MONTAN
UNIVERSITÄT
WWW.UNILEOBEN.AC.AT

Master Thesis

Monte Carlo Tree Search for Job Shop Scheduling Problems

submitted to the

Montanuniversität Leoben

created at

Lehrstuhl für Informationstechnologie

Submitted by:

Reichenhauser Catrin Samira, BSc
m1035003

Supervisor/Expert:

Ortner Ronald, assoz.Prof. Mag.phil. Mag. et Dr.rer.nat.
Montanuniversität Leoben

Leoben, on November 9, 2017

Danksagung

An dieser Stelle möchte ich mich gerne bei all jenen bedanken, die mich während des Schreibens meiner Masterarbeit und des gesamten Studiums unterstützt und begleitet haben.

Besonderer Dank gilt meinem Betreuer assoz.Prof. Mag.phil. Mag. et Dr.rer.nat Ronald Ortner, der für meine Fragen immer ein offenes Ohr hatte.

Ich möchte auch meinen Freunden, die ich während meines Studiums kennen gelernt habe und die mich auf diesem Weg begleitet haben, Dank aussprechen.

Vor allem aber bedanke ich mich herzlich bei meinen Eltern Erika und Franz Reichenhauser, meinem Bruder Romeo Reichenhauser und meinem Partner Manuel Petersmann für die liebenswerte Unterstützung.

Kurzfassung

Scheduling-Probleme sind Problemstellungen, denen man häufig in der Industrie begegnet. Beispiele dafür sind Personalplanung, Maschinenbelegungsplanung oder auch die Zuweisung von Zügen zu Gleisen. Die Aufgabe ist es, eine bestimmte Anzahl von Objekten einer bestimmten Anzahl von Ressourcen unter Berücksichtigung der entsprechenden benötigten Kapazitäten zuzuweisen. Je größer die Anzahl an Objekten und die Anzahl an Ressourcen ist, desto schwieriger wird es, eine solche Zuteilung zu finden. Außerdem erschweren oft zusätzliche Randbedingungen das Lösen eines solchen Scheduling-Problems.

Heutzutage wird versucht, Scheduling-Probleme mittels unterschiedlicher Algorithmen zu lösen, um Zeit, Kosten oder auch Energie zu sparen. In dieser Arbeit wird Monte Carlo Tree Search, eine Methode des Reinforcement Learning, angewandt, um speziell Job Shop Scheduling Probleme zu lösen. Dabei werden zwei unterschiedliche Evaluierungsmethoden (Threshold Ascent und Upper Confidence Bound for Trees) getestet und miteinander verglichen. Schließlich werden die gefundenen Schedules mit den optimalen verglichen und Aussagen über ihre Effektivität und Effizienz getroffen.

Abstract

Scheduling problems are among the most common problems in industry. They deal with the allocation of a number of objects to a number of resources. Examples are human resource planning, machine scheduling, or the allocation of arriving trains to station platforms. These problems can be simple, if for example the number of objects and resources is very small and if there are no further constraints. But the higher the number of objects or resources and the more constraints have to be considered, the more difficult the problems become.

Different algorithms try to solve such problems in order to reduce costs, time, or energy and to increase quality and performance. In this master thesis the Reinforcement Learning method Monte Carlo Tree Search is used for solving Job Shop Scheduling problems. In particular, as evaluation functions we use the bandit algorithms Upper Confidence Bound for Trees and Threshold Ascent. These methods are tested on a set of Job Shop Scheduling problems.

Contents

List of Figures

List of Tables

Abbreviations

AMPE	Average Mean Percentage Error
BaB	Branch and Bound
FIFO	First In First Out
LB	Lower Bound
LIFO	Last In First Out
MCTS	Monte Carlo Tree Search
MPE	Mean Percentage Error
MWKR	Most WorK Remaining
RL	Reinforcement Learning
SPT	Shortes Processing Time
Std	Standard Deviation
TA	Threshold Ascent
UB	Upper Bound
UCB	Upper Confidence Bound
UCT	Upper Confidence Bound for Trees

1. Introduction

1.1. Motivation

Industry 4.0 has shifted the focus on problems in operations research. Machines are no longer exclusively used as operational tools, but also for the automated execution of control and planning activities. However, it is not only the number of machines, but also the complexity of activities that is increasing. Scheduling problems are a particularly interesting type of problems, that are frequently encountered in industry. The larger the number of objects that need to be allocated to certain resources and the larger the number of resources, the more difficult it is to find an optimal schedule. Exploiting the increasing power of computers, simple processes can be optimized in advance, saving time and thus money. However, many problems of practical interest fall into a class of complexity, where exact algorithms will not run in reasonable time and efficient approximate algorithms or heuristics are required to produce feasible solutions in a reasonable amount of time. In this master thesis the Reinforcement Learning method Monte Carlo Tree Search is investigated for its use for Job Shop Scheduling problems.

1.2. Overview

In Chapter 2, the multi-armed bandit problem and its setting are explained in detail. Two different algorithms for the multi-armed bandit problem are introduced, the Upper Confidence Bound algorithm proposed by Auer et al. [2] and Threshold Ascent proposed by Streeter and Smith [14].

In Chapter 3, Reinforcement Learning (RL) is explained. Since tree structures play a profound role for RL, they are described alongside. Monte Carlo Tree Search (MCTS) is introduced as a method for finding an optimal policy in an RL setting.

Chapter 4 starts with an introduction to general scheduling problems. Next the important sub-category of Job Shop Scheduling problems, on which we focus in this thesis, are highlighted. Furthermore the main adaptations, that have to be made in order to be able to use Monte Carlo Tree Search for Job Shop Scheduling problems are pointed out in detail.

In Chapter 5 the experimental setting is presented. The computational experiments carried out are described in detail, i.e. the numbers of independent experiments, the used/-calibrated parameters and the resulting characteristic values, used for the evaluation of the performance of the algorithms.

In Chapter 6 the results of the computational experiments (for MCTS using UCT and TA) are presented and discussed.

In Chapter 7 data quality is discussed in the context of scheduling.

2. Multi-armed Bandit Problem

Table 2.1.: Notation I - Multi-armed Bandit Problems

Abbreviation	**Explanation**
A	set of arms
a_i	arm i
δ	positive real parameter (error-probability)
i_{best}	best arm
k	number of arms
n	total number of pulls
n_i	number of times arm i has been pulled
Q_i	cumulative reward for arm i
s	number of best memorized rewards for TA
s_i	number of rewards received from arm i that are under the s best
$\bar{X}_i$	mean reward for arm i
μ_i	expected reward of arm i
μ^*	maximum expected value

Setting:

Imagine a slot-machine (environment) and a gambler (the learning system). The slot-machine has several arms (these correspond to actions) that follow a fixed but unknown reward-distribution. Denote $A = \{a_1, \ldots, a_k\}$ as the set of possible actions (arms) and k as the number of possible actions. In each time step $t = 1, 2, \ldots, n$, the gambler can pull one of the arms and receives the corresponding reward. We assume that pulling an arm i gives independent and identically distributed rewards in the interval $[0, 1]$.[3]

Objective:

There are various different objectives one may consider. One objective can be to maximize the expected total reward over n pulls and another one can be to identify the arm that leads to the single best reward.[3]

Let us assume that the gambler's objective is to maximize the total reward over n pulls. In general, various different algorithms exist for the multi-armed bandit problem (see Subsection 2.0.1 and 2.0.2). For each algorithm it is important to keep a balance between exploitation (pulling the arm with the highest reward so far often) and exploration (trying out different arms). Here we investigate two different algorithms. The first is the Upper

Confidence Bound algorithm proposed by Auer et al. [2] and the second one is Threshold Ascent proposed by Streeter and Smith [15].[3]

2.0.1. Upper Confidence Bound (UCB)

We consider a multi-armed bandit setting with the following objective.

Objective:
The objective is to maximize the total reward over n pulls.[3]

Algorithm:
The algorithm proposed by Auer et al. [2] first pulls each arm once. Then for each of the subsequent pulls the UCB-value

$$UCB_i = \bar{X}_i + \sqrt{\frac{2\ln(n)}{n_i}} \tag{2.1}$$

is calculated for each arm i. $\bar{X}_i$ denotes the mean reward gained from arm i so far and n_i denotes the number of times arm i has been pulled. The algorithm pulls the arm with the highest UCB-value and receives the corresponding reward. $\bar{X}_i$ ensures the exploitation of promising arms, whereas the second term in Equation 2.1 encourages the pulling of less-played arms. In Algorithm 1 the corresponding code is presented.[3]

Algorithm 1: Upper Confidence Bound Algorithm [2]

```
Function UCBSelection($n$, $A$)
    foreach arm $i \in A$ do
        $n_i = 0$;
    end
    $counter = 0$;
    while $counter < n$ do
        if $n_i = 0$ then
            $UCB_i = \infty$;
        else
            $\bar{X}_i = \frac{Q_i}{n_i}$;
            $UCB_i = \bar{X}_i + c\sqrt{\frac{2\ln(n)}{n_i}}$;
        end
        $i_{\text{best}} = \arg\max_i UCB_i$;
        pull arm $i_{\text{best}}$;
        $i = i_{\text{best}}$;
        receive reward $r$;
        $n_i = n_i + 1$;
        $Q_i = Q_i + r$;
        $counter$++;
    end
```

To measure the quality of an algorithm one considers how much it loses with respect to the optimal arm. Correspondingly, the regret is defined as:[2]

$$\mu^* n - \mu_i \sum_{i=1}^{k} \mathbb{E}[n_i], \tag{2.2}$$

where μ_i is the expected reward of arm i and let μ^* be defined as follows:[2]

$$\mu^* = \max_{1 \leq \mathrm{i} \leq \mathrm{k}} \mu_i$$

Theorem:
The regret of the UCB-algorithm is upper-bounded by:[2]

$$\left[8 \sum_{i:\mu_i<\mu^*} \left(\frac{\ln n}{\Delta_i}\right)\right] + \left(1 + \frac{\pi^2}{3}\right)\left(\sum_{j=1}^{K} \Delta_j\right), \tag{2.3}$$

where μ_i is the expected reward for arm i, k is the number of arms, and Δ_i is

$$\Delta_i \overset{def}{=} \mu^* - \mu_i. \tag{2.4}$$

2.0.2. Threshold Ascent (TA)

The Threshold Ascent algorithm, first proposed by Streeter and Smith, aims at a multi-armed bandit setting with a different objective.[14]

Objective:
We consider a multi-armed bandit setting. The objective is to maximize the single best reward over n pulls.[14]

Algorithm:
The basic idea is to track the s best rewards and the respective arms that have led to these rewards. Let δ be a positive real parameter, which describes the error probability for confidence intervals implicitly used by TA. Let s_i be the number of rewards received from arm i that are among the s best rewards. Calculate for each arm $i = 1, \ldots, k$ a value $h(s_i, n_i)$ by using the following formula.[14]

$$h(s_i, n_i) = \begin{cases} \dfrac{s_i + \alpha + \sqrt{2s_i\alpha + \alpha^2}}{n_i} & , \quad \text{if } n_i \geq 1 \\ \infty & , \quad \text{if } n_i = 0 \end{cases} \tag{2.5}$$

$$\alpha = \ln\left(\frac{2nk}{\delta}\right) \tag{2.6}$$

Then pull arm i_{best},

$$i_{best} = \max_{1 \leq i \leq k} h(s_i, n_i), \tag{2.7}$$

receive its reward and increment n_i. This procedure is repeated n times. The corresponding code is presented below.

Algorithm 2: Threshold Ascend Algorithm [14]

```
Function TASelection($n$, $s$, $A$)
    $s_i = 0$;
    $n_i = 0$;
    $counter = 0$;
    while $counter < n$ do
        if $n_i = 0$ then
            $h(s_i, n_i) = \infty$;
        else
            $h(s_i, n_i) = \frac{s_i + \alpha + \sqrt{2s_i\alpha + \alpha^2}}{n_i}$;
        end
        $i_{best} = \arg_i \max h(s_i, n_i)$;
        pull arm $i_{\text{best}}$;
        $i = i_{\text{best}}$;
        receive reward $r$;
        $n_i = n_i + 1$;
        if arm $i$ is among $s$ best arms then
            Update the list with the $s$ best arms and the values $s_j$ for the affected arms $j$;
        end
        $counter$++;
    end
```

Parameter s influences the trade-off between exploitation and exploration. If $s = 1$, Threshold ascent behaves like Round-Robin Sampling (see [14]). If $s = \infty$, TA behaves like Chernoff Intervall Estimation (see [14]).

The algorithm of Streeter and Smith works best, when the reward distributions fulfil the following criteria:[14]

1. Let t_{critical} be a threshold, that is very low at the beginning of the algorithm. For all $t > t_{\text{critical}}$ it holds that the arm that is most likely to lead to a $reward > t$ is the same arm that is most likely to yield to a $reward > t_{\text{critical}}$. This arm is denoted as i^*. Note that this is an assumption that may not be fulfilled in general.[14]

2. The gap between the probability that arm i^* leads to a $reward > t$ and the probability that some other arm gives $reward > t$ grows, when t increases beyond t_{critical} (see Figure 2.1). Hence the ratio $\frac{p_{i^*}(t)}{p_i(t)}$, where $p_i(t)$ denotes the probability that the i^{th} arm returns a $reward > t$, should increase as well for any $i \neq i^*$.[14]

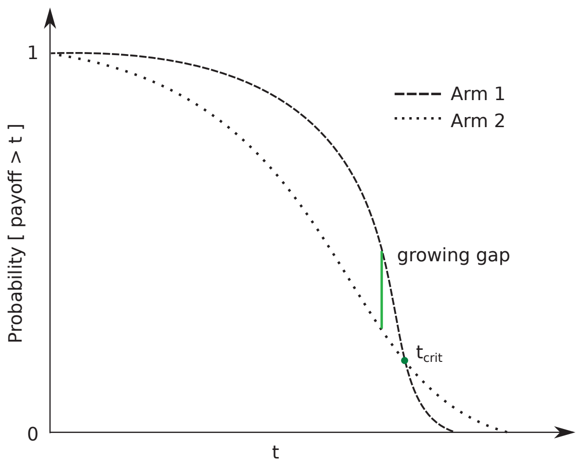

Figure 2.1.: Reward distributions of k-armed bandit instances [14]

3. Monte Carlo Tree Search

Table 3.1.: Notation II - MCTS

Abbreviation	Explanation
A	set of possible actions
a_t	action at time step t
δ	error probability
E	set of edges
G	graph
k	number of possible actions
n	total number of rollouts
$N(y)$	number of visits in node y
$p(s'\|s,a)$	probability for landing in state s' when taking action a in state s
$Q(y)$	total pay-off received from node y
S	set of possible states
s_t	state at time step t
s_{init}	initial state
s_{end}	final state
$S(y)$	number of times y has led to one of the s best rewards
t	a discrete time step ($t \in \{1, 2, \dots\}$)
V	set of vertices
y_0	root-node
y_{next}	node selected by the tree-policy
y'	child node of node y

3.1. Reinforcement Learning

Setting:

A typical Reinforcement Learning (RL) setting consists of a learning-system and an environment. Let S be a set of possible states and let A be a set of actions the learning system can take. Denote $p(s'|s,a)$ as the probability for landing in state s' when taking action a in state s. When choosing action a in state s, one obtains a random reward according to an unknown but fixed reward distribution depending on a and s. The expected reward for taking action a in state s and landing in s is denoted as $r(s,a)$. Note that rewards are independent are identically distributed for (s,a). [16]

Objective:
Find a policy that minimizes or maximizes a certain reward-function (depends on the underlying problem setting).

The learning-system typically interacts with its environment in discrete time steps $t = \{1, 2, 3, \dots\}$. The learning-system starts in an initial state s_{intit} in time step 1. At each time step t the learning-system selects an action a according to a certain policy. A policy is a mapping of states to actions at a certain time step. Consider that at step t the learning-system finds itself in state $s_t \in S$, from where it can choose an action $a \in A(s_t)$ (a set of possible actions that can be taken in state s_t). After choosing action $a_t \in A(s_t)$, the environment reacts, offers a new state and communicates the corresponding reward r_{t+1} in the next time step $t + 1$. As a consequence of its choice the learning-system finds itself in a new state s_{t+1}. [16]

The following picture illustrates the RL setting described above.

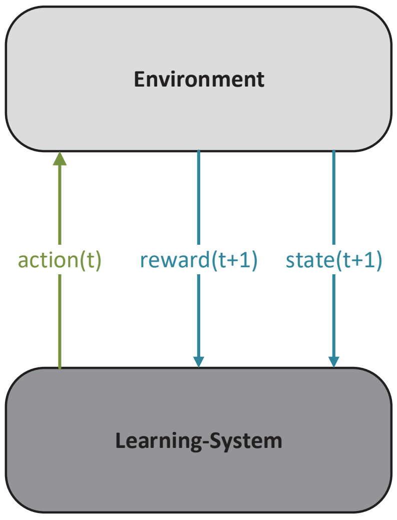

Figure 3.1.: RL setting

3.2. Trees

The following terms are explained following the work of Cormen [6]:

Graph:
Given a set of nodes V, an undirected graph is an ordered pair (V, E), where the set of edges $E \subseteq V \times V$ and edges (u, v) and (v, u) are identified.

Path:
A path in a graph $G = (V, E)$ is a sequence of pairwise disjoint nodes $P = (v_1, v_2, \ldots, v_k)$ with edges between v_i and $v_{\mathrm{i}+1}$ (for $i = 1$ to $i = k - 1$). The length of a path is the number of contained edges.

Circle:
A circle is a path for which the first node v_1 and last node v_k are connected via an edge.

Acyclic:
A graph is acyclic, if the graph does not contain circles.

Connected:
A graph is connected if there is a path between any two nodes.

Tree:
A tree is a connected and acyclic graph.

Rooted tree:
A tree is called rooted tree if one node (the "root") of the tree is distinguished.

In the following, we introduce some terminology for rooted trees. We assume a given rooted tree with all edges directed away from the root of the tree.

Predecessors:
Each node v_i for $i < k$ in a directed path $P = (v_1, v_2, \ldots, v_k)$ starting in the root-node is called predecessor of v_k.

Successor:
Each node v_i for $i > 1$ in a directed path $P = (v_1, v_2, \ldots, v_k)$ starting in the root-node is called successor of v_1.

Parent-node:
A parent-node v of a node v' is the first immediate predecessor of the node v'.

Child-node:
A child-node v' of a node v is the first immediate successor of the node v.

Siblings:
Two nodes are siblings, if they have the same parent-node.

Leaf-node:
A leaf-node is a node that has no child-nodes.

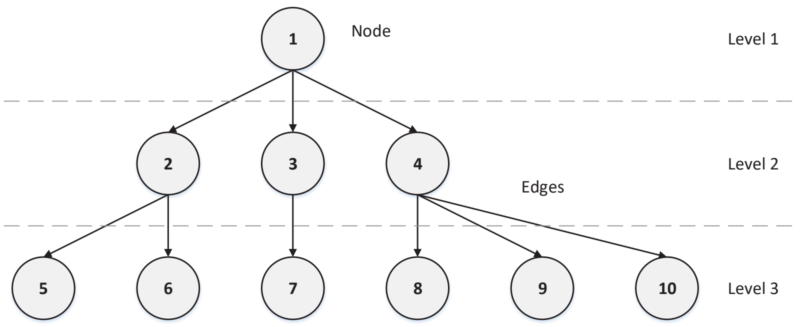

Figure 3.2.: Tree Structure

The tree in Figure 3.2 is a rooted tree. In this case the root is node 1. Each node in the tree of Figure 3.2 has a parent-node except the root node and each node has at least one child-node except the leaf-nodes (i.e. a node can only have one parent-node, whereas it can have more than one child-nodes). In Figure 3.2 each node in level 3 is a leaf-node. For example node 3 is the parent-node of node 7. Therefore node 7 is the child of node 3. Nodes that have the same parent-node are called siblings, like nodes 5 and 6.[17]

3.3. Monte Carlo Tree Search (MCTS)

Subsequently, MCTS is introduced following Browne [3] by taking advantage of the notion of trees. Monte Carlo Tree Search is an RL-method for solving sequential decision problems.

Setting:
The setting corresponds to the RL-setting described in Chapter 3, with the difference

that now there is a predefined set of terminal states. When such a state is reached there are no actions at disposal and the interaction with the environment ends.

Objective:
The objective is to find an optimal policy for selecting actions in order to maximize the total reward.

Algorithm:
Monte Carlo Tree Search is a method of RL. It sequentially generates a tree, with states represented as nodes, and actions represented as edges. We assume that the transitions from one state into another are deterministic. In general this is not a necessary condition in order to apply MCTS. However for our particular application it holds and the description of MCTS is simpler for the case of deterministic transformations. We also consider that the reward is not received directly after taking an action, but only in the terminal states. More precisely, this means that the (deterministic) reward is 0 in all nodes except the leaf nodes. In general MCTS is based on four steps: Select, Expand, Simulate, and Back-up. In the Expand step unvisited child-nodes are selected. In the Select step an evaluation function is used for selecting already-visited child-nodes. Simulation (or roll-out) means randomly choosing a path down the tree until a leaf node (terminal node) is reached. Depending on the evaluation function used different parameters need to be updated in the back-up step (for example the reward or the number of visits of a node). The MCTS-algorithm finishes, either if a terminal-state is reached or if a certain number of rollouts has been executed. The following picture depicts the four steps.[5]

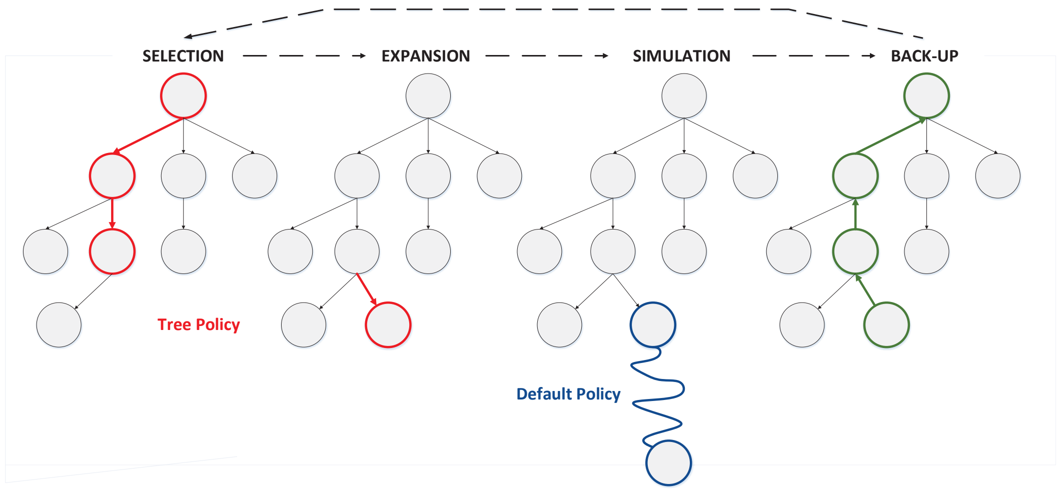

Figure 3.3.: The four major steps in the MCTS algorithm - redesigned from [3].

Let s_{init} be the initial state of the RL-setting (root-node). Starting at this node the tree-policy is executed. This policy covers the selection and the expand step, which sequentially adds child-nodes to the tree. At the very beginning the learning system is in state s_{init} and does not know any of its children, that means the system is not able to choose the "best" action yet and the algorithm starts by applying the expand step first [3].

If each child-node of the root-node has been visited at least once. MCTS starts with the selection step. At each node, where every child-node has been visited at least once (the node is fully expanded), the best child-node is selected by using an evaluation function. The evaluation function gives for a given node the corresponding best child-node. Different approaches for the evaluation of the best child-node exist, but in this work the UCT-algorithm and the TA-method are applied (see Subsection 2.0.1 and 2.0.2). Figure 3.4 visualizes the first step [3].

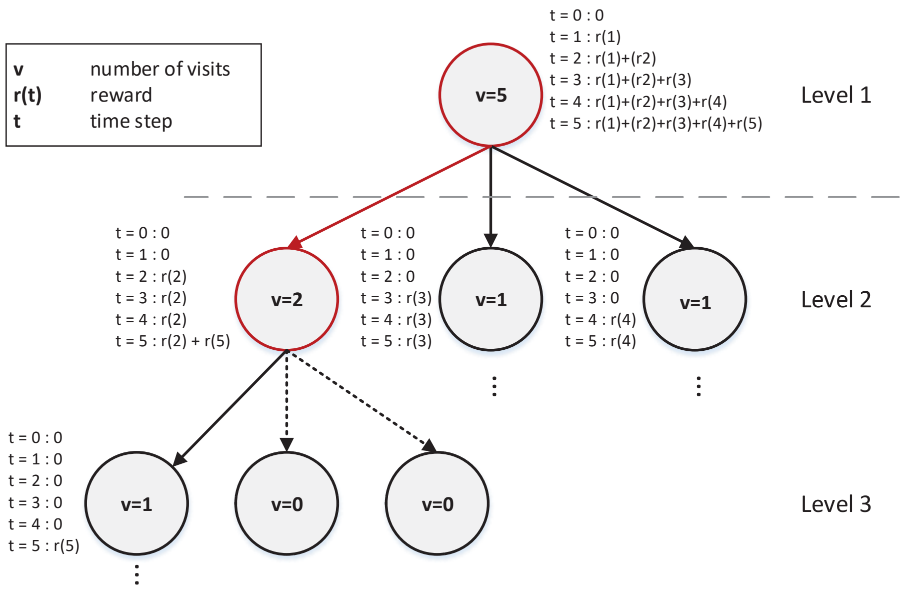

Figure 3.4.: Select Step

In general, MCTS first conducts a Select step and only after a node is reached, where not every child has been visited yet, an Expand step is triggered. That is, as explained before for the initial state, a child with no visits so far is selected (an untried action) and added as a new node to the tree. The action that leads to this child-node is the edge between the child-node and its parent-node. This step is depicted in Figure 3.5 [5].

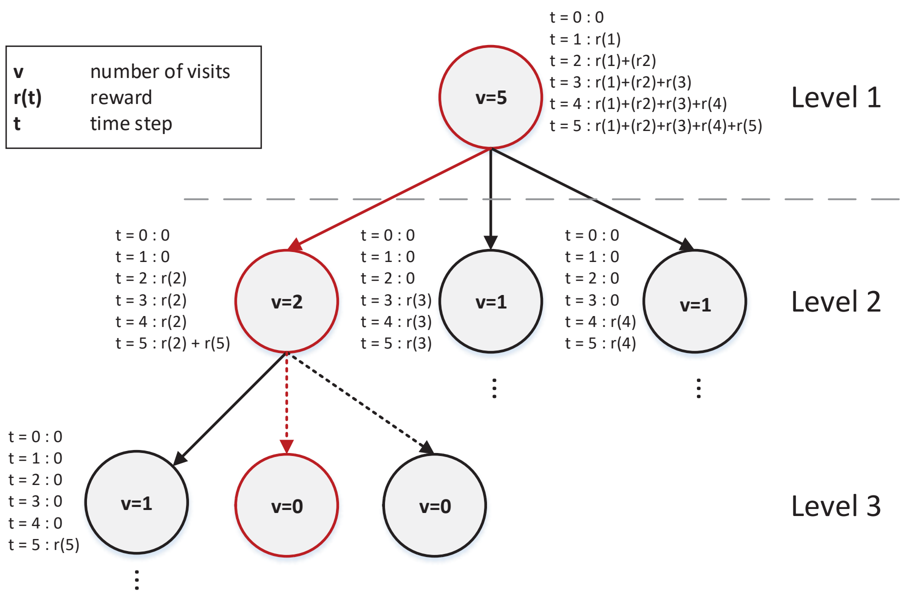

Figure 3.5.: Expand Step

In the Simulation step a random path from the node that has been chosen in the Expand step to a leaf-node is chosen. This step covers the so-called default policy, a random selection of nodes until the terminal node is reached.[5]

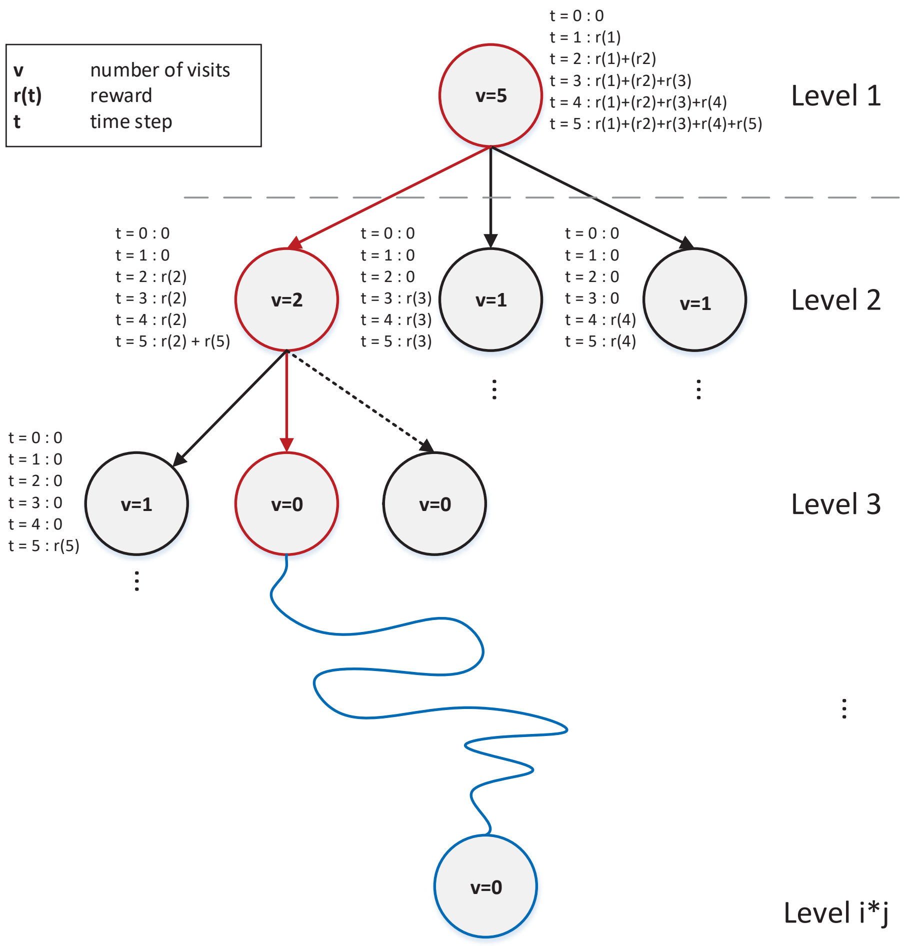

Figure 3.6.: Simulate Step

The last step is called the Back Up step. Nodes that have been selected through the tree policy are updated based on the results of the simulation and the parameters needed for the evaluation function. The following Figure illustrates the Back Up step.[5]

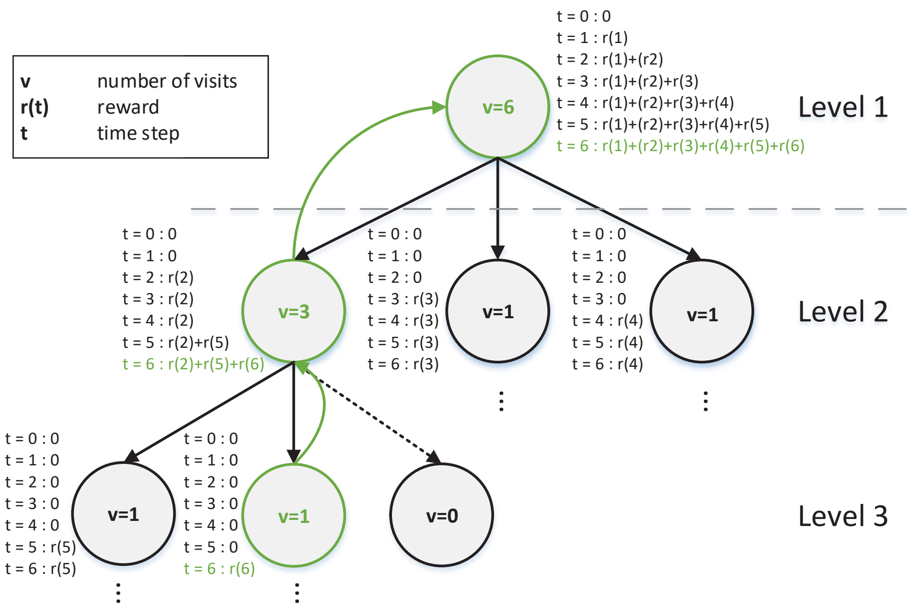

Figure 3.7.: Back Up Step

The more often the four steps (depicted in Figure 3.3) are repeated the more accurate the tree gets, because more actions have been tried out. In practise, a computational budget is defined, which regulates the number of iterations of the MCTS-algorithm (see Algorithm 3). The computational budget typically is adopted to a specific problem considering the total number of time steps or memory efficiency. Hence the main computational steps are presented in Algorithm 3 the following way.[3]

Algorithm 3: MCTS-Algorithm [3]

```
Input: $s_{\text{init}}$, Evaluation($node$), $budget$;
Function MCTS($s_{\text{init}}$)
    create a root-node $y_0 = s_{\text{init}}$;
    $counter = 0$;
    while $counter < budget$ do
        $y_{\text{next}}$ = TreePolicy($y_0$);
        $reward$ = DefaultPolicy($y_{\text{next}}$);
        BackUp($y_{\text{next}}$, $r$);
        counter++;
    return $a$(Select($y_0$));
Function TreePolicy($y$)
    while $y$ is $non-terminal$ do
        if $y$ is not fully expanded then
            return Expand($y$);
        else
            $y$ = Select($y$);
    return x;
Function Expand($y$)
    choose an untried child-node $y'$ of $y$;
    return $y'$;
Function Select($y$)
    $y_{\text{best}}$=Evaluation($y$);
    return $y_{best}$;
Function DefaultPolicy($y$)
    while $y$ is non-terminal do
        $y$ = randomly selected child of $y$;
    return $reward$ of $y$ (terminal node)
Function BackUp($y$, $reward$)
    while $y$ is not null do
        update all parameters of $y$ needed to compute the evaluation function of $y$ in the
          select step;
    $y$ = parent of $y$;
```

The following paragraphs are based on [3] and [7]. An evaluation function is used in the Selection step in order to identify the best child-nodes. In this thesis we apply two functions, the TA-algorithm [15] and the UCT-algorithm[2] (see Chapter 2). Note that in this case UCT is used for a minimization problem (instead of an maximization problem as described in Subsection 2.0.1). Below, an algorithmic representation of both evaluation-methods and the corresponding Back-up functions are presented. In the following, y denotes a certain node and y' denotes the child-node of y. $N(y)$ is the number of visits to node y and $Q(y')$ is the total pay-off received from node y'. c is a constant and δ is a error probability parameter. $S(y')$ is the number of times node y' leads to one of the s best rewards and k is the number of actions that can be taken next. At the beginning $N(y)$, $Q(y)$ and $S(y)$ are set to zero for every y.

UCT:

```
Function Evaluation($y$)
    foreach child $y'$ of $y$ do
        $reward(y') = \frac{Q(y')}{N(y')} - c\sqrt{\frac{2\log N(y)}{N(y')}}$;
    return child $y'$ with maximum reward;
```

```
Function BackUp($y$, $reward(y)$)
    $reward^* = reward(y)$;
    while $y \neq null$ do
        $N(y) = N(y) + 1$;
        $Q(y) = Q(y) + reward^*$;
        $y$ = parent of $y$;
```

TA:

```
Function Evaluation($y$)
    $\alpha = \log \frac{2N(y)k}{\delta}$;
    foreach child $y'$ of $y$ do
        $h(S(y'), N(y')) = \frac{S(y') + \alpha + \sqrt{2S(y')\alpha + \alpha^2}}{N(y')}$;
    return child $y'$ with maximum $h(S(y'), N(y'))$;
```

```
Function BackUp($y$, $reward(y)$
    $reward^* = reward(y)$;
    if $reward^*$ is under the $s$-best rewards then
        Let $z$ be the node that gave the worst reward among the $s$ best;
        Update the $s$-best list;
    while $y \neq null$ do
        $N(y) = N(y) + 1$;
        $y$ = parent of $y$;
        $S(y) = S(y) + 1$;
    while $z \neq null$ do
        $S(z) = S(z) - 1$;
        $z$ = parent of $z$;
```

4. MCTS for Job Shop Scheduling

Table 4.1.: Notation III - MCTS Scheduling

Abbreviation	**Explanation**
J	set of n jobs
m	number of machines
M_j	set of m machines
O_i	set of operations that correspond to J_i
$o_{i,j}$	operation that belongs to job i and has to be processed on machine j
P	set of processing-times
$p_{i,j}$	processing-time $o_{i,j}$ needs to be processed on machine j
R_i	ordering numbers for operations in J_i
σ	set of routes
W	factor for norming the confidence interval in the UCB formula
z	number of jobs

4.1. Scheduling

The basic setting of a scheduling problem after Brucker [4] is defined as follows. Let $M = \{M_1, \ldots, M_j\}$ be a set of m machines and let $J = \{J_1, \ldots, J_i\}$ be a set of z jobs, where J_i consists of a set of operations $O = \{o_{i,1}, \ldots, o_{i,z}\}$, where operation $o_{i,j}$ has to be performed on machine j. The mapping of all these operations for all jobs to machines is called a schedule, an allocation of operations to machines. The amount of time it takes to process operation $o_{i,j} \in J_i$ on machine M_j is defined as $p_{i,j}$. The processing times of all jobs on all machines can be summarized in a matrix $P = (p_{i,j})_{i,j}$.

Example 1:
Scheduling problems are often written in vector/matrix form:

$$J = \begin{bmatrix} J_1 \\ J_2 \\ J_3 \end{bmatrix} = \begin{bmatrix} o_{1,1} & o_{1,2} & o_{1,3} \\ o_{2,1} & o_{2,2} & o_{2,3} \\ o_{3,1} & o_{3,2} & o_{3,3} \end{bmatrix}; M = \begin{bmatrix} M_1 \\ M_2 \\ M_3 \end{bmatrix}; P = \begin{bmatrix} 10 & 3 & 5 \\ 4 & 13 & 7 \\ 3 & 35 & 12 \end{bmatrix}$$

This problem has size 3×3 ($z \times m$), which means it consists of 3 jobs and 3 machines.

Each job consists of 3 operations. Operation $o_{1,1}$ needs 10 units of time for the processing step on machine 1. Operation $o_{1,2}$ needs 3 units on machine 2, operation $o_{1,3}$ needs 5 units on machine 3 and so on. The assignment of the jobs to machines is called schedule. The amount of time it takes to finish all operations of all jobs is called completion time for the schedule or also make-span.

4.2. Job Shop Scheduling

Job Shop Scheduling is a special subclass of general scheduling problems.[12] The problem-setting encountered in Job Shop Scheduling problems additionally provides an ordering of operations. That is for each job J_i there is a route $R_i = \{\sigma_{i,1}, \ldots, \sigma_{i,z}\}$, which indicates the order in which the operations have to be processed. $\sigma_{i,k}$ is the index l of the k-th operation $o_{i,l}$ in the route of job J_i.[12]

Example 1 - extended

Again the routes of a Job Shop Scheduling problem can be summarized in a matrix:

$$\sigma = \begin{bmatrix} R_1 \\ R_2 \\ R_3 \end{bmatrix} = \begin{bmatrix} \sigma_{1,1} & \sigma_{1,2} & \sigma_{1,3} \\ \sigma_{2,1} & \sigma_{2,2} & \sigma_{2,3} \\ \sigma_{3,1} & \sigma_{3,2} & \sigma_{3,3} \end{bmatrix} = \begin{bmatrix} 2 & 1 & 3 \\ 1 & 3 & 2 \\ 1 & 2 & 3 \end{bmatrix}$$

For job J_i, $\sigma_{1,2}$ is in this case 2, which means that operation $o_{1,2}$ has to be processed on machine 2 first, then according to this example operation $o_{1,1}$ has to be processed on machine 1 and then $o_{1,3}$ on machine 3.

4.3. Implementation of Job Shop Scheduling as a MCTS problem

In order to apply MCTS to Job Shop Scheduling we first have to explain how to fit Job Shop Scheduling into the RL-setting of Section 3.1. For this we consider an online setting, where the allocation of the operations is done sequentially. Then states correspond to partial schedules and we start with the empty schedule as initial state. In each state we can decide, which job to assign to a machine as a next step.

At the beginning, no operation-machine combination has been selected, which means that the first node, from which the algorithm starts, corresponds to an empty schedule. In the next step, an operation-machine combination is selected from a set of next possible combinations. This results in a new state, a node with a schedule that has been extended by the newly selected assignment. The set of terminal states is defined as those states that correspond to a complete schedule. Concerning the rewards (or rather costs), each state gives reward 0, except the terminal states, where the reward corresponds to the make-span of the respective complete schedule. The objective is obviously to minimize the make-span.

In case of using UCT as an evaluation function in MCTS applied to Job Shop Scheduling, two adaptations have to be made to Equation 2.1. First, the sign between the first and second term of the UCB-formula has to be changed, since we want to minimize rewards. Second, the formula has to be scaled, because rewards are not between 0 and 1 anymore. The scaling-factor W and is difficult to define, because processing times are generated randomly and differ from example to example. The first idea for parameter W was to calculate it according to the following formula.

$$W1 = \frac{\sum_{i=1}^{n} \sum_{j=1}^{m} p_{\mathrm{i,j}}}{m} \tag{4.1}$$

To see whether this value is suitable, the differences between make-spans produced during one MCTS rollout and the optimal make-span ($W2$) were calculated. It turned out that the maximum difference was between two and three times higher than $W1$ for each problem size. Considering that very high $W2$ values are mostly outliers, the following values for W for different problem sizes were defined.

Table 4.2.: Normalising factor for different problem sizes

Problem size	**W**
6×6	500
10×10	900
14×14	1600

The additional parameter c in the UCB formula compensates inaccuracies in the estimation of W. When using TA as evaluation function no further adaptations have to be made.

4.4. Branch and Bound

The Branch and Bound method is often used for solving combinatorial optimizations problems which repeats Branch and Bound steps. The basic idea is to divide the search space obtaining sub-problems with smaller search space (branch). For each of these a lower bound is calculated. Furthermore, at the beginning of the algorithm, an upper bound (UB) is calculated by means of a suitable heuristic. If the lower bound of a sub-problem is already larger than the UB, this sub-problem is rejected for further calculations (Bound). The lower UB is at the beginning, the less branching has to be done. The algorithm finishes when a point is reached, at which the sub-problem offers only one feasible solution. Then UB is set to LB (if LB<UB) and LB is set to the current best solution.[4]

In this thesis we used the Branch and Bound algorithm provided by Google using or-tools in order to compute the optimal solutions for the job shop scheduling problems considered.[10]

5. Experiments

5.1. Problem Instances

We tested MCTS using UCT and MCTS using TA on a variation of Job Shop Scheduling problems. The problem instances were created randomly in three different sizes: 6×6, 10×10 and 14×14 and with processing times between 0 and 100. Recall that the size is defined as number of jobs $\times$ number of machines. For each size, five different examples were generated and each example was tested 30 times applying UCT and 30 times applying TA unless described differently. Although the examples were not tested very often, the results presented in Chapter 6 do not show large variance. Below Example 1 in dimension 6×6 is presented.

Example 1 (6×6):

$$\sigma = \begin{bmatrix} 2 & 4 & 6 & 5 & 3 & 1 \\ 6 & 4 & 1 & 5 & 2 & 3 \\ 3 & 5 & 6 & 1 & 2 & 4 \\ 6 & 4 & 5 & 1 & 3 & 2 \\ 4 & 2 & 5 & 3 & 6 & 1 \\ 6 & 1 & 2 & 5 & 3 & 4 \end{bmatrix}; \quad P = \begin{bmatrix} 28 & 54 & 71 & 16 & 47 & 91 \\ 51 & 59 & 12 & 68 & 66 & 80 \\ 17 & 4 & 7 & 68 & 32 & 38 \\ 35 & 79 & 55 & 69 & 39 & 49 \\ 1 & 74 & 16 & 27 & 81 & 58 \\ 33 & 39 & 26 & 78 & 38 & 64 \end{bmatrix}$$

For each problem instance the optimal solution was computed by applying the Branch and Bound algorithm (BaB) provided by google (see Section 4.4).[10] This algorithm solves Job Shop Scheduling problem instances optimally. Up to size 10×10, BaB solves problem instances very quickly. The bigger the size gets (already starting at problem size 14×14), the longer the algorithm takes to calculate a solution. For some examples of problem size 14×14, BaB was not able to find a solution at all in less than two weeks running on a commercial available hardware. In case of Example 1 (6×6) the optimal make-span is 469 units. [4]

5.2. Parameters

UCT-Algorithm:

Applying UCT on MCTS, four different sizes of computational budgets (number of roll-outs) were tested: 100, 1000, 5000 and 10000. For each example and computational budget a variation of eight different c values was tested: 0.001, 0.01, 0.1, 0.5, 1, 2, 5 and 10, unless described differently. The positive real parameter δ was set to 0.01. The scaling-factor W is used as described in Section 4.3.

TA-Algorithm:

Applying TA on MCTS, four different sizes of computational budgets were tested: 100, 1000, 5000 and 10000. For each example and computational budget a variation of five different s values was tested: 30, 50, 70, 100 and 200, unless described differently.

5.3. Key Values

In order to be able to compare the different methods, different key values are used.

Mean percentage error (MPE):

Let k be the number of times the algorithm has been applied (number of trials) onto example j, let y_j be the optimal solution for the problem-instance j and let x_i be the result gained from trial i ($i = 1, \dots, k$). Then we consider the mean percentage error

$$MPE_j = 1 + \frac{1}{k}\sum_{i=1}^{k}\frac{(x_i - y_j)}{y_j} \tag{5.1}$$

If the MPE for a certain problem instance is equal to 1, the optimal solution has always been found. For example, an MPE of 1.5 means that results produced are 1.5 times worse than the optimum.

Average mean percentage error (AMPE):

Let z be the number of examples for a given problem size. Then the average mean percentage error is defined by

$$AMPE = \frac{1}{z}\sum_{j=1}^{z} MPE_j \tag{5.2}$$

Relative Standard Deviation:

The standard deviation Std is a quantity for measuring dispersion. A high standard deviation means that the probability for receiving a value x_i (in our case a make-span) that is much greater or much smaller than the mean value $\bar{x}$ (expected value) is very high. A low standard deviation indicates that the probability for getting values close to the mean is very high. Again k is the number of trials. The standard deviation for the computed make-spans is defined as

$$Std = \sqrt{\frac{\sum_{i=1}^{k}(x_i - \bar{x})^2}{k}} \tag{5.3}$$

The relative standard deviation is then defined as:[8]

$$RStd = \frac{Std}{\bar{x}} \tag{5.4}$$

6. Results

6.1. Results for UCT

In this chapter the results for MCTS using UCT are presented. UCT has been tested for four different numbers of rollouts and the other parameters described in Section 5.3. In the following, parameter n denotes the number of rollouts, W denotes the normed factor explained in Section 4.3, and c is the parameter to calibrate the confidence interval for UCT. The following characteristic values will be compared to each other:

- **Min:** Minimum MPE found for all examples.
- **Mean:** AMPE
- **Max:** Maximum MPE found for all examples.
- **RStd:** Average Relative Standard deviation found for make-spans for all examples.
- **Opt:** Average number of times the optimal make-span was found (in percent).

6.1.1. Results for 6×6

In Table 6.1 the average values computed from the results over the different numbers of rollouts ($n = 100, 1000, 5000, 10000$) and all tested problem instances of size 6×6, using UCT are depicted for each tested constant c respectively. The blue highlighted line marks the results that correspond to the parameter with the smallest mean error. For this problem size parameter 0.1 leads to the smallest mean error.

Table 6.1.: Average characteristic values taken over all tested problem instances and all numbers of rollouts for problem instances of size 6×6 for MCTS using UCT for different c-parameters

Problem	c	Min	Mean	Max	RStd [%]	Opt [%]
6x6 Average	0.001	1.0296	1.0444	1.0638	3.0521	7.4000
	0.01	1.0268	1.0411	1.0614	2.8651	10.6000
	0.1	1.0180	1.0311	1.0453	2.3767	20.6000
	0.5	1.0333	1.0496	1.0748	2.8475	10.4000
	1	1.0371	1.0602	1.0985	2.7483	6.9000
	2	1.0441	1.0675	1.1050	3.0008	3.2000
	5	1.0477	1.0699	1.1113	3.0020	3.2000
	10	1.0527	1.0746	1.1189	3.0592	2.4000

Table 6.2 gives more detailed information, showing additionally to the results for each parameter also the respective results for each number of rollouts. Green highlighted lines show the results that correspond to the parameter, which was identified as the average best one in terms of the characteristic values. In this case the average best parameter for c is 0.1. It offers very low AMPEs. Gray highlighted cells mark single optima outside the average best. Obviously, the higher n, the better the results. Looking at the results for UCT with 5000 and 10000 rollouts it is clear that $c = 0.1$ offers much better results than other c values. In this case the optimal make-span is found about 35 percent of the time and the difference to the optimal solution is on average about 1% . For a number of 1000 rollouts, parameter $c = 0.01$ offers slightly better results than parameter $c = 0.1$.

Table 6.2.: Average results over all tested problem instances of size 6×6 for MCTS using UCT for different number of rollouts and different c-parameters

Problem	**c**	**Min**	**Mean**	**Max**	**RStd [%]**	**Opt [%]**
6x6	0.001	1.0529	1.0731	1.1046	4.1389	2.80
n=100	0.01	1.0525	1.0761	1.1068	3.7864	0.40
W=500	0.1	1.0547	1.0736	1.0986	3.9876	3.20
	0.5	1.0779	1.1091	1.1470	4.5586	0.40
	1	1.0824	1.1243	1.2031	3.9972	0.00
	2	1.0935	1.1299	1.2032	4.4094	0.40
	5	1.1025	1.1365	1.2077	4.1990	0.00
	10	1.1141	1.1478	1.2181	4.4119	0.00
6x6	0.001	1.0269	1.0379	1.0592	2.9803	7.60
n=1000	0.01	1.0198	1.0257	1.0333	2.3824	15.20
W=500	0.1	1.0121	1.0308	1.0480	2.4806	11.60
	0.5	1.0358	1.0503	1.0927	3.2813	4.80
	1	1.0435	1.0622	1.1065	2.8844	1.60
	2	1.0474	1.0689	1.1029	3.3031	1.20
	5	1.0456	1.0668	1.1160	3.0953	1.60
	10	1.0517	1.0707	1.1190	3.1768	0.40
6x6	0.001	1.0153	1.0341	1.0461	2.5449	9.60
n=5000	0.01	1.0161	1.0322	1.0515	2.7635	12.40
W=500	0.1	1.0050	1.0108	1.0193	1.7802	35.20
	0.5	1.0129	1.0245	1.0359	1.9947	12.80
	1	1.0158	1.0328	1.0536	2.1140	7.20
	2	1.0223	1.0418	1.0731	2.4006	4.40
	5	1.0251	1.0423	1.0660	2.3792	4.00
	10	1.0287	1.0460	1.0778	2.4281	3.20
6x6	0.001	1.0234	1.0324	1.0454	2.5444	9.60
n=10000	0.01	1.0188	1.0305	1.0540	2.5279	14.40
W=500	0.1	1.0050	1.0091	1.0151	1.2585	32.40
	0.5	1.0064	1.0144	1.0237	1.5553	23.60
	1	1.0066	1.0214	1.0306	1.9975	18.80
	2	1.0133	1.0292	1.0409	1.8903	6.80
	5	1.0177	1.0341	1.0553	2.3344	7.20
	10	1.0165	1.0340	1.0607	2.2201	6.00

Figure 6.1 presents the comparison of calculation time with the mean error for UCT (with $c = 0.1$) and Branch and Bound. The error bars show the average maximum MPE (upper bound) the average minimum MPE (lower bound) and the average AMPE. Each error bar corresponds to a different number of rollouts (left: $n = 100$, middle: $n = 1000$, right: $n = 10000$). The solution of the Branch and Bound algorithm is marked with a red cross. As can be seen, for $n = 100$ UCT needs on average slightly more time for calculating one problem instance than Branch and Bound. For this problem size UCT cannot compete with BaB in terms of performance.

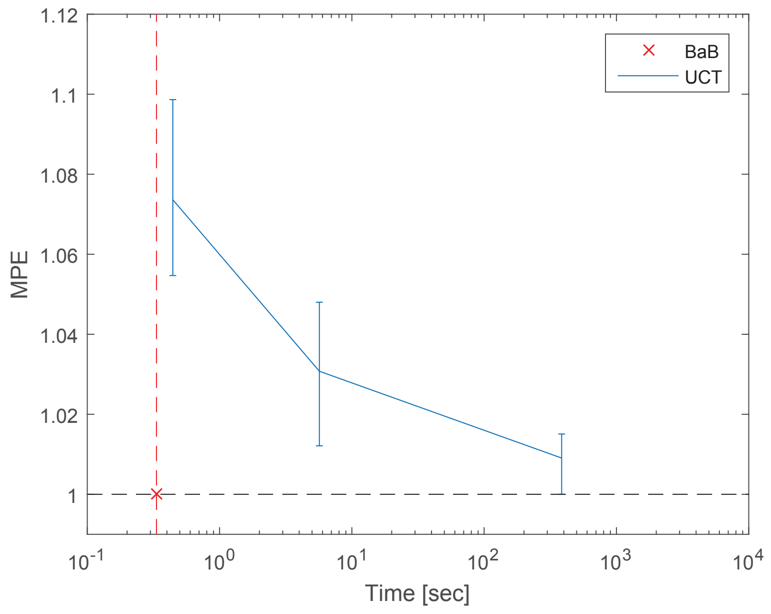

Figure 6.1.: Branch and Bound compared to MCTS using UCT for problem size 6×6 for different rollouts and parameter $c = 0.1$

6.1.2. Results for 10×10

In Table 6.3 the results for size 10×10 are presented. Again, parameter $c = 0.1$ yields the smallest mean error and the best results on average.

Table 6.3.: Average characteristic values taken over all tested problem instances and all numbers of rollouts for problem instances of size 10×10 for MCTS using UCT for different c-parameters

Problem	**c**	**Min**	**Mean**	**Max**	**RStd** [%]	**Opt** [%]
10×10	0.001	1.1445	1.1565	1.1727	3.0501	0
	0.01	1.1328	1.1543	1.1774	2.9635	0
	0.1	1.1276	1.1454	1.1687	2.9264	0
	0.5	1.2454	1.2716	1.2986	3.5831	0
	1	1.2870	1.3060	1.3327	3.4921	0
	2	1.2902	1.3158	1.3475	3.1295	0
	5	1.3003	1.3274	1.3516	2.9926	0
	10	1.2728	1.2987	1.3285	3.1751	0

Table 6.4 presents the average results for problem instances of size 10×10 over all examples. For $n = 5000$ each example was calculated 15 times. For $n = 100$ parameter $c = 0.01$ offers slightly better results than $c = 0.1$. For every other tested n clearly $c = 0.1$ offers the best results. For $c > 0.1$ characteristic values are a lot worse, the mean error being about 20% higher.

Table 6.4.: Average results over all tested problem instances of size 10 × 10 for MCTS using UCT for different number of rollouts and different *c*-parameters

Problem	c	Min	Mean	Max	RStd [%]	Opt [%]
10×10	0.001	1.2311	1.2475	1.2644	5.0390	0
n=100	0.01	1.2238	1.2472	1.2691	4.2917	0
W=900	0.1	1.2280	1.2634	1.3074	4.1214	0
	0.5	1.3341	1.3656	1.4057	3.6652	0
	1	1.3699	1.3868	1.4076	3.5765	0
	2	1.3528	1.3865	1.4297	3.4480	0
	5	1.3567	1.3968	1.4226	4.0365	0
	10	1.2619	1.2796	1.3144	4.5255	0
10×10	0.001	1.1657	1.1816	1.1969	3.2740	0
n=1000	0.01	1.1522	1.1733	1.1931	3.4805	0
W=900	0.1	1.1380	1.1565	1.1728	3.4211	0
	0.5	1.2655	1.2949	1.3214	3.6547	0
	1	1.3094	1.3219	1.3477	3.5622	0
	2	1.3041	1.3233	1.3451	3.4296	0
	5	1.3049	1.3357	1.3706	3.0172	0
	10	1.3000	1.3299	1.3594	3.0910	0
10×10	0.001	1.0930	1.1003	1.1152	2.0666	0
n=5000	0.01	1.0854	1.1125	1.1387	2.4182	0
W=900	0.1	1.0791	1.0886	1.1035	2.0997	0
	0.5	1.2029	1.2326	1.2531	3.4022	0
	1	1.2420	1.2714	1.3162	3.7028	0
	2	1.2694	1.2887	1.3149	2.7757	0
	5	1.2713	1.2965	1.3176	2.2128	0
	10	1.2604	1.2984	1.3285	2.5802	0
10x10	0.001	1.0882	1.0967	1.1142	1.8208	0
n=10000	0.01	1.0700	1.0840	1.1086	1.6636	0
W=900	0.1	1.0653	1.0729	1.0910	2.0632	0
	0.5	1.1790	1.1935	1.2141	3.6101	0
	1	1.2267	1.2440	1.2595	3.1270	0
	2	1.2344	1.2647	1.3003	2.8646	0
	5	1.2682	1.2804	1.2957	2.7037	0
	10	1.2691	1.2871	1.3117	2.5037	0

In Figure 6.2 the mean error and the computation time for UCT (for $c = 0.01$) and Branch and Bound are presented. Interestingly, for this problem size Branch and Bound works even slighty faster than UCT for $n = 100$. This indicates that Branch and Bound is quite sensitive to the specific problem instance. The average mean errors are in this case higher than for problem size 6×6.

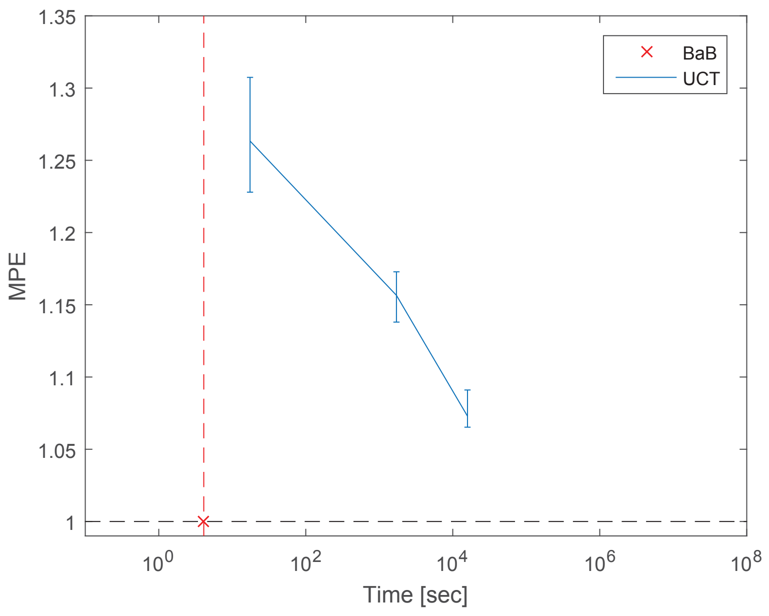

Figure 6.2.: Branch and Bound compared to MCTS using UCT for problem size 10×10 for different rollouts and parameter $c = 0.1$

6.1.3. Results for 14×14

For size 14×14 five different problem instances were calculated five times respectively. Parameters $c = 5$ and $c = 10$ have not been considered in this case, due to bad results in previous calculations. In Table 6.5 the average results for UCT for problem size 14×14 are presented. The parameter that offers the lowest mean error is in this case $c = 0.001$. Due to the fact that parameter W is an empiric value, c has to compensate possible inaccuracies.

Table 6.6 presents the results for UCT for six different parameters. Results found for parameter $c = 0.001$ and for $n = 10000$ are 1.2 times worse than the optimal solution.

Table 6.5.: Average results over all tested problem instances of size 14×14 for MCTS using UCT for differnt number of rollouts and different c-parameters

Problem	**c**	**Min**	**Mean**	**Max**	**RStd [%]**	**Opt [%]**
14x14	0.001	1.2470	1.2570	1.2668	3.0792	0.0000
	0.01	1.3041	1.3192	1.3337	3.8253	0.0000
	0.1	1.4359	1.4568	1.4776	3.2753	0.0000
	0.5	1.4866	1.5053	1.5268	2.7985	0.0000
	1	1.5040	1.5267	1.5519	2.7287	0.0000
	2	1.5023	1.5295	1.5616	2.7185	0.0000

Table 6.6.: Average results over all tested problem instances of size 14×14 for MCTS using UCT for different number of rollouts and different c-parameters

Problem	**c**	**Min**	**Mean**	**Max**	**RStd [%]**	**Opt [%]**
14x14	0.001	1.3667	1.3853	1.4065	3.5022	0
n=100	0.01	1.3836	1.3981	1.4100	3.8228	0
W=1600	0.1	1.3977	1.4150	1.4427	4.1557	0
	0.5	1.5100	1.5503	1.5814	3.0066	0
	1	1.5747	1.6018	1.6338	3.1478	0
	2	1.5679	1.6128	1.6725	3.1073	0
14x14	0.001	1.2226	1.2302	1.2351	2.6953	0
n=1000	0.01	1.3856	1.4091	1.4356	5.3771	0
W=1600	0.1	1.5136	1.5401	1.5579	2.8766	0
	0.5	1.5326	1.5433	1.5598	2.1256	0
	1	1.5031	1.5329	1.5493	2.3505	0
	2	1.5031	1.5329	1.5493	2.3505	0
14x14	0.001	1.2018	1.2018	1.2018	2.5167	0
n=5000	0.01	1.2658	1.2658	1.2658	3.5603	0
W=1600	0.1	1.4448	1.4448	1.4448	2.9992	0
	0.5	1.4531	1.4531	1.4531	3.3688	0
	1	1.4821	1.4821	1.4821	2.8511	0
	2	1.4821	1.4821	1.4821	2.8511	0
14x14	0.001	1.1967	1.2109	1.2237	3.6025	0
n=10000	0.01	1.1814	1.2040	1.2233	2.5411	0
W=1600	0.1	1.3875	1.4273	1.4649	3.0698	0
	0.5	1.4507	1.4746	1.5127	2.6929	0
	1	1.4561	1.4902	1.5426	2.5652	0
	2	1	1.4902	1.5426	2.5652	0

In Figure 6.3 results for UCT (for $c = 0.001$) are presented. In this case Branch and Bound needs almost the same amount of computation time as UCT for $n = 10000$. Note that some of the randomly generated problem instances of size 14×14 could not be solved using Branch and Bound within two weeks. Hence, those examples have been exchanged with new ones, Branch and Bound was able to solve, in an acceptable amount of time. Considering these information, the advantage of UCT is, that it computes solutions for problem instances of size 14×14 independent of how complex they are in almost always the same amount of time.

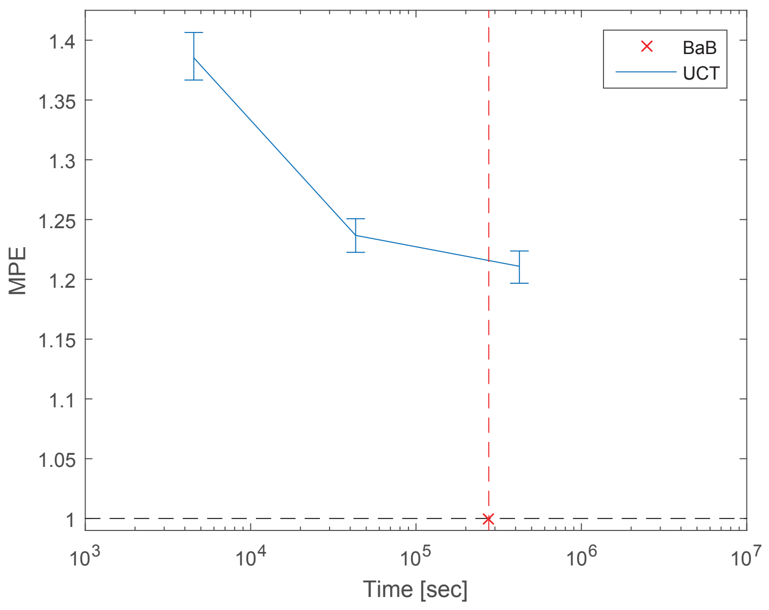

Figure 6.3.: Branch and Bound compared to MCTS using UCT for problem size 14×14 for different rollouts and parameter $c = 0.001$

6.2. Results for TA

6.2.1. Results for 6×6

In Table 6.7 the average values for each parameter s are presented. $s = 70$ is identified as the average best parameter giving the smallest mean error. Nevertheless, looking at the number of times the optimum has been found (Opt) for $s = 100$ and $s = 200$, those parameter values evidently yield slightly better results in some cases.

Table 6.7.: Average characteristic values taken over all tested problem instances and all numbers of rollouts for problem instances of size 6×6 for MCTS using TA for different s-parameters

Problem	s	**Min**	**Mean**	**Max**	**RStd** [%]	**Opt** [%]
6x6	30	1.0460	1.0640	1.0874	3.0743	1.6000
	50	1.0435	1.0636	1.0994	2.8616	1.6000
	70	1.0321	1.0586	1.0948	2.8931	2.4000
	100	1.0491	1.0743	1.1111	2.8208	2.5000
	200	1.0527	1.0775	1.1202	2.5712	2.7000

The results for MCTS using TA are depicted in Table 6.8. Again n denotes the number of rollouts. The parameter s to calibrate as well as the characteristic values are explained in Section 5.3.

Table 6.8.: Average results over all tested problem instances of size 6×6 for MCTS using TA for different number of rollouts and different s-parameters

Problem	**s**	**Min**	**Mean**	**Max**	**RStd** [%]	**Opt** [%]
6x6 n=100	30	1.0949	1.1183	1.1613	4.4399	0.40
	50	1.0892	1.1149	1.1864	4.1876	0
	70	1.0570	1.1085	1.1929	4.1126	0.40
	100	1.1130	1.1785	1.2739	4.1545	0
	200	1.1408	1.1940	1.2868	3.0747	0
6x6 n=1000	30	1.0495	1.0679	1.0836	3.3176	0
	50	1.0362	1.0677	1.1084	2.9251	0.80
	70	1.0367	1.0637	1.0830	2.9647	0.40
	100	1.0461	1.0598	1.0852	2.9741	0.80
	200	1.0407	1.0584	1.0957	2.8880	0.80
6x6 n=5000	30	1.0228	1.0381	1.0624	2.3676	3.20
	50	1.0298	1.0444	1.0607	2.3157	1.60
	70	1.0171	1.0355	1.0635	2.4076	2.80
	100	1.0223	1.0358	1.0531	2.5061	5.60
	200	1.0197	1.0317	1.0525	2.2993	3.60
6x6 n=10000	30	1.0170	1.0316	1.0424	2.1721	2.80
	50	1.0187	1.0272	1.0421	2.0180	4.00
	70	1.0177	1.0267	1.0398	2.0875	6.00
	100	1.0149	1.0233	1.0320	1.6483	3.60
	200	1.0094	1.0258	1.0460	2.0227	6.40

Although $s = 70$ offers the best results on average, the higher the number of rollouts get, the worse the results for this parameter compared to higher s-values become. For $n = 100$ parameter $s = 70$ works very good, giving small mean error. For $n = 1000$ parameter $s = 100$ and $s = 200$ offer better results. Moreover, in this case the optimum was found more often. The same trend is identified for $n = 5000$ and $n = 10000$.

Figure 6.4 again compares the results obtained with TA (for $s = 70$ identified as the best) to Branch and Bound. The error bars for TA again are three different numbers of rollouts (left: $n = 100$, middle: $n = 1000$, right: $n = 10000$). For $n = 100$, TA needs less computation time than Branch and Bound.

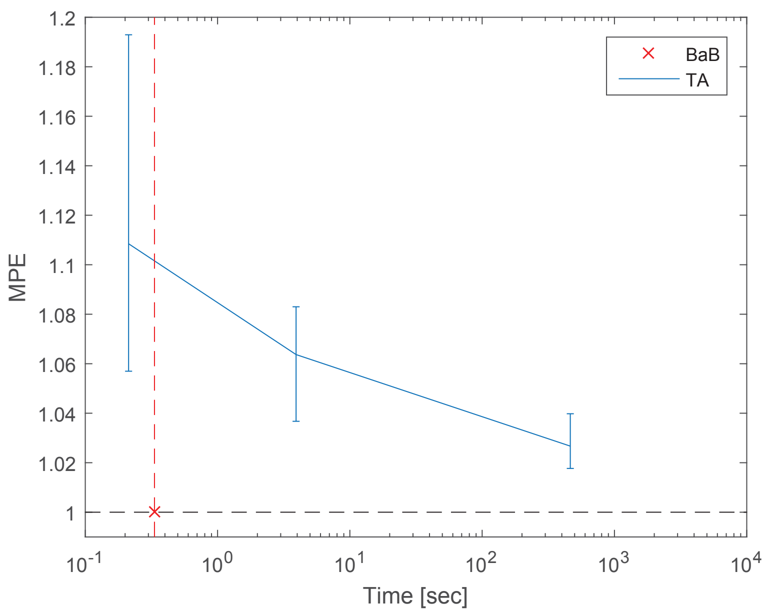

Figure 6.4.: Branch and Bound compared to MCTS using TA for problem size 6 × 6 for different rollouts and parameter $s = 70$

6.2.2. Results for 10×10

Table 6.9 presents the average characteristic values for each parameter s for size 10×10. For MCTS using TA and problem size 10×10 parameter $s = 70$ promises the best average results.

Table 6.9.: Average characteristic values taken over all tested problem instances and all numbers of rollouts for problem instances of size 10×10 for MCTS using TA for different s-parameters

Problem	s	Min	Mean	Max	RStd [%]	Opt [%]
6x6	30	1.2511	1.3246	1.3746	3.0553	0.000
	50	1.2538	1.3187	1.3613	3.0737	0.000
	70	1.2327	1.3087	1.3595	2.9713	0.000
	100	1.2699	1.3283	1.3718	3.0971	0.000
	200	1.2416	1.3217	1.3695	3.2419	0.000

The results for problem size 10×10 are depicted on Table 6.10. For $n = 100$ and $n = 1000$ five different examples were calculated for each parameter 30 times. For $n = 5000$ and $n = 10000$ the five problem instances were calculated for each parameter 15 times.

Table 6.10.: Average results over all tested problem instances of size 10×10 for MCTS using TA for different number of rollouts and different s-parameters

Problem	s	**Min**	**Mean**	**Max**	**RStd** [%]	**Opt** [%]
10x10	30	1.369	1.408	1.440	3.814	0
n=100	50	1.356	1.397	1.424	3.922	0
	70	1.324	1.369	1.414	3.313	0
	100	1.431	1.464	1.492	3.391	0
	200	1.409	1.470	1.498	3.138	0
10x10	30	1.115	1.307	1.411	3.087	0
n=1000	50	1.128	1.301	1.371	3.242	0
	70	1.101	1.299	1.407	3.033	0
	100	1.124	1.291	1.377	3.687	0
	200	1.088	1.284	1.382	3.621	0
10x10	30	1.263	1.299	1.333	2.645	0
n=5000	50	1.273	1.300	1.335	2.554	0
	70	1.265	1.290	1.320	3.165	0
	100	1.269	1.286	1.319	2.767	0
	200	1.251	1.277	1.316	3.054	0
10x10	30	1.257	1.284	1.315	2.675	0
n=10000	50	1.258	1.277	1.314	2.577	0
	70	1.241	1.276	1.297	2.374	0
	100	1.256	1.272	1.299	2.544	0
	200	1.219	1.256	1.282	3.154	0

For $n = 100$ it is true that $s = 70$ is the average best parameter. Looking at the results for $n = 1000$, $n = 5000$, and $n = 10000$ parameter $s = 200$ seems to be better, because it leads to the smallest mean error. For this problem size TA was not able to find the optimal solution once.

In Figure 6.5 the results for TA for $s = 70$ are presented. In this case Branch and Bound works even faster than UCT for $n = 100$ (left bar). The maximum MPEs found are much higher than for problem size 6×6. The second bar in this diagram is particularly striking, because the difference between the maximum and minimum MPE found is very high. This is due to outliers that result from the random simulation.

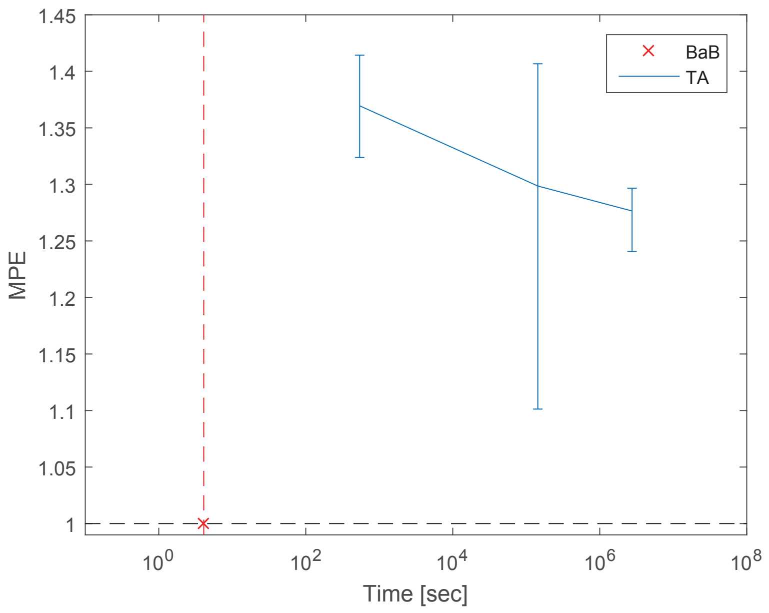

Figure 6.5.: Branch and Bound compared to MCTS using TA for problem size 10×10 for different rollouts and parameter $s = 70$

6.2.3. Results for 14×14

In Table 6.11 the average characteristic values for each parameter are presented. For $n = 100$ and $n = 1000$ each of the five problem instances was calculated 15 times for each parameter s; for $n = 5000$ and $n = 10000$ each of the five problem instances was calculated 5 times for each parameter, due to time reasons. For this problem size, parameter $s = 30$, is identified as the average best parameter, giving the smallest mean error. Note that in this case the mean errors for different s are very close to each other.

Table 6.11.: Average characteristic values taken over all tested problem instances and all numbers of rollouts for problem instances of size 14×14 for MCTS using TA for different s-parameters

Problem	s	Min	Mean	Max	RStd [%]	Opt [%]
6x6	30	1.490	1.534	1.574	2.416	0.000
	50	1.500	1.540	1.583	2.938	0.000
	70	1.518	1.544	1.575	2.556	0.000
	100	1.529	1.564	1.607	2.213	0.000
	200	1.532	1.559	1.587	2.557	0.000

Table 6.12 presents the characteristic values for TA for different s-values and for different numbers of rollouts. For this problem size the optimum was not found once. For $n = 10000$ the mean error is approximately 1.5. This means that generated solutions were about 1.5 times worse than the optimum solution.

Table 6.12.: Average results over all tested problem instances of size 14×14 for MCTS using TA for different number of rollouts and different s-parameters

Problem	s	Min	Mean	Max	RStd [%]	Opt [%]
14x14	30	1.5240	1.5835	1.6274	3.3054	0
n=100	50	1.5774	1.5957	1.6268	3.0220	0
	70	1.5783	1.6238	1.6588	2.3989	0
	100	1.6559	1.6938	1.7522	2.0124	0
	200	1.6591	1.7061	1.7456	2.2680	0
14x14	30	1.5199	1.5445	1.5919	2.8250	0
n=1000	50	1.5101	1.5501	1.6069	3.4221	0
	70	1.5217	1.5443	1.5661	2.4640	0
	100	1.5267	1.5490	1.5714	2.9439	0
	200	1.5062	1.5346	1.5830	3.3091	0
14x14	30	1.4764	1.5081	1.5266	1.9420	0
n=5000	50	1.4440	1.5137	1.5715	2.1685	0
	70	1.4944	1.5111	1.5397	2.2044	0
	100	1.4712	1.5086	1.5558	1.8983	0
	200	1.4800	1.5042	1.5234	2.1255	0
14x14	30	1.4394	1.4984	1.5500	1.5925	0
n=10000	50	1.4692	1.4987	1.5273	3.1410	0
	70	1.4782	1.4981	1.5340	3.1585	0
	100	1.4626	1.5055	1.5502	1.9986	0
	200	1.4814	1.4895	1.4959	2.5241	0

Figure 6.6 shows the comparison of the mean error and the computing time for TA (for $s = 30$) and Branch and Bound. In this case Branch and Bound needs almost as long as TA for $n = 10000$. The mean error for every number of rollouts is very high and compared to results for UCT (for the same problem instances), TA works significantly worse concerning time and quality. Although the number of performed tests for each problem instance is rather small, one can note that different values for parameter s influence the quality of the results marginally.

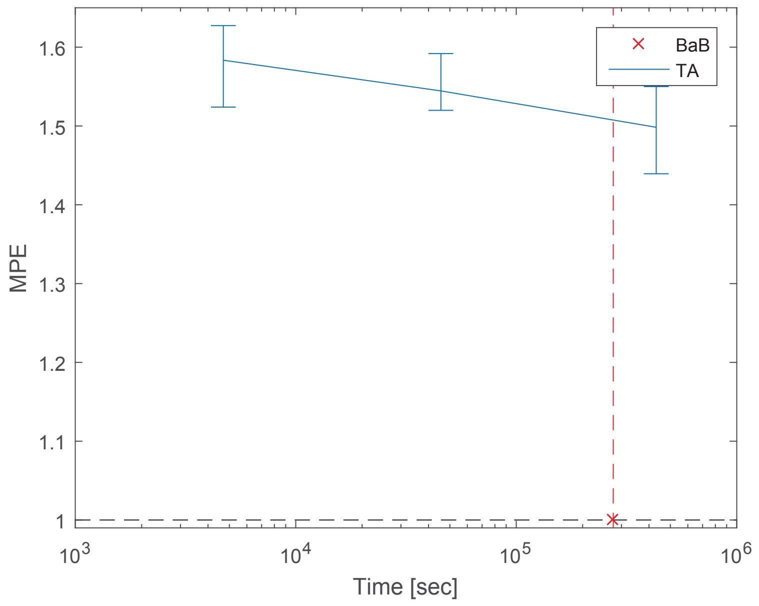

Figure 6.6.: Branch and Bound compared to MCTS using TA for problem size 14×14 for different rollouts and parameter $s = 30$

6.3. Other Rollout algorithms

The Rollout algorithm, also Pilot Method, aims at efficiently solving combinatorial optimization problems using heuristics. They are special forms of MCTS differing in the heuristics used for the rollouts (or simulations). After performing the rollouts, the best action (in case of minimization, the action that leads to the smallest reward) is selected.[13]

Average rollout algorithm:
The average rollout algorithm is a special form of the rollout algorithm and it differs concerning the selection of the next best action. The average rollout algorithm does not select the action that leads to the smallest reward (in case of minimization), but the average best one. To ensure that the final solution is a very good one, the final solution is the best solution found during the rollouts.[9]

Below some heuristics, that can be used for rollout algorithms are presented:

- Random heuristic: This heuristic randomly chooses next actions during the simulation step.[9]
- Randomly Chosen Dispatch Rules: This heuristic randomly chooses one dispatch rule out of a certain set of dispatch rules for each time a new action has to be taken during the simulation step. Possible dispatch rules can for example be FIFO (first in first out) or LIFO (last in last out).[9]

- Threshold Ascent: Another heuristic used for average rollout algorithms is Threshold Ascent (see 2.0.2).
- Most work remaining heuristic: The job with the longest processing time is selected next.[13]
- Shortest processing time heuristic: The job with the shortest processing time is selected next.[13]

6.4. Comparison with other research

Results presented in the master thesis of Einar Geirsson [9] are compared to the two algorithms used in this master thesis. DH_{ave} denotes an average rollout algorithm using random dispatch rules. RH_{ave} denotes an average rollout algorithm using a random heuristic and RH_{TA} denotes an average rollout algorithm using a random heuristic and Threshold Ascent (see Section 6.3).

Table 6.13.: Average rollout algorithms using different heuristics compared to MCTS using UCT and TA for different problem sizes and $n = 10000$

Problem	**Method**	**Min**	**Mean**	**Max**	**RStd**	**Opt**
6x6	DH_{ave}	1.00	1.03	1.18	0.96	22.63
	RH_{ave}	1.00	1.05	1.16	1.30	14.77
	RH^{TA}_{ave}	1.01	1.07	1.17	3.84	8.10
	UCT	1.00	1.01	1.02	1.26	32.40
	TA	1.01	1.02	1.04	1.91	13.20
10x10	DH_{ave}	1.01	1.06	1.13	1.87	0.28
	RH_{ave}	1.07	1.03	1.12	2.21	0.08
	RH^{TA}_{ave}	1.05	1.10	1.15	3.19	0.05
	UCT	1.07	1.07	1.09	2.06	0.00
	TA	1.22	1.26	1.28	3.15	0.00
14x14	DH_{ave}	1.04	1.09	1.14	1.91	0.28
	RH_{ave}	1.04	1.09	1.13	1.95	0.00
	RH^{TA}_{ave}	1.06	1.13	1.17	2.67	0.18
	UCT	1.20	1.21	1.22	3,60	0.00
	TA	1.44	1.50	1.55	1.60	0.00

The UCT-algorithm yields the best results for problem size 6×6 compared to the other algorithms presented in Table 6.13. Its mean is close to one, which means that the algorithm finds the optimal or an almost optimal solution on average very often. More precisely, the optimum was found in about 33%. Although TA has the second lowest mean error, the optimum was only found in 13%.

For problem size 10×10, TA seems to work much worse than any other algorithm. UCT again performs quite well. It yields the second best mean error. UCT and TA were not able to find the optimum once.

For problem size 14×14, UCT and TA seem to work worse than any of the rollout algorithms. Again, UCT offers better characteristic values than TA.

In the paper of Runarsson et al. [13] results for the pilot method using different heuristics are presented. In the Table 6.14 those results are compared to results of this thesis. Denote MWKR as the most work remaining heuristic and SPT as the shortest processing time heuristic (see Section6.3).

For problem size 6×6, UCT seems to work better than the other presented rollout algorithms: the optimum was found in 35% of times and the mean error is very small. The same trend is identified for the other problem sizes (10×10 and 14×14) as well. For problem size 14×14 the rollout algorithm using MWKR as a heuristic offers slightly better results. MCTS using TA yields the worst characteristic values for problem size 10×10 as well as for size 14×14.

Table 6.14.: Rollout algorithms using MWKR and SPT compared to MCTS using UCT and TA for different problem sizes and $n = 5000$

Size	**Heuristic**	**Min**	**Mean**	**Max**	**RStd**	**Opt**
6x6	MWKR	1.000	1.025	1.104	2.9	33
	SPT	1.000	1.052	1.265	4.5	14
	UCT	1.005	1.011	1.019	1.780	35.20
	TA	1.020	1.032	1.053	2.299	3.60
10x10	MWKR	1.004	1.082	1.158	3.5	0
	SPT	1.063	1.172	1.296	4.8	0
	UCT	1.079	1.089	1.104	2.100	0
	TA	1.251	1.277	1.316	3.054	0
14x14	MWKR	1.046	1.129	1.230	3.4	0
	SPT	1.153	1.286	1.517	6.0	0
	UCT	1.202	1.202	1.202	2.517	0
	TA	1.476	1.508	1.527	1.942	0

6.5. Summary

The performance of the algorithms tested in this thesis yield quite satisfying results compared to other rollout algorithms from the literature. Especially, the combination of MCTS using UCT yields promising results. For the 6×6 problems UCT finds the optimal solution more often than the other presented algorithms. For the problem sizes 10×10 and 14×14 UCT generates quite good solutions considering the comparatively low mean errors.

Results generated by TA are generally worse than those of UCT. They show much higher mean errors and find the optimal solutions less often. In addition, TA requires more computation time than UCT. The behaviour of the results for TA do not vary significantly for different s-values at a constant number of rollouts. It is also noteworthy that for UCT with a parameter $c < 1$ (see Formula 2.1) better results have been found throughout the tested examples. Summing up, the absolute value of the parameter s for TA has little influence on the results, whereas the tuning of value c for UCT leads to better characteristic values.

Finally, it has to be pointed out that above a certain size (already observed at 14×14 for a significant fraction of tested configurations), Branch and Bound fails to find a solution within a reasonable amount of time, whereas MCTS is always able to calculate acceptable solutions. Although it is clear that the results obtained for larger problems merely depict decent approximations of the optimum, it has to be kept in mind that the number of rollouts can still improve the results, although at the cost of higher calculation time.

7. Relevance for Logistics

7.1. Quality

Quality is the match between the characteristics of an object with the object-requirements someone has. In other words, it describes the degree of requirements- and expectation-fulfilments. [1]

More detailed, an object may be a product, a process, a system, or raw data. The person defining the requirements is normally either a customer or a producer or a seller. Thus best quality products and services are dependent on the performance requirements, on the performance itself and the interaction of the people involved.[1]

If one of the requirements is not fulfilled, the product is regarded to be qualitatively low-order and it is not possible to compensate this by over-achieving another requirement. As a matter of fact the customer will be dissatisfied with the output. Bad quality does not only affect customer-satisfaction as well as the acquisition of new customers, but it also impacts the performance of internal and external processes and the corporate identity as well as it affects costs and time (processing times, delivery times etc.).[11]

7.2. Data-Quality

Data quality, also called information-quality, is defined as the suitability of data for its intended use. Thus data quality can be subdivided into intrinsic, contextual, representational and accessible data quality. [18]

Table 7.1.: A Conceptual Framework of Data Quality[18]

Type	Characteristics
Intrinsic data quality	Credibility Accuracy Objectivity Reputation
Contextual data quality	Value-added Relevancy Timeliness Completeness Appropriate account of data
Representational data quality	Interpretability Ease of understanding Representational consistency Concise Representation
Accessibility data quality	Accessibility Access security

Table 7.1 lists aspects that people associate with high quality data. Intrinsic data quality refers to direct properties of data. In contrast, contextual data quality highlights the characteristics that support its intended use. Representational and accessible data quality focuses on the interpretability and accessibility of data for a system.[18]

7.3. Data quality in terms of scheduling problems

Scheduling problems are very common problems in industry. The higher the number of resources and objects, that have to be matched to each other, the more complex the scheduling process becomes. For large problems, it is not even possible to find the optimal solution in acceptable time. At this point different scheduling algorithms are used to compute very good approximate solutions.

Examples for scheduling problems in industry are the allocation of jobs to machines, train/bus/plane schedules, or personnel planning. Each of these problems has a certain objective. Depending on the specific objective, potentials in saving money or time, in reducing set-up operations or in decreasing the lead time can be found.

Looking at scheduling problems, the quality of input data is as essential, as the definition of an objective. In this master thesis the objective is the minimization of the make-span, which means keeping the overall processing time at a minimum. Besides a precise formulated objective the algorithm needs the following input data: the number of jobs, the number of machines, the corresponding processing times and in the case of Job Shop

Scheduling problems corresponding partial routes (see Chapter 4). The parameter "number of machines" is mostly dictated by the nature of the production lines. The number of jobs is already more difficult to determine, because depending on the company, new jobs can enter the system at any time. The route a job follows is dictated by the type of the job. The input parameters with the highest error-proneness are the processing times or in general capacities, which correspond to the objects that have to be allocated to some resources. It is crucial that the quality of the processing times is as good as possible, because it has a great impact on the quality of the output data. In Figure 7.1 two schedules are depicted on machine-oriented Gantt-charts. The underlying problem consists of a set of 3 machines ($M1$,$M2$, $M3$) and a set of 3 jobs ($J1$, $J2$, $J3$). In this figure the processing time of each job is illustrated directly next to the respective job (for example job 1 needs 3 units on machine 3).

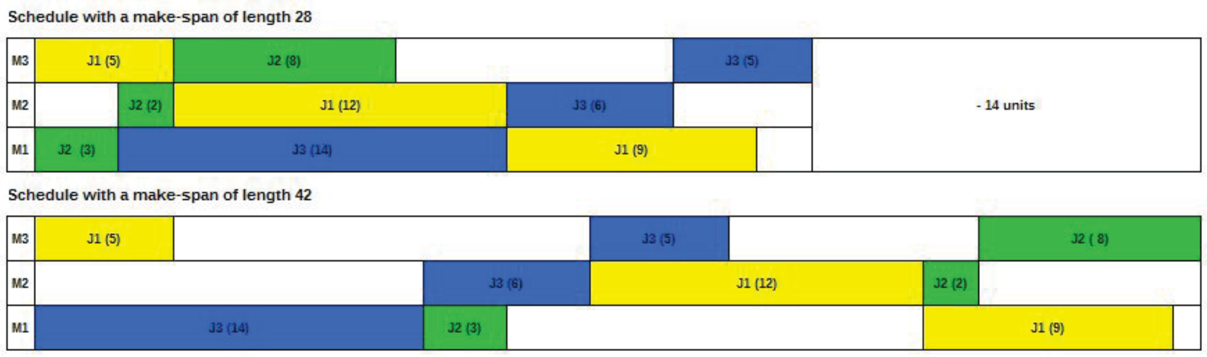

Figure 7.1.: Representation of two schedules of different lengths for the same example

The upper schedule (see Figure 7.1) leads to a make-span of 28 units. There are very few dwell-times and a certain job is mostly processed directly without any idle time. The schedule with a make-span of 42 units, needs 14 lengths more than the other one. The quality of this schedule is worse due to very long dwell times. The problem is that the machines are not used for such high capacities and moreover the lead-time for job 1 and job 2 is unnecessarily high.

To sum it up, a schedule is of high quality, if the make-span is short and if there are no or only a few dwell-times. Moreover it is preferable to have short lead-times of jobs and that machines are used to capacity. In order to achieve high quality output data a good algorithm as well as high-quality input data are necessary. In this thesis Monte Carlo Tree Search is used to solve scheduling problems. For smaller problems, the algorithm finds the optimal solution in most cases. The larger the problems become, the less often the optimal solution is found. Nevertheless, the solutions the algorithm finds are good and in any case better than manually found solutions.

The quality of the input data is strongly dependent on precise data acquisition. Particularly inaccuracies in the collection of the process times can have fatal effects on the accuracy of the output data. The greater the deviations of the fixed process times in comparison to the real times are, the less the make-span calculated by an algorithm will coincide with reality. Inaccuracies in the processing times can find their roots in different causes. If automated detection is not used, differences could be caused by the estimation of processing times or by the use of experience values. In case of estimating processing times, it is important to preserve objectivity in order to not overstate or underestimate certain processing times. In the case of automated time recording, errors could be caused by misinterpretations, due to different data structures and by incomplete data (for example due to system failures).

Consequences of poor data quality can be various. Longer overall processing times lead to higher costs. This includes costs for overtime, energy costs and any resulting payments for delays in delivery. Furthermore, the overall utilization of the machines is lower and less output is produced in a certain time interval. Poorly estimated or incorrect processing times can cause unnecessary dwell- and lead-times as well as poor utilization. Personnel- or shift-plans based on machine scheduling plans of low quality could be over- or under-sized and thus affect the entire company's processes. Oversized machine-schedules lead to an oversized personnel plan at the appropriate workplaces. Furthermore, fewer orders than possible are accepted, because of the consideration of a wrong lead time. Under-sized schedules lead to delays within and outside the company. More orders than produced with existing capacities, were accepted in advance.Thus additional costs are the consequence due to backorders and bad reputation.

For the calculations in this thesis no practical data was used. The input data was created independently and the corresponding processing times were generated randomly between 0 and 100. The data was stored in text-files and the main difficulty was to obtain the wanted data from the files and use them in the programming-environment correctly. The data-import process is very important and it has to be tested sufficiently in order to guarantee the completeness and correctness of the input data. The structure of the input data corresponds to the structure described in Section 4.1. Therefore the input data fully meets the previous suggested quality requirements. This forms the basis for an objective assessment of the tested algorithms. Besides high quality input-data the representation of the output-data is very important. The latter was stored in text-files, including the example-number, the applied algorithm, the complete schedule, the corresponding make-span, the computation time and the parameters used for calculation. Based on these records, the algorithms were evaluated in terms of their performance (see Section 6).

References

Books

[1] Katrin Alisch, Eggert Winter, and Ute Arentzen. *Gabler Wirtschafts Lexikon.* Springer-Verlag, 2013

[2] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. “Finite-time Analysis of the Multiarmed Bandit Problem”. In: *Machine Learning* 47.2 (05/2002), pp. 235–256. ISSN: 1573-0565. DOI: 10.1023/A:1013689704352

[3] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. “A Survey of Monte Carlo Tree Search Methods”. In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.1 (03/2012), pp. 1–43. ISSN: 1943-068X. DOI: 10.1109/TCIAIG.2012.2186810

[4] Peter Brucker. *Scheduling algorithms.* Vol. 5. Springer, 2007

[5] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. “Monte-Carlo Tree Search: A New Framework for Game AI.” In: (2008)

[6] Thomas H Cormen and Karin Lippert. *Algorithmen-Eine Einführung.* Vol. 2. Oldenbourg, 2010

[7] F. De Mesmay, A. Rimmel, Y. Voronenko, and M. Püschel. “Bandit-based optimization on graphs with application to library performance tuning”. In: (2009), pp. 729–736

[8] Brian Everitt and Anders Skrondal. *The Cambridge dictionary of statistics.* Vol. 106. Cambridge University Press Cambridge, 2002

[9] Einar Geirsson. *Rollout Algorithms for Job-Shop Scheduling.* 05/2012

[11] Tilo Pfeifer and Robert Schmitt. *Masing Handbuch Qualitätsmanagement.* Carl Hanser Verlag GmbH Co KG, 2014

[12] Michael Pinedo. *Scheduling.* Springer, 2015

[13] Thomas Philip Runarsson, Marc Schoenauer, and Michele Sebag. “Pilot, Rollout and Monte Carlo Tree Search Methods for Job Shop Scheduling.” In: *LION*. Springer. 2012, pp. 160–174

[14] Matthew J Streeter and Stephen F Smith. “A simple distribution-free approach to the max k-armed bandit problem”. In: *International Conference on Principles and Practice of Constraint Programming.* Springer. 2006, pp. 560–574

[15] Matthew J Streeter and Stephen F Smith. “Selecting among heuristics by solving thresholded k-armed bandit problems”. In: (2006)

[16] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning : An Introduction.* MIT Press, 1998

[17] Gerald Teschl and Susanne Teschl. *Mathematik für Informatiker: Band 1: Diskrete Mathematik und Lineare Algebra.* Springer-Verlag, 2007

[18] Richard Y Wang and Diane M Strong. “Beyond accuracy: What data quality means to data consumers”. In: *Journal of management information systems* 12.4 (1996), pp. 5–33

Internet-resource

[10] *Google Optimization Tools The Job Shop Problem.* `https://developers.google.com/optimization/scheduling/job_shop`. Accessed: 2017-12-10

A. Appendix

A.1. TAG-Code

A.1.1. Class: AlgorithmTA

```
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.FileWriter;
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.io.Writer;
import java.security.GeneralSecurityException;
import java.sql.Timestamp;
import java.util.ArrayList;
import java.util.Formatter;
import java.util.HashSet;
import java.util.Random;

public class MCTSAlg {
   //Input-data
   private static int[][] p;
   private static int[][] sigma;
   private static int jobs;
   private static int machines;
   //Parameter
   private static int s;
   private static int compB;
   private static int k;
   private static double delta;
   private static int n;
   //Output-data
   private static int[][] bestSchedule;
private static double bestMakeSpan;
private static Node[][] sBest;

private static HashSet<Node> nodesList;

public static void main(String[] args) throws IOException{
   //Dimensions
   jobs = 3;
   machines = 3;
   s = 0;
   //Tested values for s
   int[] parameterS = new int[5];
   parameterS[0] = 200;
   parameterS[1] = 30;
   parameterS[2] = 50;
   parameterS[3] = 70;
   parameterS[4] = 100;
   //Number of the Example
   int exnum =8;
   //Number of Rollouts
   n =1000;

   p = new int[jobs][machines];
   sigma = new int[jobs][machines];
   inputData = new ArrayList<Integer>();
   for(int j = 0; j<parameterS.length; j++){
      s = parameterS[j];

      //Server Pfad
      File ff = new
         File("OutputData//"+jobs+"x"+machines+"//TA//Example"+exnum+"//results//n"+
      n+"//"+"s"+s+"DIM"+jobs+"x"+machines+".txt");
      File outputF = new
         File("OutputData//"+jobs+"x"+machines+"//TA//Example"+exnum+"//schedules//n"
      +n+"//"+"s"+s+"DIM"+jobs+"x"+machines+".txt");

      //Output-File
      BufferedWriter bw = new BufferedWriter(new FileWriter(ff,true));
      OutputStreamWriter(fout));
      BufferedWriter bw2 = new BufferedWriter(new FileWriter(outputF,
         true)); OutputStreamWriter(fout2));
      //Write into the Output-File
      for(int counter = 1; counter<6; counter++){
         nodesList = new HashSet<Node>();
         sBest = new Node[s][2];
         bestMakeSpan = Integer.MAX_VALUE;
         bestSchedule = new int[(jobs*machines)+1][2];
         k = jobs; // number of levers that can be pulled
         delta = 0.01; // error probability
         setInputData(); //fetches the input-data
         long beginT = System.currentTimeMillis();
         Node startNode = createRootNode(0);
         int end = 0;
         Node bestNextNode = null;
         while(end < (jobs*machines)){
            bestNextNode = doTASearch(startNode);
            startNode = new
                Node(bestNextNode.getT(),bestNextNode.getSchedule(), s) ;
            end = end +1;
            nodesList.clear();
            nodesList.add(startNode);
         }
         String string = ""+bestMakeSpan;
         bw = new BufferedWriter(new FileWriter(ff,true));
         bw.append(string);
         bw.newLine();
         bw.close();
         Formatter out;
         long endT = System.currentTimeMillis();
         long diff = endT-beginT;
         try{
            bw2 = new BufferedWriter(new FileWriter(outputF, true));
            writeIntoOutputFile2(bw2, diff);
         }
         catch(Exception e){
            System.out.println(e);
         }
      }
   }
}
```

```
//Writes the second Output-File (Content: Schedules)
public static void writeIntoOutputFile2(BufferedWriter b, long diff)
    throws IOException{
   int[][] results = new int[jobs][machines];
   for(int count1 = 1; count1<=(machines*jobs); count1++){
      int lineIndex = bestSchedule[count1][0]-1;
      int columnIndex = bestSchedule[count1][1]-1;
      results[lineIndex][columnIndex] = count1;
   }
   for(int count1 = 0; count1<jobs; count1++){
      String line = "";
      for(int count2 = 0; count2<machines; count2++){
         line = line+""+results[count1][count2]+"\t";
      }
      b.append(line);
      b.newLine();
   }
   b.append("BestMakespan:"+"\t"+ bestMakeSpan+"\t");
   b.newLine();
   b.append("Begin Time:"+"\t"+diff);
   b.newLine();
   b.append("Konstanten:");
   b.append("s ="+s);//write("s ="+s);
   b.newLine();
   b.append("compB="+compB);
   b.newLine();
   b.newLine();
   b.close();
}

public static void setInputData(){
   //Processing times
   p[0][0] = 13;
   p[0][1] = 5;
   p[0][2] = 8;
   p[1][0] = 2;
   p[1][1] = 7;
   p[1][2] = 12;
   p[2][0] = 21;
   p[2][1] = 9;
   p[2][2] = 7;
```

```
    //Routes
    sigma[0][0] = 0;
    sigma[0][1] = 1;
    sigma[0][2] = 2;
    sigma[1][0] = 2;
    sigma[1][1] = 0;
    sigma[1][2] = 1;
    sigma[2][0] = 2;
    sigma[2][1] = 1;
    sigma[2][2] = 0;
  }

  //4 Steps of MCTS
  public static Node doTASearch(Node root){
    int compBudget = 1;
    while(compBudget<=n){
      Node lastNode = doTreePolicy(root);
      Node finalNode = doDefaultPolicy(lastNode);
      doBackUp(lastNode, finalNode);
      compBudget++;
    }
    return findBestChild(root);
  }
  //Creates root-node with empty schedule
  public static Node createRootNode(int index){
    int[] t = new int[jobs];
    for(int i = 0; i<jobs; i++){
      t[i] = 0;
    }
    int[][] rootSchedule = new int[(jobs*machines)+1][2];
    Node rootNode = new Node(t, rootSchedule, s);
    nodesList.add(rootNode);
    return rootNode;
  }
  //Execute the expand and the select step
  public static Node doTreePolicy(Node dn){
    while(isNonTerminal(dn)){
      if(notFullyExpanded(dn)){
        return expand(dn);
      }
      else{
            dn = findBestChild(dn);
        }
    }
    return dn;
}
//Tests whether a node is a leaf-node
public static boolean isNonTerminal(Node no){
    for(int i = 0; i<jobs; i++){
        if(no.getT()[i]<machines){
            return true; // the node is not terminal
        }
    }
    return false; // the node is terminal
}
//Tests whether every child of a node has been visited at least once
public static boolean notFullyExpanded(Node no){
    int l1 = no.getT().length;
    int[] tCopy = new int[l1];
    int count = 0;
    for(int i = 0; i<l1; i++){
        tCopy[i] = no.getT()[i];
        if(tCopy[i]<machines){
            count++;
        }
    }
    if(no.getChildren().size()<count){
        return true; // the node is not fully expanded
    }
    return false; // the node is fully expanded
}
//Expand-Step
public static Node expand(Node e){
    int[] tCopy = new int[jobs];
    int count = 1;
    for(int i = 0; i<jobs; i++){
        tCopy[i] = e.getT()[i];
        count = count + tCopy[i]; // index of the new entry in the schedule
    }
    int l1 = (jobs*machines)+1;
    int[][] partialSchedule = new int[l1][2];
    System.arraycopy(e.getSchedule(), 0, partialSchedule, 0, l1);
    for(int i = 0; i<jobs; i++){
        if(tCopy[i]+1<=machines){ // adding the expand-child to its parent
            tCopy[i] = tCopy[i] +1;
            partialSchedule[count][0] = i+1;
            partialSchedule[count][1] = sigma[i][tCopy[i]-1];
            Node newChild = new Node(tCopy, partialSchedule,s);
            e.addChild(newChild);
            if(isExistingInList(newChild)==false){
              newChild.setParent(e);
              nodesList.add(newChild);
              tCopy[i] = tCopy[i] -1;
              return newChild;
            }
            tCopy[i] = tCopy[i] -1;
        }
    }
    return null;
}

public static boolean isExistingInList(Node ex){
    return nodesList.contains(ex);
}

public static Node findBestChild(Node bc){
    int n_i = 0;
    int s_i = 0;
    double alpha = 0;
    int h_ges = calculateHges(bc);
    double h = Integer.MIN_VALUE;
    Node bestNode = null;
    double bestValue = 0;
    k = bc.getChildren().size();
    for(int i =0; i<bc.getChildren().size(); i++){
        Node parent = bc;
        n_i = bc.getChildren().get(i).getVisits();
        if(n_i>0){
            s_i = countNode(bc.getChildren().get(i), bc.getSBestList());
            alpha = Math.log(2*h_ges*k/delta);
            h = (s_i + alpha +Math.sqrt((2*s_i*alpha)+alpha*alpha))/n_i;
        }
        else{
            h = Integer.MAX_VALUE;
         }
         if(h>=bestValue){
            bestValue = h;
            bestNode =bc.getChildren().get(i);
         }
      }
      return bestNode;
   }

   public static int calculateHges(Node nod){
      int sum =0;
      int l = nod.getChildren().size();
      for(int i = 0;i<l; i++){
         sum = sum +nod.getChildren().get(i).getVisits();
      }
      return sum;
   }

   public static int countNode(Node n1, Node[][] sL){
      int counter = 0;
      int l = sL.length;
      for(int i = 0;i<l; i++){
         if(sL[i][1]!=null){
            if(n1.getKey().equals(sL[i][1].getKey())){
               counter = counter +1;
            }
         }
      }
      return counter;
   }

   public static Node doDefaultPolicy(Node n){
      // chooses randomly a path from n to the end
      Node endNode = null;
      while(isNonTerminal(n)){
         Random rand = new Random();
         int[] tCopy = new int[jobs];
         ArrayList<Integer> indexList = new ArrayList<Integer>();
         int count = 1;
          for(int i = 0; i<jobs; i++){
            tCopy[i] = n.getT()[i];
            count = count +tCopy[i];
            if(tCopy[i]<machines){
               indexList.add(i);
            }
         }
         indexList.trimToSize();
         int randomIndex = 0;
         if(indexList.size()>1) randomIndex = rand.nextInt(indexList.size());
         tCopy[indexList.get(randomIndex)] =
             tCopy[indexList.get(randomIndex)] +1;
         int l1 = (jobs*machines)+1;
         int[][] partialS = new int[l1][2];
         System.arraycopy(n.getSchedule(), 0, partialS, 0, l1);
         partialS[count][0] = indexList.get(randomIndex)+1;
         partialS[count][1] =
             sigma[indexList.get(randomIndex)][tCopy[indexList.get(randomIndex)]-1];
         n = new Node(tCopy, partialS, s);
         endNode = n;
      }
      return endNode;
   }

   private static void addNodesToSBest(Node begin, Node end){
      double rewardOfEnd = calculateReward(end);
      //If the list is already full
      if(sBest[s-1][0]!=null){
         sBest[s-1][0] = null;
         sBest[s-1][1] = null;
         for(int i=s-1; i>=0; i--){
            if(sBest[i][0] != null){
               if(rewardOfEnd<=calculateReward(sBest[i][0])){
                  sBest[1+i][0]=sBest[i][0];
                  sBest[1+i][1]=sBest[i][1];
                  if(i==0){
                     sBest[i][0] = end;
                     sBest[i][1] = begin;
                     break;
                  }
                  if(rewardOfEnd>calculateReward(sBest[i-1][0])){
                     sBest[i][0] = end;
                     sBest[i][1] = begin;
                     break;
                  }
               }
            }
         }
      }
      if(sBest[s-1][0]==null){
         for(int i=s-1; i>=0; i--){
            if(sBest[i][0] != null){
               if(rewardOfEnd<=calculateReward(sBest[i][0])){
                  sBest[1+i][0]=sBest[i][0];
                  sBest[1+i][1]=sBest[i][1];
                  if(i==0){
                     sBest[i][0] = end;
                     sBest[i][1] = begin;
                     break;
                  }
                  if(rewardOfEnd>calculateReward(sBest[i-1][0])){
                     sBest[i][0] = end;
                     sBest[i][1] = begin;
                     break;
                  }
               }
               else{
                  sBest[1+i][0]=end;
                  sBest[1+i][1]=begin;
                  break;
               }
            }
            if(i==0 && sBest[i][0] == null){
               sBest[i][0] = end;
               sBest[i][1] = begin;
               break;
            }
         }
      }
   }

   public static void sortSBest(){
      for(int i = 0; i<s-1; i++){
           for(int j = i+1; j<s; j++){
               double value1 = Integer.MAX_VALUE;
               double value2 = Integer.MAX_VALUE;
               if(sBest[i][0] != null){
                   value1 = calculateReward(sBest[i][0]);
               }
               if(sBest[j][0] != null){
                   value2 = calculateReward(sBest[j][0]);
               }
               if(value1>value2){
                   Node help1 = sBest[i][0];
                   sBest[i][0] = sBest[j][0];
                   sBest[j][0] = help1;
                   Node help2 = sBest[i][1];
                   sBest[i][1] = sBest[j][1];
                   sBest[j][1] = help2;
               }
           }
       }
   }

   private static double calculateReward(Node no){
       int[] jobFinishingTime = new int[jobs];
       int[] machineIsFreeTime = new int[machines];
       int[][] s = new int[bestSchedule.length][2];
       int s_l = s.length;
       System.arraycopy(no.getSchedule(), 0, s, 0, s_l);
       for(int i = 1; i<s.length; i++){
           if(jobFinishingTime[s[i][0]-1]>machineIsFreeTime[s[i][1]-1]){
               jobFinishingTime[s[i][0]-1] = jobFinishingTime[s[i][0]-1] +
                   p[s[i][0]-1][s[i][1]-1];
               machineIsFreeTime[s[i][1]-1] = jobFinishingTime[s[i][0]-1];
           }
           else{
               machineIsFreeTime[s[i][1]-1] = machineIsFreeTime[s[i][1]-1] +
                   p[s[i][0]-1][s[i][1]-1];
               jobFinishingTime[s[i][0]-1] = machineIsFreeTime[s[i][1]-1];
           }
       }
       double shortesTime=Integer.MIN_VALUE;
       for(int i = 0; i<jobFinishingTime.length; i++){
             if(shortesTime <= jobFinishingTime[i]){
               shortesTime = jobFinishingTime[i];
             }
           }
           if(bestMakeSpan>=shortesTime){
             bestMakeSpan = shortesTime;
             System.arraycopy(s, 0, bestSchedule, 0, s_l);
           }
           return shortesTime;
         }

         private static void doBackUp(Node n1, Node n2){
         // Updates the visits
           double reward = 0;
           while(n1!=null){
             if(isNonTerminal(n1)==false) n2 = n1;
             reward = calculateReward(n2);
             n1.setReward(reward);
             if(n1.getParent()!=null){
               setList(n1.getParent().getSBestList());
               addNodesToSBest(n1, n2);
               n1.getParent().updateSBestList(sBest);
               n1.increaseVisits();
             }
             if(n1.getParent()==null) {
               n1.increaseVisits();
               n1 = null;
             }
             else{
               n1 = n1.getParent();
             }
           }
         }

         private static void setList(Node[][] nlist){
           System.arraycopy(nlist, 0, sBest, 0, sBest.length);
         }
       }
```

A.1.2. Class: Node

```java
import java.util.ArrayList;

public class Node {

   private int[] t;
   private int[][] schedule;
   private ArrayList<Node> children;
   private Node[][] sBestList;
   private int visits;
   private double reward;
   private Node parent;
   private String key;
   private int t_crit;
   private int h_ges;
   private int lengthOfSBest;

   public Node(int[] x, int[][] s, int lengthSBest){
      int l1 = x.length;
      t = new int[l1];
      key ="";
      for(int i = 0; i<l1; i++){
         t[i] = x[i];
         key = key+""+t[i]+";";
      }
      int l2 = s.length;
      schedule = new int[l2][2];
      for(int i = 0; i<l2; i++){
         for(int j = 0; j<2; j++){
            schedule[i][j] = s[i][j];
            key = key+"*"+schedule[i][j];
         }
      }
      children = new ArrayList<Node>();
      visits = 0;
      reward = 0;
      parent = null;
      lengthOfSBest = lengthSBest;
      sBestList = new Node[lengthOfSBest][2];
      h_ges =0;
   }
```

```
public int[] getT() {
   return t;
}

public int[][] getSchedule() {
   return schedule;
}

public ArrayList<Node> getChildren() {
   return children;
}

public void deleteChildren(){
   children.clear();
   children.trimToSize();
}

public String getKey(){
   return key;
}

public void addChild(Node child) {
   boolean alreadyExisting =false;
   for(int i = 0; i<children.size(); i++){
      if(child.getKey().equals(children.get(i).getKey())){
         alreadyExisting = true;
      }
   }
   if(alreadyExisting==false){
      children.add(child);
   }
}

public int getVisits() {
   return visits;
}

public void increaseVisits() {
   visits = visits+1;
}
```

```
public void setBackVisits(){
   visits =0;
}

public double getReward() {
   return reward;
}

public void setBackReward(){
   reward =0;
}

public void setReward(double reward) {
   this.reward = reward;
}

public Node getParent() {
   return parent;
}

public void setParent(Node parent) {
   this.parent = parent;
}

public void setSBest(Node[][] xy){
   int l1 = xy.length;
   for(int i = 0; i<l1; i++){
      sBestList[i][0] = xy[i][0];
      sBestList[i][1] = xy[i][1];
   }
}

public void setBackSBest(){
   sBestList = new Node[lengthOfSBest][2];
}

public void updateSBestList(Node[][] list){
   for(int i = 0; i<list.length; i++){
      for(int j = 0; j<list[0].length; j++){
         sBestList[i][j]= list[i][j];
      }
```

```
            }
        }

        public Node[][] getSBestList(){
            return sBestList;
        }

        public int getH(){
            return h_ges;
        }

        public void setBackHges(){
            h_ges =0;
        }

        public void increaseH(){
            h_ges = h_ges +1;
        }
        public int hashCode() {
            return key.hashCode();
        }

        public boolean equals(Object otherNode) {
            return (otherNode instanceof Node &&
                key.equals(((Node)otherNode).key));
        }
    }
```

A.2. UCT-Code

A.2.1. Class: MCTSAlgorithm

```
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileOutputStream;
import java.io.FileWriter;
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.Random;

public class MCTSAlgorithm {
   //Input-data
   private static int[][] p;
   private static int[][] sigma;
   private static int jobs;
   private static int machines;
   //Parameters
   private static double c;
   private static double w;
   private static int compB;
   //Output-data
   private static int[][] bestSchedule;
   private static double bestMakeSpan;

   private static HashSet<Node> nodesList;

   public static void main(String[] args) throws IOException {

      double[] cValues1 = new double[5];
      String[] cValues2 = new String[5];
      setCValues(cValues1, cValues2);
      int firstL = cValues1.length;
      for(int x = 0; x<firstL; x++){
         w = 1600;
         c = cValues1[x];
         String cc = cValues2[x];
         compB = 1000;

         int jnum = 14;
         int mnum = 14;
         int exnum =7;
         p = new int[jnum][mnum];
         sigma = new int[jnum][mnum];
         setInputData();
         jobs = p.length;
         machines = p[0].length;
         //Server
         File ff = new
            File("OutputData//"+jnum+"x"+mnum+"//UCT//Example"+exnum+"//results//n"+
compB+"//"+"c"+cc+"n"+compB+"DIM"+jnum+"x"+mnum+".txt");
File outputF = new
   File(("OutputData//"+jnum+"x"+mnum+"//UCT//Example"+exnum+"//schedules//n"+
compB+"//"+"c"+cc+"DIM"+jnum+"x"+mnum+".txt"));

BufferedWriter bw = new BufferedWriter(new
   FileWriter(ff,true));//new OutputStreamWriter(fout));
BufferedWriter bw2 = new BufferedWriter(new FileWriter(outputF,
   true));//new OutputStreamWriter(fout2));

for(int counter = 1; counter <16; counter++){
   long beginT = System.currentTimeMillis();
   bestSchedule = new int[(jobs*machines)+1][2];
   bestMakeSpan = Integer.MAX_VALUE;
   worstMakeSpan = Integer.MIN_VALUE;
   nodesList = new HashSet<Node>();

   Node startNode = createRootNode(0);
   int end = 0;
   Node bestNextNode = null;
   while(end < (jobs*machines)){
      bestNextNode = uctSearch(startNode);
      startNode = new
         Node(bestNextNode.getT(),bestNextNode.getSchedule()) ;
      end = end +1;
      nodesList.clear();
      nodesList.add(startNode);
   }
   String string = ""+bestMakeSpan;
   bw = new BufferedWriter(new FileWriter(ff,true));
   bw.append(string);
   bw.newLine();
   bw.close();
   long endT = System.currentTimeMillis();
   long diff = endT-beginT;
   try{
      bw2 = new BufferedWriter(new FileWriter(outputF, true));
      writeIntoOutputFile2(bw2, diff);
   }
   catch(Exception e){
      System.out.println(e);
```

```
            }
         }
      }
   }

   public static void setCValues(double[] list1, String[] list2){
      list1[0] = 0.01;
      list1[1] = 0.1;
      list1[2] = 0.5;
      list1[3] = 1;
      list1[4] =2;
      list2[0] = "0_01";
      list2[1] = "0_1";
      list2[2] = "0_5";
      list2[3] = "1";
      list2[4] = "2";
   }

   public static void writeIntoOutputFile2(BufferedWriter b, long diff)
       throws IOException{
      int[][] results = new int[jobs][machines];
      for(int count1 = 1; count1<=(machines*jobs); count1++){
         int lineIndex = bestSchedule[count1][0]-1;
         int columnIndex = bestSchedule[count1][1]-1;
         results[lineIndex][columnIndex] = count1;
      }

      for(int count1 = 0; count1<jobs; count1++){
         String line = "";
         for(int count2 = 0; count2<machines; count2++){
            line = line+""+results[count1][count2]+"\t";
         }
         b.append(line);
         b.newLine();
      }
      b.append("BestMakespan:"+"\t"+ bestMakeSpan+"\t");
      b.newLine();
      b.append("Time:"+"\t"+ diff);
      b.newLine();
      b.append("Konstanten:");
      b.newLine();
      b.append("c ="+c);
      b.newLine();
      b.append("compB="+compB);
      b.newLine();
      b.newLine();
      b.close();
   }

   public static void setInputData(){

   }

   public static Node uctSearch(Node root){
      int compBudget = 1;
      while(compBudget<=compB){
         Node lastNode = doTreePolicy(root);
         double delta = doDefaultPolicy(lastNode);
         doBackUp(lastNode, delta);
         compBudget++;
      }
      return findBestChild(root);
   }

   public static Node createRootNode(int index){
      int[] t = new int[jobs];
      for(int i = 0; i<jobs; i++){
         t[i] = 0;
      }
      int[][] rootSchedule = new int[(jobs*machines)+1][2];
      Node rootNode = new Node(t, rootSchedule);
      nodesList.add(rootNode);
      return rootNode;
   }

   public static Node doTreePolicy(Node aNode){
      while(isNonTerminal(aNode)){
         counterTP = counterTP+1;
         if(notFullyExpanded(aNode)){
            return expand(aNode);
         }
         else{
            aNode = findBestChild(aNode);
        }
    }
    return aNode;
}

public static boolean isNonTerminal(Node no){
    for(int i = 0; i<jobs; i++){
        if(no.getT()[i]<machines){
            return true;
        }
    }
    return false;
}

public static boolean notFullyExpanded(Node no){
    int l1 = no.getT().length;
    int[] tCopy = new int[l1];
    int count = 0;
    for(int i = 0; i<l1; i++){
        tCopy[i] = no.getT()[i];
        if(tCopy[i]<machines){
            count++;
        }
    }
    if(no.getChildren().size()<count){
        return true;
    }
    return false;
}

public static Node expand(Node e){
    int[] tCopy = new int[jobs];
    int count = 1;
    for(int i = 0; i<jobs; i++){
        tCopy[i] = e.getT()[i];
        count = count + tCopy[i];
    }
    int l1 = (jobs*machines)+1;
    int[][] partialSchedule = new int[l1][2];
    System.arraycopy(e.getSchedule(), 0, partialSchedule, 0, l1);
    for(int i = 0; i<jobs; i++){
        if(tCopy[i]+1<=machines){
            tCopy[i] = tCopy[i] +1;
            partialSchedule[count][0] = i+1;
            partialSchedule[count][1] = sigma[i][tCopy[i]-1];
            Node newChild = new Node(tCopy, partialSchedule);
            e.addChild(newChild);
            if(isExistingInList(newChild)==false){
              newChild.setParent(e);
              nodesList.add(newChild);
              tCopy[i] = tCopy[i] -1;
              return newChild;
            }
            tCopy[i] = tCopy[i] -1;
        }
    }
    return null;
}

public static boolean isExistingInList(Node ex){
    return nodesList.contains(ex);
}

public static Node findBestChild(Node bc){
    // Minimieren
    double bestValue = Double.MAX_VALUE;
    Node bestChildNode = null;
    int l = bc.getChildren().size();
    for(int i = 0; i<l; i++){
        Node childN = bc.getChildren().get(i);
        double childReward = childN.getReward();
        double childVisits = childN.getVisits();
        double parentVisits = bc.getVisits();
        double term1 = childReward/childVisits;
        if(childReward <1 || childVisits < 1){
            term1 = 0;
        }
        double term2 = 0;
        if(parentVisits>1 && childVisits >0){
            term2 = w*c*Math.sqrt((2*Math.log(parentVisits))/childVisits);
        }
      double value = term1 - term2;
      if(bestValue>value){
         bestValue = value;
         bestChildNode = childN;
      }
   }
   try {
      bestChildNode.setParent(bc);
   } catch (Exception e) {
      e.printStackTrace();
   }
   return bestChildNode;
}

public static double doDefaultPolicy(Node n){
   counterDP = counterDP+1;
   while(isNonTerminal(n)){
      Random rand = new Random();
      int[] tCopy = new int[jobs];
      ArrayList<Integer> indexList = new ArrayList<Integer>();
      int count = 1;
      for(int i = 0; i<jobs; i++){
         tCopy[i] = n.getT()[i];
         count = count +tCopy[i];
         if(tCopy[i]<machines){
            indexList.add(i);
         }
      }
      indexList.trimToSize();
      int randomIndex = 0;
      if(indexList.size()>1) randomIndex = rand.nextInt(indexList.size());
      tCopy[indexList.get(randomIndex)] =
          tCopy[indexList.get(randomIndex)] +1;
      int l1 = (jobs*machines)+1;
      int[][] partialS = new int[l1][2];
      System.arraycopy(n.getSchedule(), 0, partialS, 0, l1);
      partialS[count][0] = indexList.get(randomIndex)+1;
      partialS[count][1] =
          sigma[indexList.get(randomIndex)][tCopy[indexList.get(randomIndex)]-1];
      n = new Node(tCopy, partialS);
   }
```

```
    return calculateReward(n);
}

private static double calculateReward(Node no){
    int[] jobFinishingTime = new int[jobs];
    int[] machineIsFreeTime = new int[machines];
    int[][] s = new int[bestSchedule.length][2];
    int sL = s.length;
    System.arraycopy(no.getSchedule(), 0, s, 0, sL);
    for(int i = 1; i<s.length; i++){
        if(jobFinishingTime[s[i][0]-1]>machineIsFreeTime[s[i][1]-1]){
            jobFinishingTime[s[i][0]-1] = jobFinishingTime[s[i][0]-1] +
                p[s[i][0]-1][s[i][1]-1];
            machineIsFreeTime[s[i][1]-1] = jobFinishingTime[s[i][0]-1];
        }
        else{
            machineIsFreeTime[s[i][1]-1] = machineIsFreeTime[s[i][1]-1] +
                p[s[i][0]-1][s[i][1]-1];
            jobFinishingTime[s[i][0]-1] = machineIsFreeTime[s[i][1]-1];
        }
    }
    double shortesTime=Integer.MIN_VALUE;
    for(int i = 0; i<jobFinishingTime.length; i++){
        if(shortesTime <= jobFinishingTime[i]){
            shortesTime = jobFinishingTime[i];
        }
    }
    if(bestMakeSpan>=shortesTime){
        bestMakeSpan = shortesTime;
        System.arraycopy(s, 0, bestSchedule, 0, sL);
    }
    if(worstMakeSpan<=shortesTime){
        worstMakeSpan = shortesTime;
    }
    return shortesTime;
}

private static void doBackUp(Node n, double d){
    while(n!=null){
        n.increaseVisits();
        n.setReward(n.getReward()+d);
        if(n.getParent()==null) n = null;
        else n = n.getParent();
      }
    }
  }
```

A.2.2. Class: Node

```
import java.util.ArrayList;

public class Node {

  private int[] t;
  private int[][] schedule;
  private ArrayList<Node> children;
  private int visits;
  private double reward;
  private Node parent;
  private String key;

  public Node(int[] x, int[][] s){
    int l1 = x.length;
    t = new int[l1];
    key ="";
    for(int i = 0; i<l1; i++){
      t[i] = x[i];
      key = key+""+t[i]+";";
    }
    int l2 = s.length;
    schedule = new int[l2][2];
    for(int i = 0; i<l2; i++){
      for(int j = 0; j<2; j++){
        schedule[i][j] = s[i][j];
        key = key+"*"+schedule[i][j];
      }
    }
    children = new ArrayList<Node>();
    visits = 0;
    reward = 0;
    parent = null;
}

public int[] getT() {
   return t;
}

public int[][] getSchedule() {
   return schedule;
}

public ArrayList<Node> getChildren() {
   return children;
}

public String getKey(){
   return key;
}

public void addChild(Node child) {
   boolean alreadyExisting =false;
   for(int i = 0; i<children.size(); i++){
      if(child.getKey().equals(children.get(i).getKey())){
         alreadyExisting = true;
      }
   }
   if(alreadyExisting==false){
      children.add(child);
   }
}

 public int getVisits() {
   return visits;
}

public void increaseVisits() {
   visits = visits+1;
}

public double getReward() {
   return reward;
}
```

```
    public void setReward(double reward) {
        this.reward = reward;
    }

    public Node getParent() {
        return parent;
    }

    public void setParent(Node parent) {
        this.parent = parent;
    }

    public int hashCode() {
        return key.hashCode();
    }

    public boolean equals(Object otherNode) {
        return (otherNode instanceof Node &&
            key.equals(((Node)otherNode).key));
    }
}
```