$ random such that $0 < r < 1$
6:     for $i = 1$ to $N$ do
7:         if $T(i) + s > r$ and $P'(i) = 1$ then
8:             $knapsack(i) \leftarrow 1$
9:             $P' \leftarrow updatePos(knapsack, Constraint)$(Eq 2.10)

10:            $T \leftarrow updateBoltzmannD(P')$(Eq 2.9)
11:            $break$
12:        else
13:            $s \leftarrow s + T(i)$
14:        end if
15:    end for
16: end for
17: /*Punish for under-packing:*/
18: for $i = 1$ to $N$ do
19:     if $P(i) = 1$ (Eq 2.6) then
20:         if $knapsack(i) = 1$ and $S(i) > Q$ then
21:             $S(i) \leftarrow S(i) - 1$
22:         end if
23:         if $knapsack(i) = 0$ and $S(i) \leq Q$ then
24:             $S(i) \leftarrow S(i) + 1$
25:         end if
26:         /* $k_i = 0$ and $S(i) > Q$ not punished */
27:     end if
28: end for
29: return $knapsack$
```

Algorithm 2.10 updateStates(knapsack)

```
1:  $S \leftarrow$ state values per element
2:  $\hat{F} \leftarrow$ stochastic fitness
3:  $\bar{R} \leftarrow$ average reward
4:  for $i \leftarrow 1$ to $N$ do
5:    if $\hat{F} > \bar{R}$ then
6:      if $knapsack(i) = 1$ and $S(i) > Q$ then
7:        $S(i) \leftarrow S(i) + 1$
8:      end if
9:      if $knapsack(i) = 0$ and $S(i) \leq Q$ then
10:       $S(i) \leftarrow S(i) - 1$
11:     end if
12:   else
13:     if $knapsack(i) = 1$ and $S(i) > Q$ then
14:       $S(i) \leftarrow S(i) - 1$
15:     end if
16:     if $knapsack(i) = 0$ and $S(i) \leq Q$ then
17:       $S(i) \leftarrow S(i) + 1$
18:     end if
19:   end if
20: end for
21: return $S$
```

3 Experimental Setup

The tests are run on Linux Ubunto where all the code including the knapsack problem is written in C/C++. The three different algorithms will be tested on the knapsack problem. Fully random knapsack problems do not serve as good test cases because, first there are too many experimental variables that can influence the outcome. Second, fully random knapsack problems can be easy to solve which will be explained in this section.

The set of knapsack problems that are used are from (Pisinger, 2005). This set of knapsack problems are generated so that the values v and the weights w are strongly correlated and have high range intervals for the coefficients. High range coefficient intervals mean that the values v and w have a very high range of possibilities. Furthermore, the optimal solutions are given together with the corresponding knapsack problems*.

Fully randomly generated knapsack problems are uncorrelated. This means that there is not much or no correlation between the values and weights. This in turn means that there can be a large difference between the values and weights. The large difference causes the algorithms to learn fast and correctly when some item is clearly important or when it is not. Consider an item that has a value/weight ratio of $\frac{1}{100}$, while the other items have a much higher ratio. This situation is possible in uncorrelated knapsack problems and is clearly an item to set to 0 ($k_i = 0$). In contrast, the test cases used in this experiment are strongly correlated. They are made in such a way that the weights are chosen randomly but the values are a function of the weights. These problems have a small variation between weight/value ratios, making the optimal solution harder to find (Pisinger, 2005).

Three independent parameters (controlled input) will be changed: the length N, the coefficient interval range R and the percentage of items that can be chosen (C^*).

The length N is the length of the knapsack problem. This is the same length that has been used throughout this thesis. When N is not the controlled input the value will be $N = 50$.

As mentioned before, the weights will be chosen randomly. The weight can be randomly chosen from some interval R. When R is not the controlled input the value will be $R = 10^6$ ($w \in (0, 10^6)$).

When the constraint is higher relative to the coefficients, more elements can be chosen. The optimal solution of the knapsack problem k^* will have a number of elements that are chosen. These chosen elements can be expressed as a percentage of all the elements:

$$C^* = 100 \frac{\sum_{i=1}^{N} k_i^*}{N} \tag{3.1}$$

The value of C^* can be calculated because the optimal solution is given together with the knapsack problems. To be clear, the relative constraint C^* is not equal to the actual constraint c which is expressed in a precise number.

3.1 Condition 1: Increasing N

In this experiment only the knapsack length will be increased keeping all other variables constant. R will be set to 10^6, the relative constraint will be ≈ 40 and $N \in \{50, 100, 200\}$. For each of the three values of N every algorithm will run 100 times. Each run will give the maximum fitness the algo-

*`http://hjemmesider.diku.dk/~pisinger/codes.html`

rithm can find. The mean value of the 100 fitness outputs will be calculated for each algorithm.

3.2 Condition 2: Increasing R

In this experiment we will keep the relative constraint set at $\approx$ 40 and the length N set at 50. The ranges will be: $R \in \{10^6, 10^7, 10^8\}$. For each of the three values of R every algorithm will run 100 times. Each run will give the maximum fitness the algorithm can find. The mean value of the 100 fitness outputs will be calculated for each algorithm.

3.3 Condition 3: Increasing C^*

In this experiment the length N will be set to 50 and the interval range will be set to $R = 10^6$. The relative constraints will be approximately: 20%, 40% and 80%. For each of the three values of C^* every algorithm will run 100 times. Each run will give the maximum fitness the algorithm can find. The mean value of the 100 fitness outputs will be calculated for each algorithm.

3.4 Settings and Formal Testing

The parameters of the three algorithms can be observed in the appendix. Each algorithm will do 10000 evaluations per knapsack problem.

We will make use of the Kolmogorov Smirnov test to test the difference of the fitness distributions without assuming a normal distribution and without assuming equal variance. We will use a significance level of $p = 0.025$. The Kolmogorov-Smirnov test measures the distance between distributions. In this test the null hypothesis (H_0) is that the mean values of $Dist_1$ and $Dist_2$ are drawn from the same distribution. The alternative hypothesis (H_1) is that the mean value of $Dist_1$ is drawn from a distribution that is greater than the distribution of $Dist_2$.

4 Results

In table 4.1 the independent variable is N. It is not possible to use the same knapsack problem and increase the length. For that reason the same knapsack problems are used in the cases where N is equal. For example, row 1, 2 and 3 used the same knapsack problem for testing. The mean values of the fitness distributions are denoted by $\bar{F}$ and the standard deviation is denoted as σ. Furthermore, 'Alg comparison' shows which two algorithms are compared in the corresponding rows.

Table 4.1: Increasing knapsack length N.

		N	$\bar{F}(10^{-3})$	$\sigma(10^{-3})$
1	PBIL	50	2125.53	38.22
2	Bolt	50	2194.00	6.38
3	Tset	50	2199.06	0.14
4	Maximum	50	2199.18	-
5	PBIL	100	2394.67	22.98
6	Bolt	100	2469.22	1.35
7	Tset	100	2470.25	0.13
8	Maximum	100	2470.35	-
9	PBIL	200	1948.67	20.72
10	Bolt	200	2063.67	3.92
11	Tset	200	2070.49	4.81
12	Maximum	200	2075.26	-
13	Alg comparison	N	p-value	
14	Tset-PBIL	50	2.2e-16	
15	Tset-Bolt	50	2.2e-16	
16	Bolt-PBIL	50	2.2e-16	
17	Tset-PBIL	100	2.2e-16	
18	Tset-Bolt	100	2.2e-16	
19	Bolt-PBIL	100	2.2e-16	
20	Tset-PBIL	200	2.2e-16	
21	Tset-Bolt	200	2.2e-16	
22	Bolt-PBIL	200	2.2e-16	

Table 4.2: Increasing coefficient range R.

		R	$\bar{F}(10^{-3})$	$\sigma(10^{-3})$
1	PBIL	10^6	3291.02	43.98
2	Bolt	10^6	3400.40	14.45
3	Tset	10^6	3411.04	.18
4	Maximum	10^6	3411.18	-
5	PBIL	10^7	5346.81	108.65
6	Bolt	10^7	5537.23	.76
7	Tset	10^7	5537.37	.75
8	Maximum	10^7	5538.11	-
9	PBIL	10^8	113863.97	1484.58
10	Bolt	10^8	116990.15	18.42
11	Tset	10^8	117001.48	4.51
12	Maximum	10^8	117005.94	-
13	Alg comparison	R	p-value	
14	Tset-PBIL	10^6	2.2e-16	
15	Tset-Bolt	10^6	2.2e-16	
16	Bolt-PBIL	10^6	2.2e-16	
17	Tset-PBIL	10^7	2.2e-16	
18	Tset-Bolt	10^7	0.02431	
19	Bolt-PBIL	10^7	2.2e-16	
20	Tset-PBIL	10^8	2.2e-16	
21	Tset-Bolt	10^8	3.21e-09	
22	Bolt-PBIL	10^8	2.2e-16	

Table 4.3: Increasing C^*.

		C^*	$\bar{F}(10^{-3})$	$\sigma(10^{-3})$
1	PBIL	20	228.52	6.82
2	Bolt	20	235.81	.62
3	Tset	20	235.99	.98
4	Maximum	20	236.12	-
5	PBIL	40	29.32	12.98
6	Bolt	40	956.04	2.61
7	Tset	40	958.12	0.05
8	Maximum	40	958.17	-
9	PBIL	80	3261.30	13.48
10	Bolt	80	3285.36	.006
11	Tset	80	3285.35	.02
12	Maximum	80	3285.37	-
13	Alg comparison	C	p-value	
14	Tset-PBIL	20	2.2e-16	
15	Tset-Bolt	20	3.571e-05	
16	Bolt-PBIL	20	2.2e-16	
17	Tset-PBIL	40	2.2e-16	
18	Tset-Bolt	40	2.2e-16	
19	Bolt-PBIL	40	2.2e-16	
20	Tset-PBIL	80	2.2e-16	
21	Tset-Bolt	80	1	
22	Bolt-PBIL	80	2.2e-16	
23	**Bolt-Tset**	**80**	**2.2e-16**	

Table 4.1 and 4.2 show that for each value of N and R the mean fitness value of Tsetlin is significantly greater than that of Boltzmann and PBIL. Furthermore, table 4.1 and 4.2 show that for each value of N and R the mean fitness value of Boltzmann is significantly greater than that of PBIL.

From table 4.3 it can be deriven that C^* has some influence on the relative performance of the algorithms. In the conditions $C^* = 20$ and $C^* = 40$ the mean fitness value of Tsetlin is significantly greater than that of Boltzmann and PBIL. However, when $C^* = 80$ we see that the mean fitness value of Tsetlin is significantly smaller than that of Boltzmann. The high C^* condition is the only situation where Tsetlin does not have the greatest mean fitness value.

Table 4.1, 4.2 and 4.3 show that Boltzmann and Tsetlin have a very small standard deviation compared to PBIL. Six out of the nine experiments Tsetlin has the smallest standard deviation. Both Tsetlin and Boltzmann have a smaller standard deviation than PBIL in every experiment.

The mean fitness values of Tsetlin and Boltzmann are close to the maximum in each experiment wheres PBIL was not. We further note that each comparison has significant results while the difference between Tsetlin and Boltzmann is not always large. In table 4.2 when $R = 10^7$ the mean fitness values of Tsetlin and Boltzmann are very close to each other with respect to there standard deviation. However, the difference is still measured to be significant. A similar point can be made when looking at table 4.3 in the condition of $C^* = 20$.

5 Discussion

First it will be discussed why Tsetlin does not always perform better than Boltzmann. Tsetlin uses rules that make it impossible to choose an element that has a state lower or equal to Q. In the conditions $C^* = 80$ about 80% percent of the elements need to be set to 1 in order to find the optimal solution. This means that Tsetlin must exclude fewer elements. This can have the consequence that Tsetlin will perform worse because too many elements are excluded each time. What makes Tsetlin perform well is that is can exclude elements so that there are fewer choices. What happens when C^* increases is that most elements need to be chosen. this means that the excluding factor will be of less use for the Tsetlin algorithm. What then happens is that Tsetlin will become more like the Boltzmann algorithm but with less differences in the Boltzmann distribution since $S(i) \in (1, 2Q)$ and $E(i) \in (0, M)$ Where $M >> 2Q$.

Second, the low standard deviation of Tsetlin and Boltzmann can be explained by the fact that the solutions of these algorithms are close to the maximum. When the computed solutions are always close tot the maximum the standard deviation can not be large.

Third, when looking at the p-values one could ask why they are always significant. Before starting the experiments we chose to run the algorithms 100 times on each knapsack problem with 10000 iterations. The amount of runs and iterations could be the reason for the low p-values for increasing the runs we increase the statistical power. We assumed non-normal distributions and non-equal variances so the Wilcoxon rank sum test and the Welch Two Sample t-test are inappropriate to use. However, there are two comparisons where we could doubt the outcome of the Kolmogorov Smirnov test, such as in the the conditions of $R = 10^7$ and $C^* = 20$ which where mentioned in section 4.

6 Conclusion and Further Work

This thesis first explained the knapsack problem and some of the related work. Then this thesis introduced three algorithms: first, PBIL which is an evolutionary algorithm. Second, Boltzmann which is a bandit algorithm. Third, Tsetlin which is a finite state machine/bandit algorithm. These three algorithms were tested on the knapsack problem to evaluate the performance of the algorithms and compare them to each other. The results showed that Tsetlin performs better than Botzmann and PBIL in every condition except for the high constraint condition.

These findings suggest that, when it comes to the knapsack problem, algorithms based on reinforcement learning perform better than evolutionary algorithms. What these findings also suggests is that Tsetlin on average seems to outperform both Boltzmann and PBIL.

For further research it could be interesting to test our Tsetlin algorithm on other problems like the traveling salesman problem. When given some city, the states of the Tsetlin algorithm could determine the next city to visit or exclude some optional cities.

References

Baluja, S. (1994). Population-based incremental learning. a method for integrating genetic search based function optimization and competitive learning. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Dept Of Computer Science.

Bellman, R. (1952). On the theory of dynamic programming. *Proceedings of the National Academy of Sciences of the United States of America*, 38(8):716.

Bernhard, K. and Vygen, J. (2018). Combinatorial optimization: Theory and algorithms. *Berlin, Germany: Springer.*

Chinnamgari, S. (2019). *R machine learning projects : Implement supervised, unsupervised, and reinforcement learning techniques using r 3.5.* Packt Publishing.

Granmo, O.-C. (2018). The tsetlin machine-a game theoretic bandit driven approach to optimal pattern recognition with propositional logic. *arXiv preprint arXiv:1804.01508.*

Granmo, O.-C., Oommen, B. J., Myrer, S. A., and Olsen, M. G. (2007). 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)*, 37(1):166–175.

Kellerer, H., Pferschy, U., and Pisinger, D. (2004). Knapsack problems. Berlin etc.: Springer.

Liu, J., Abbass, H., and Tan, K. (2019). *Evolutionary computation and complex networks.* Cham, Switzerland: Springer.

Pirkul, H. (1987). A heuristic solution procedure for the multiconstraint zero-one knapsack problem. *Naval Research Logistics (NRL)*, 34(2):161–172.

Pisinger, D. (2005). Where are the hard knapsack problems? *Computers & Operations Research*, 32(9):2271–2284.

Rutenbar, R. A. (1989). Simulated annealing algorithms: An overview. *IEEE Circuits and Devices magazine*, 5(1):19–26.

Sniedovich, M. (2010). *Dynamic programming: foundations and principles.* CRC press.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement learning: An introduction.* MIT press.

Tsetlin, M. L. (1963). Finite automata and models of simple forms of behaviour. *Russian mathematical surveys*, 18:1–27.

Vamplew, P., Dazeley, R., and Foale, C. (2017). Softmax exploration strategies for multiobjective reinforcement learning. *Neurocomputing*, 263:74–86.

Wiering, M. A. (1999). *Explorations in efficient reinforcement learning.* PhD thesis, University of Amsterdam.

Zhou, A.-H., Zhu, L.-P., Hu, B., Deng, S., Song, Y., Qiu, H., and Pan, S. (2019). Traveling-salesman-problem algorithm based on simulated annealing and gene-expression programming. *Information*, 10(1):7.

A Appendix

Table A.1: Parameters of PBIL

	Parameter	Meaning	Value
1	α_1	Learning rate 1	0.025
2	α_2	Learning rate 2	0.0125
3	I	Amount of iterations	1000
4	$Rows$	Size population	10
5	m_v	Mutation value	0.05
6	m_p	Mutation probability	0.02

Table A.2: Parameters of Boltzmann

	Parameter	Meaning	Value
1	α	Learning rate	0.00001
2	I	Amount of iterations	10000
3	T_{start}	Temperature start	40000
4	T_{end}	Temperature end	1000
5	$T_{decrease}$	Temperature decrease	0.95

Table A.3: Parameters of Tsetlin

	Parameter	Meaning	Value
1	α	Parameter average reward	0.5
2	I	Amount of iterations	10000
3	τ_{start}	Stochastic fitness $\tilde{F}$ start	0.9995
4	τ_{end}	Stochastic fitness $\tilde{F}$ end	0.0001
5	$\tau_{decrease}$	Stochastic fitness $\tilde{F}$ decrease	0.999
6	T_{start}	Temperature start	50000
7	T_{end}	Temperature end	10
8	$T_{decrease}$	Temperature decrease	0.9995
9	$2Q$	Number of states	120