UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

Monte Carlo Tree Search Applied to Finite-Capacity Scheduling Optimization

Bruno Schrappe

SCS_3547_006 Intelligent Agents and Reinforcement Learning - Prof. Larry Simon - August 2020

Abstract

Scheduling industrial operations competing for limited resources has been the subject of operations research for years with limited success given its intractable nature as an NP-hard problem [1, 3, 4].

More recently, Monte Carlo Tree Search (MCTS) and deep learning approaches have been considered [3, 4], given recent successes on other fields such as gaming (i.e. Alpha Go) [11].

As a preliminary investigation on the feasibility of deep reinforcement learning to scheduling optimization, this project explores MCTS with upper confidence bounds for trees (UCT) as well as pruning and alternative exploitation/exploration decisions using e-greedy algorithms. A complete Python scheduling application was purpose-built to facilitate research and is herein presented.

Introduction

Industrial scheduling has historically depended on human expert knowledge or problem-specific dedicated systems. Although the problem definition is deceptively simple, it is an NP-hard problem as the quantity of possible states on a planning board rises in proportion to the number of orders to the power of number of tasks.

A generic model for the problem involves a set of orders that require processing time on one or more resources with specific constraints, as shown on the blue order below:

The order needs sequential processing steps on resources 1, 2, 3 and 4 and has a single constraint on earliest start time on Resource 1, identified by the solid line preceding the task.

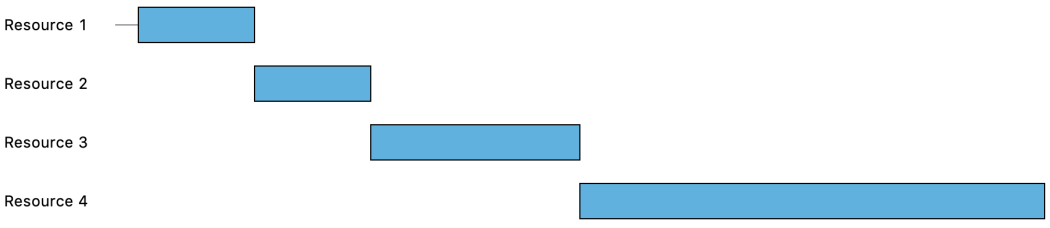

Figure 1 - Example of a scheduling plan order with an earliest start constraint on the first task

So as to illustrate the scheduling challenge, consider the green order shown next, competing for resources 1, 2 and 3 with no earliest start constraint:

Figure 2 - Example of a scheduling plan order

A basic operations research scheduling algorithm would pick the first available tasks on each resource and schedule it, resulting on the following scheduling plan:

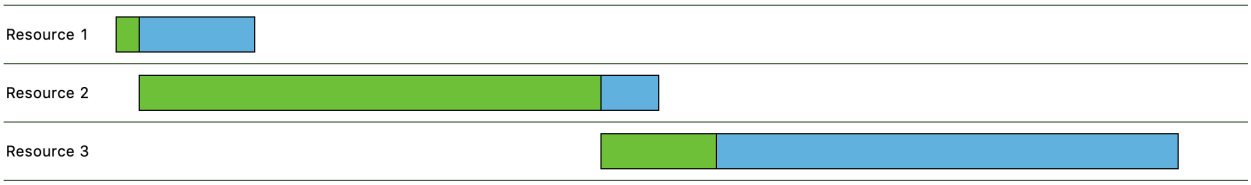

Figure 3 - Basic scheduling rule applied with no look-ahead capabilities

Even though the green task can be scheduled before of the Blue task on Resource 2, it does prevent the Blue Task from being executed early on Resource 2 and consequently, on Resource 3. In order to optimize this scheduling plan, look-ahead capabilities are needed before deciding on each possible task. A properly optimized scheduling plan, reducing overall lead time, is shown below:

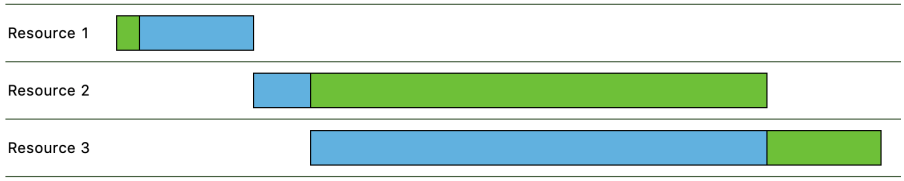

Figure 4 - Optimized schedule minimizing overall lead time, achievable with look-ahead decisions

Having established the need for look-ahead capabilities, possible techniques include:

Exhaustive tree search, where every possible final scheduling state is computed, which is only feasible for the simplest cases as this is an NP-hard problem.

On the extremely simple example shown in Figure 4, with 2 orders and 6 tasks, there are 16 possible branches.

A slightly more complex scenario with 3 orders and 10 tasks shown in Figure 5 below requires analysis of 1,585 branches:

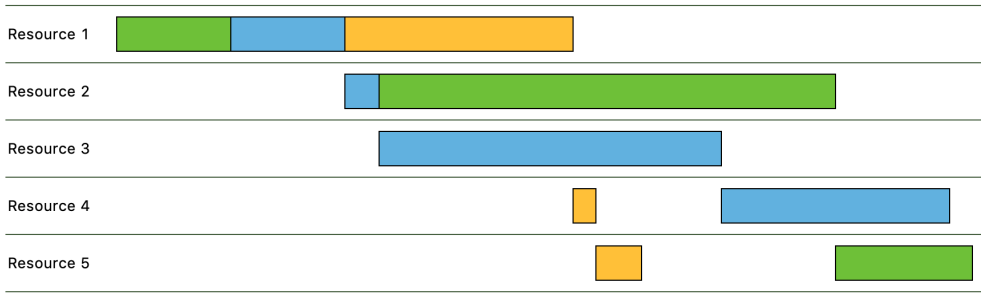

Figure 5 - Slightly more complex plan with 3 orders and 10 tasks, for which 1,585 scheduling states are found

Including a fourth order with a single task increased the number of possible scheduling paths to 8,622 as shown below in Figure 6:

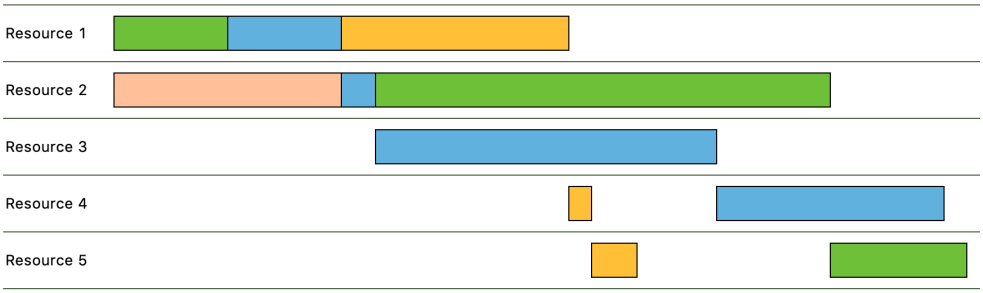

Figure 6 - Scheduling plan with 4 orders and 11 tasks with 8,622 possible scheduling states

Expert rule-based systems, which may be efficient on specific scenarios but do not provide a generic solution for the scheduling problem.

Tree search techniques that reduce the number of required branches to analyze, including Monte Carlo Tree Search (MCTS) which are the focus of this project.

MCTS differs from classical Monte Carlo simulations by progressively selecting actions that lead to preferred end states, and has been famously applied to Alpha-Go, combined with deep learning to defeat the reigning Go game champion, Lee Sedol in March 2016.

This project investigates the feasibility of applying MCTS to a generic production scheduling engine, setting the stage for future work involving deep learning techniques.

Production scheduling is a Markov Decision Process wherein a possible action is selected from a list on each state, leading to a future state. An action involves selecting an outstanding task and scheduling it on its required resource, given resource and task constraints. Once scheduled, the resulting state can be represented as a Gantt chart with tasks scheduled on resources as well as the list of outstanding (not yet scheduled) tasks.

A given state is considered terminal if no tasks are outstanding.

Unlike games, where a tree search involves winning or reward probabilities, any searched path leading to a terminal state on a scheduling application can be considered as a viable path and can be selected as a solution, if so desired.

Approach and Solution

In order to quickly iterate through and visualize scheduling plans, a Python application that displays resources and allocated tasks over time on a Gantt-style chart has been developed, sharing the code base with our proposed scheduling algorithms. The simple UI uses native Python Tkinter libraries and provides useful graphical feedback and insights when tuning scheduling algorithms:

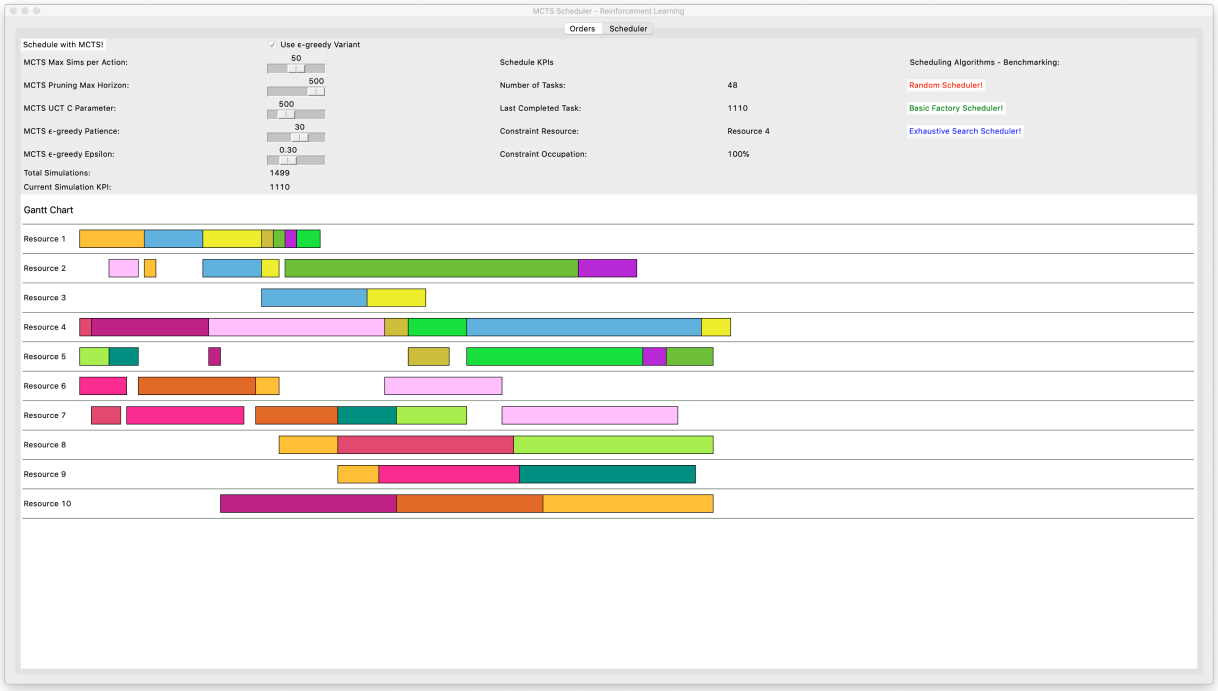

Figure 7 - Application user interface showing a fully scheduled plan

Tasks are assigned to specific resources based on their Order requirements. The Order class has name, color, earliest start date/ time and a list of production steps as properties.

The panel on the right shows the data model describing an order depicted in Figure 2.

The application reads JSON data files containing orders, so we were able to iterate through multiple plans to extensively test proposed algorithms. JSON compatibility may facilitate future integration to external production systems if so desired.

```
{
      "name": "Order 2",
      "color": "#6ec038",
      "earliest_start": 0,
      "steps": [
        {
          "step": 1,
          "resource": 1,
          "duration": 100,
          "predecessor": null
        },
        {
          "step": 2,
          "resource": 2,
          "duration": 400,
          "predecessor": 1
        },
        {
          "step": 3,
          "resource": 5,
          "duration": 120,
          "predecessor": 2
        }
       ]
}
```

A Production Plan class object (represented on the right) is a collection of steps coming from work orders, with attributes including:

- Sequence, as the sorting order used to schedule the task
- Index, a sequential number of a task on a plan, used to identify predecessors / parents
- Parent order to which the task belongs
- Resource on which the task is required to be processed
- Predecessor task
- Earliest start, which may be an Order-related constraint or a constraint posed by scheduling its predecessor
- Time duration
- Scheduled start time, after the task has been scheduled
- Scheduled finished time.

```
[{'sequence': None,
  'index': 1,
  'order': 'Order 1',
  'color': '#62b1df',
  'resource': 1,
  'predecessor': None,
  'earliest_start': 100,
  'duration': 100,
  'start': None,
  'finish': None},
 {'sequence': None,
  'index': 2,
  'order': 'Order 1',
  'color': '#62b1df',
  'resource': 2,
  'predecessor': 1,
  'earliest_start': None,
  'duration': 30,
  'start': None,
  'finish': None},
…
```

A scheduling State is thus defined by the state of its Production Plan. A terminal state is reached when all tasks have start and finish dates (i.e. there are no outstanding tasks).

For non-terminal states, possible Actions include selection of unscheduled tasks with no dependencies (no unscheduled predecessors) for inclusion on the schedule at the earliest possible date between the task earliest start date and the corresponding resource earliest availability date. An interesting complication is computing available “windows” of resource availability in which a task could fit, which has also been implemented.

Reward functions depend on scheduling goals, which may include minimizing work-in-progress, minimizing lead time or maximizing throughput. For this experiment, we established a goal of maximizing throughput producing a batch of orders in the minimum possible time. With that, the reward observed when a terminal state is reached is the completion date of the last task, which we would like to minimize.

As a key performance indicator (not used on the reward function), one can look at how busy the constraint resource was kept during the scheduling plan interval. A constraint is defined as the resource with the highest workload, and of course no plan can finish in a time shorter than the sum of all task durations assigned to the constraint resource.

Given the desired scheduling goal of this experiment, we can consider a scheduling solution as optimal if the total plan duration is equal to the sum of all task durations assigned to the constraint resource. In other words, if the constraint resource was kept busy with no idle windows across the entire schedule. Such is the solution depicted on Figure 7, where Resource 4 is fully booked across the schedule.

Basic Operations Research Scheduler

For benchmarking, a basic scheduler algorithm commonly used was implemented using the algorithm shown in Figure 8 below:

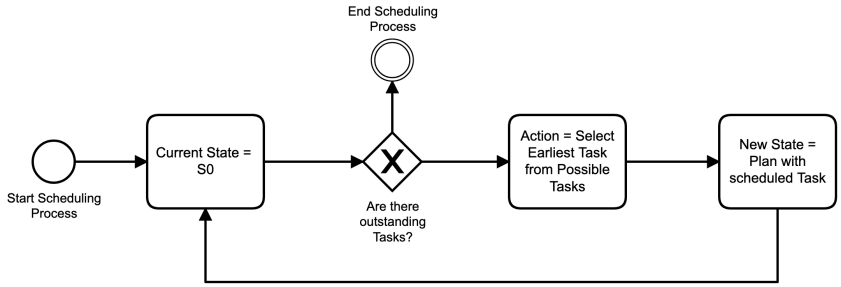

Figure 8 - Basic Operations Research Scheduling Algorithm

In a nutshell, given a current state, it recursively selects an Action from possible actions (outstanding tasks), whereas the policy selects the task with the earliest possible start time, includes it on the plan (schedules it) and reiterates until a terminal state with no outstanding tasks is reached. As this simple algorithm has no look-ahead capabilities, it is prone to scheduling mistakes as displayed on Figure 3, on which this scheduler was used.

Exhaustive Tree Search Scheduler

So as to explore the limits of full tree search on the scheduling problem, a recursive scheduling algorithm that explores all possible states has been implemented, following the algorithm below:

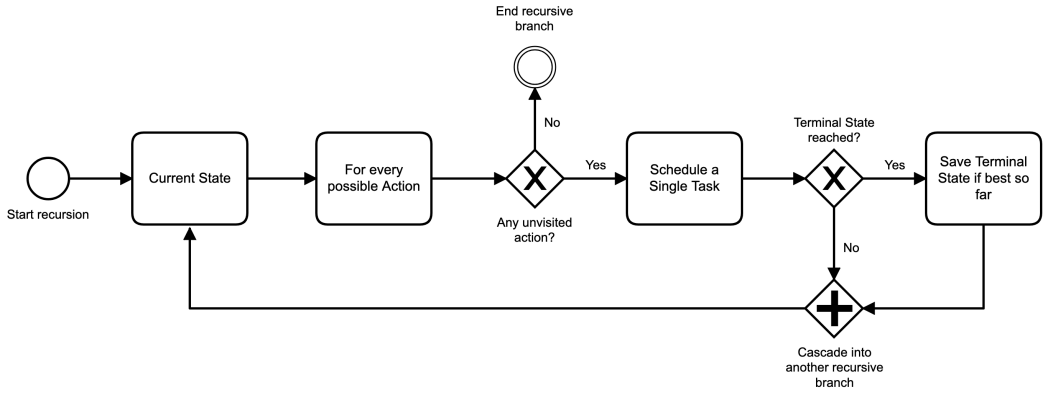

Figure 9 - Exhaustive Tree Search Scheduling Algorithm

The algorithm finds optimal solutions based on any reward function but is limited to very few orders and tasks as the number of possible tree nodes quickly escalates into an intractable problem given its NP-hard nature. Just as an example, for an order set of 10 orders, each of which with three tasks (which is still a very simple problem), the number of possible ways to schedule a plan is proportional to (albeit slightly less than) 3^30, as for the first state one can choose from three possible actions, then additional 3 actions on the second state and so on until all tasks from all orders are scheduled.

A mathematical analysis of this problem is provided by [1] but for the scope of this project it suffices to consider the full-search problem as intractable for any real practical application.

The plans below show experimental results and exploration of all possible paths using the full tree search algorithm:

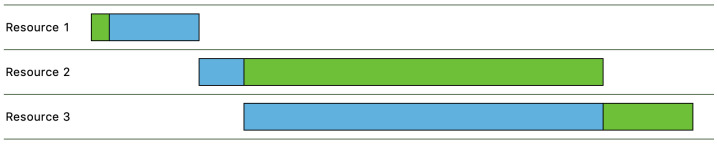

Figure 10 - Exhaustive Tree Search applied to 2 orders, 6 tasks and 20 possible solution paths

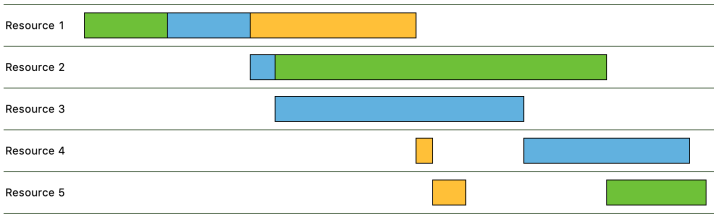

Figure 11 - Exhaustive Tree Search applied to 3 orders, 10 tasks and 3,132 possible solution paths

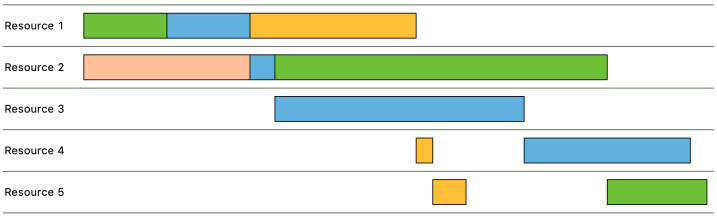

Figure 12 - Exhaustive Tree Search applied to 4 orders, 11 tasks and 23,167 possible solution paths

Any plan more complex than the one depicted in Figure 12 is not a good candidate for exhaustive search given the required computation.

Random Scheduler

In order to support random rollouts to be used on Monte Carlo Tree search algorithms, a random scheduler was implemented, capable of randomly choosing possible actions from any given state, as shown in the diagram below:

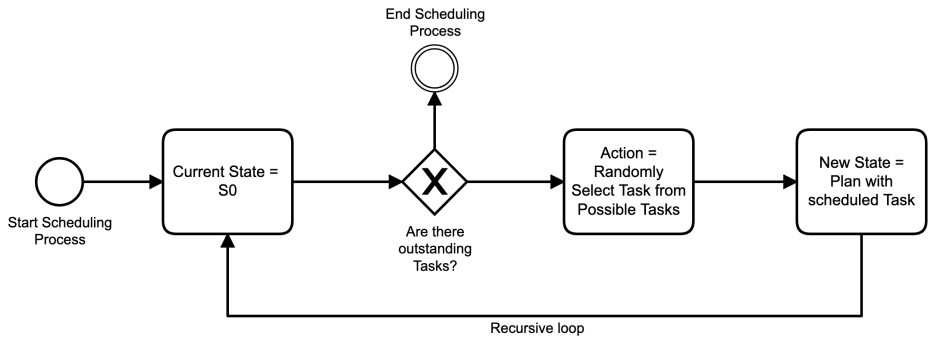

Figure 13 - Random Scheduler algorithm to support rollouts

Monte Carlo Tree Search (MCTS) Scheduler Implementation

The MCTS algorithm is able to probe deeper into possible tree paths that look more rewarding than others by averaging rewards received by all nodes under a given node, whereas rewards are calculated after rollouts to a terminal state. Each MCTS cycle involves four separate steps:

1. Selection of leaf nodes, defined as nodes for which successor states have not been added to the tree yet (unless they are terminal states). Selection is performed in a way to balance exploitation (testing nodes that look more promising) and exploration (testing or expanding unvisited nodes) using the Upper Confidence Bound (UCT) applied to trees.
2. Expansion of selected leaf node if it has already been sampled/visited, otherwise proceed to rollout or simulation
3. Simulation with random rollouts from the selected node until a terminal state is reached and rewards can be assigned
4. Backup or back-propagation of rewards all the way up to the MCTS root node (starting node)

After a predefined number of cycles or any other resource is exhausted (i.e. time limit), a node selection from the root node is made, and the process continues until a terminal state is reached.

The diagram below shows the implemented algorithm for MCTS with UCT decisions:

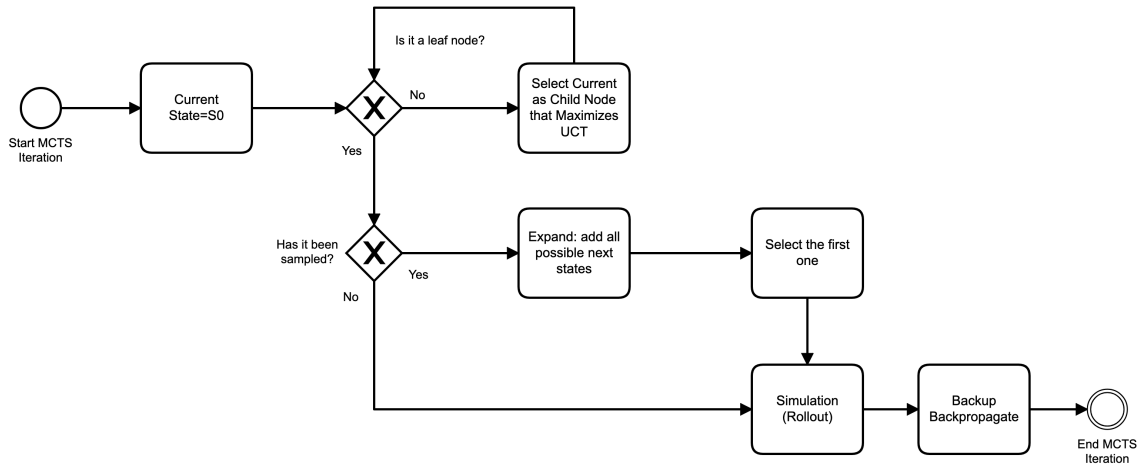

Figure 14 - Monte Carlo Tree Search Algorithm with UCT-based decisions

Traversing the tree starts from the root state node, given by the state of the scheduling plan at the moment. MCTS allows a recursively navigation on branches that seem more rewarding, following the average returns of simulated rollouts from all children nodes. However, a simple greedy selection of nodes with best average returns will potentially "blind" the algorithm into exploiting these branches, so some balance between exploitation of nodes with good returns and exploration of unvisited branches is required.

A method to achieve such balance was proposed by Kocsis and Szepesvári [2] as a derivation of the multi-armed bandit UCB1 decision process, called Upper Confidence Bound applied to trees as shown below:

$$UCT(s_i) = \frac{v_i}{n_i} + c\sqrt{\frac{ln(N)}{n_i}}$$

Where $UCT(s_i) = \infty$ if n_i is zero (if the node has not been visited) and v_i is the accumulated reward back-propagated from all children nodes.

The c parameter is used to control the balance between exploitation and exploration, and is usually set to within 1.5 to 2 for binary outcome simulations such as win/lose games, lacking dimension as on such cases v_i/n_i represents the average winning outcome, which has no dimension.

However, on our case v_i/n_i represents the average scheduling plan duration, which has not only dimension but also a varying magnitude (time units on a scheduling case). This requires constant tuning of the c parameter, which is undesirable. Lubosch et al [3] proposed a machine-learning approach to finely tuning that parameter which simplifies the problem.

Node Selection Pruning

In order to reduce the number of possible Action/States for a given state, we implemented a simple pruning filter based on the earliest possible date that a given task is available to be scheduled. From all tasks that can be scheduled at any given state (no predecessor dependencies), the earliest possible start date is computed. Then only the subset of tasks that may be scheduled between that date and a maximum horizon date are returned. The important implication of pruning this way is that tasks that are far into the horizon will not play a major role on more immediate scheduling decisions, so a “rolling horizon” can be applied to the planning process, effectively limiting the computing resources required to achieve acceptable results with high scalability, as planning time will increase only linearly for tasks beyond the horizon, as opposed to increasing exponentially (given it is an NP-hard problem after all) for tasks within a scheduling horizon.

Experimental Results

Although we tested the application with multiple sets of orders, we chose a representative set with 14 orders and 48 tasks to run simulated plans on.

Results from a random scheduler, which computes a feasible scheduling plan by randomly selecting actions within the scheduling policy (only valid tasks that are available for scheduling) returns mostly suboptimal plans as shown in Figure 15. However, these randomized plans are useful as they are the base of rollouts used on MCTS cycles.

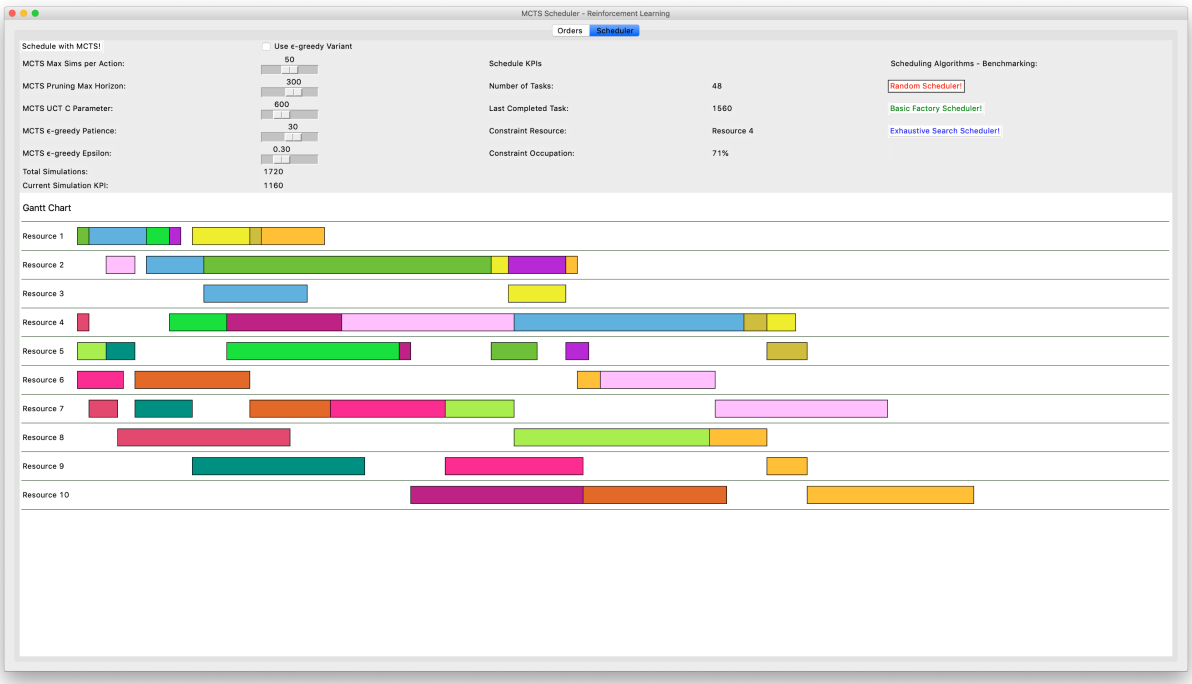

Figure 15 - Sample scheduling plan results from a Random Scheduler

The application shows the occupation of the constraint resource relative to the plan duration. As seen in Figure 15, that particular random simulation achieved a constraint occupation rate of 71%.

Results from the basic scheduler, selecting on each state tasks that can be scheduled at the earliest date are shown in Figure 16, with improved albeit still sub-optimal results, achieving 78% of constraint occupation.

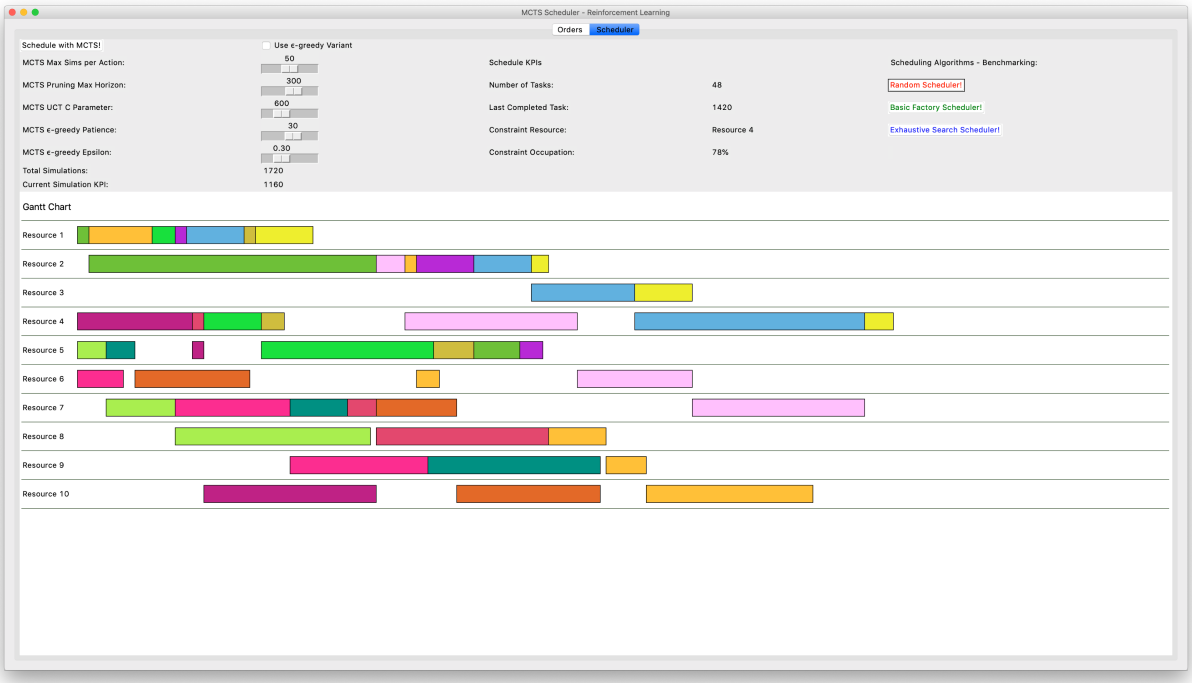

Figure 16 - Sample scheduling plan results from a Basic Scheduler

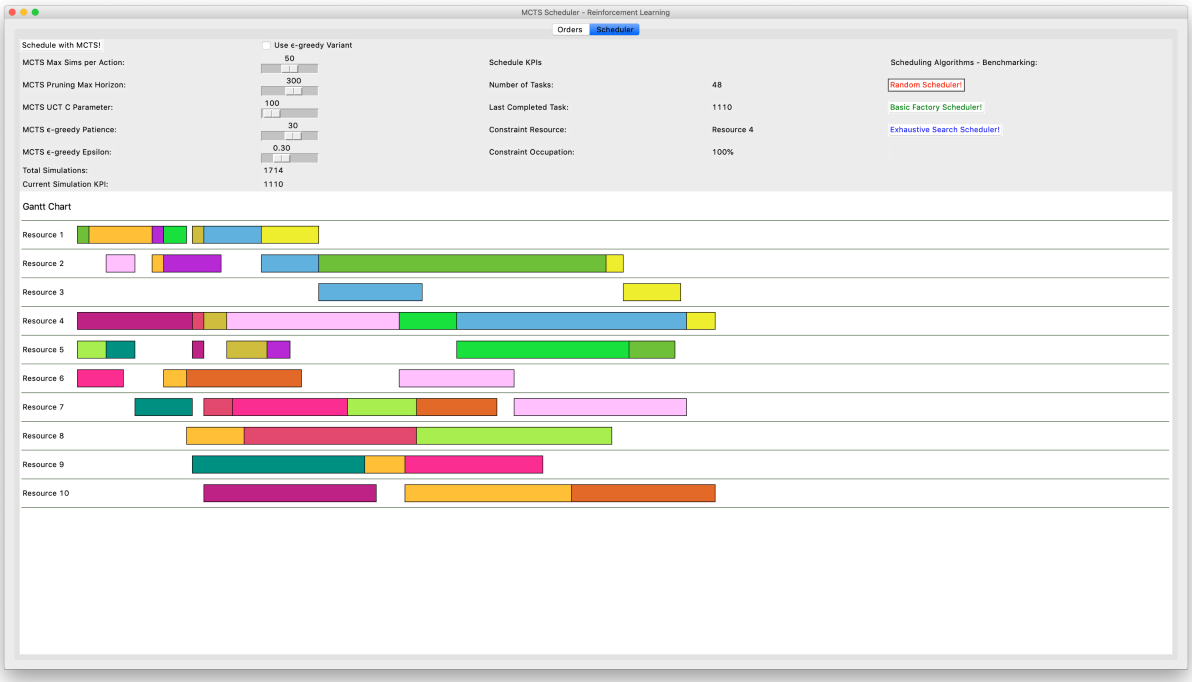

Figure 17 - Sample scheduling plan results using the MCTS Scheduler with UCT decisions, leading to an optimized solution

Using MCTS with UCT decisions with tuned parameters returns optimized plans with quite acceptable (>95%) success rates, as shown on the computed plan in Figure 17, using a maximum of 50 simulations per action, a pruning horizon of 300 time units and a c parameter of 100, which required 1,714 simulations.

For practical purposes this is an optimized plan given the constraint resource (Resource 4) is fully utilized during the entire plan duration, which lasts for 1110 time units. No plan can be computed with a shorter duration.

Tuning MCTS Parameters

Being stochastic in nature, MCTS is not guaranteed to return optimal results, although given enough simulations per cycle it will converge to optimal results at the cost of increased computational resources. As a simple benchmark, the plan on Figure 17 took 44 seconds to compute on a single i5 CPU. Memory consumption is irrelevant on this case.

The most important parameter to tune, besides the maximum number of simulations per cycle, which directly affects processing time and the likelihood of getting near-optimal results, is the exploitation/ exploration parameter c. Exploitation of promising branches is emphasized when c is small and exploration of less-visited branches when c is larger.

A series of full schedules were performed to investigate sensitivity to these parameters, with results compiled and shown in Figure 18 below with minimum and maximum finishing times (optimal finishing time on this example is 1110 units), as well as mean values and the blue box boundaries representing 1st and 3rd quartiles.

Legends represent the number of maximum simulations, the pruning parameter in time units and the c parameter, respectively.

For additional reference, performance metrics for the Random Scheduler and the Basic Scheduler (fixed performance) are also displayed.

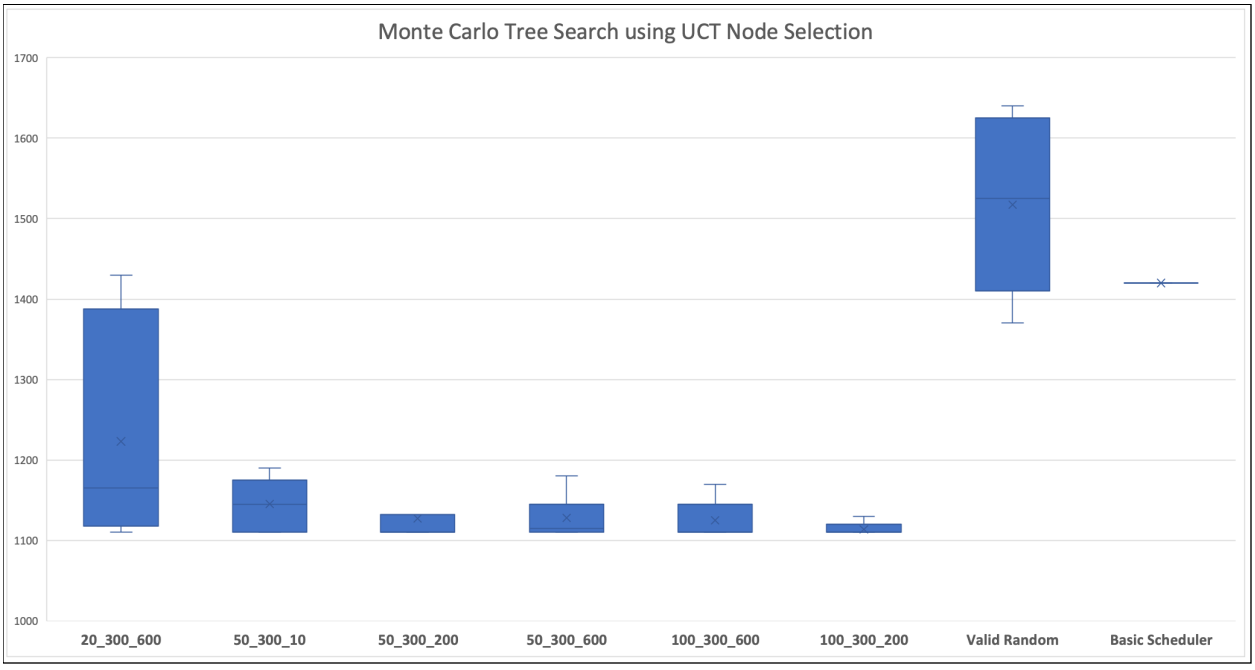

Figure 18 - MCTS with UCT performance for various configuration parameters, compared to Random and Basic schedulers. Low values are desired

Best results using MCTS were achieved with 100 simulations per action, using 300 time units as the horizon pruning parameter and 200 time units as a c parameter.

For a typical run, this setup requires an average of 86 seconds and 3,448 simulations (rollouts), yielding optimal results on 80% of runs with low dispersion (worst run is still acceptable).

Although successfully implemented and yielding good results, MCTS requires many simulations before converging, with the underlying assumption that branches that are likely to lead to good results are found with random rollouts. However, there is a major difference between gaming and scheduling that can be leveraged: any simulated schedule can actually be used as a valid plan, whereas in gaming decisions the same assumption cannot be made, especially on two-player games. This means that on decision cycles, even though average returns can be used to decide between more exploitation or further exploration, the best simulation result can still be used to select the next action, even if it is different that the action suggested by the node with best overall average.

MCTS Variant with ε-greedy Policy

In order to explore that concept, a variant of MCTS algorithm was implemented on the application as an efficient way to navigate the state tree, but using an ε-greedy policy to control the exploitation/exploration balance. Using ε-greedy decisions overcomes the need to finely tune the c parameter for UCT decisions.

Besides, selecting actions based on the best simulation so far, instead of actions with best average returns, which is feasible given the nature of this problem, significantly speeds up processing time.

Another improvement to the algorithm is implementation of a Patience parameter. During any node selection cycle, a node can be selected if no further improvements on the best outcome have been found after a number of iterations defined by the Patience parameter. This considerably speeds up processing whilst leveraging exploration/exploitation gains until they converge to a best outcome.

The implemented algorithm for this method is shown in Figure 19 below:

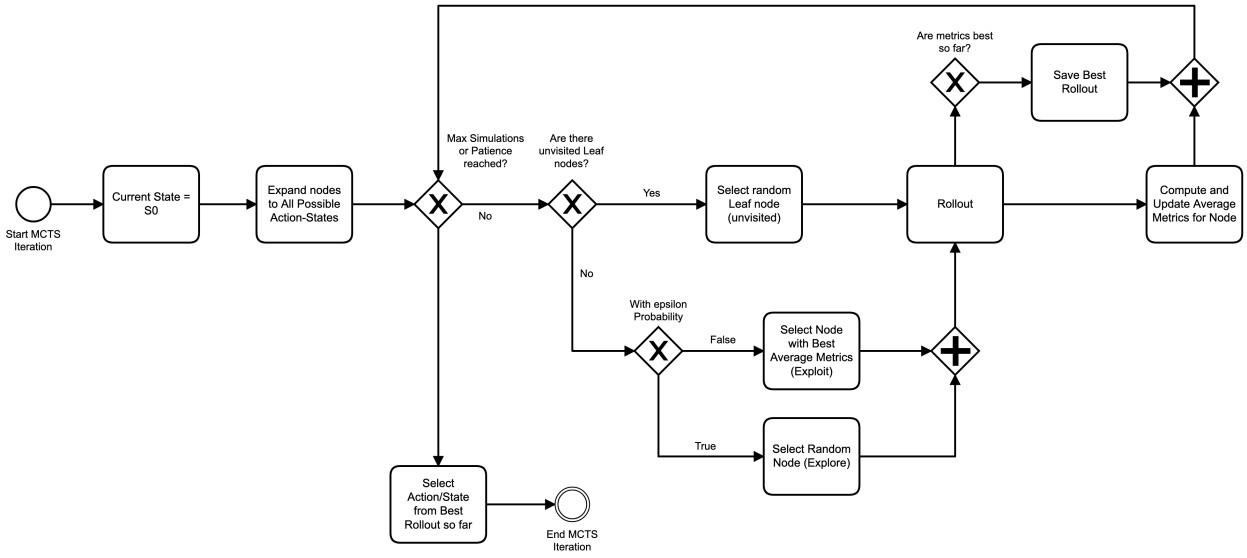

Figure 19 - MCTS with ε-greedy Decisions as an alternative to the UCT method, as implemented

This algorithm also has parameters to tune, including maximum number of simulations per Action, maximum horizon, Patience and ε. Experimental results are shown in Figure 20 below after initial tuning:

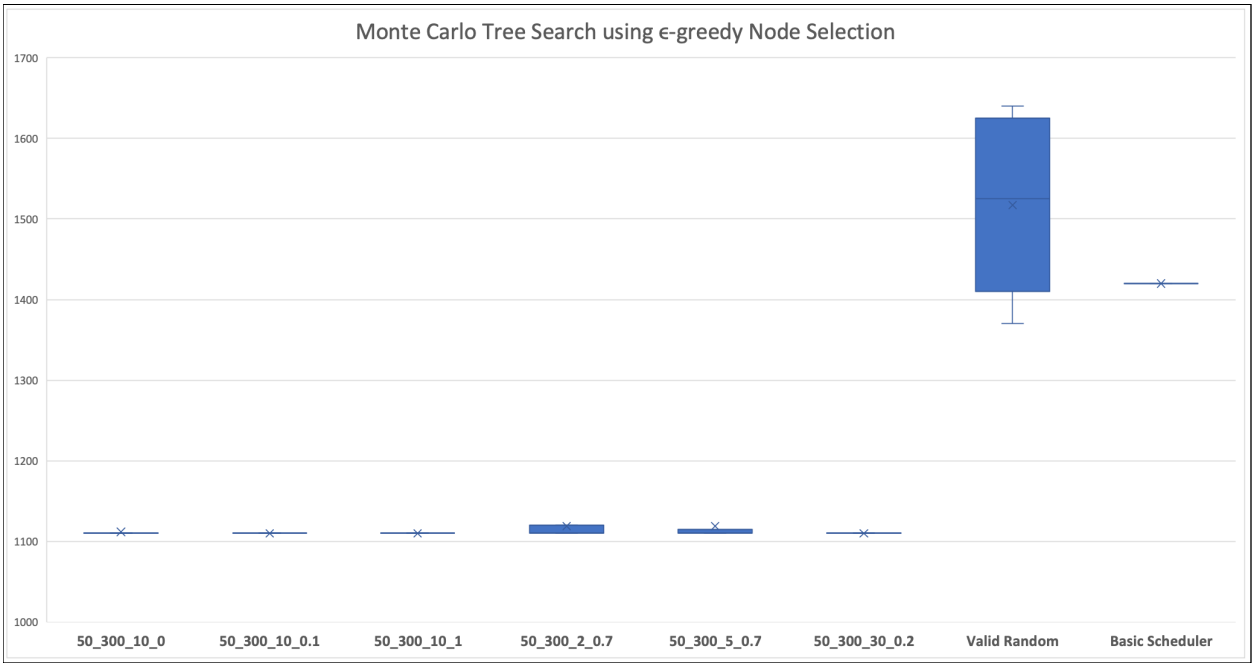

Figure 20 - MCTS with ε-greedy Decisions experimental results showing enhanced performance and low variability

In comparison to MCTS with UCT, this algorithm provides enhanced results at significantly lower computational requirements, as it continuously leverages best possible plans as the state tree during tree traversal, whilst still keeping options open to explore yet potential better branches.

As a direct comparison, the best MCT with ε-greedy configuration, using a maximum of 50 simulations per Action, 300 as the horizon pruning parameter, 10 cycles of patience and 0.1 as ε parameter, yielded optimal results on 96% of runs, taking on average 13 seconds and only 550 simulations. This is surprisingly faster and more efficient than MCTS with UCT applied to this specific problem.

Conclusions and Future Work

MCTS can be successfully applied to the industrial scheduling problem, especially if modified to leverage the fact that any simulation is not a probabilistic inference, but rather a real possibility. Our MCTS implementation resets the search tree upon state changes, so there is still a minor improvement to implement, saving and repurposing nodes that have been sampled. It will not affect performance in a drastic way considering most of the MCTS tree is discarded anyway once a node is selected.

A possible next step is leveraging optimized scheduling plans to train a deep learning model and derive a Q function that in turn will speed up decisions on MCTS, in the same fashion as Alpha-Go implementations. However, given that the definition of a current state depends on the environment (i.e. configuration of resources, scheduled tasks and outstanding tasks), the DL model needs customization for each environment, so it will be subject of future work.

Code Base

The application code specifically developed for this project as well as supporting configuration/production order files is available for download from this Github repository:

https://github.com/schrappe/mctsscheduler

References

1. Yu. N. Sotskov, V. Shakhlevich. NP-hardness of shop-scheduling problems with three jobs. Discrete Applied Mathematics, Volume 59, Issue 3, 26 May 1995, Pages 237-266
2. Levente Kocsis and Csaba Szepesvári, Bandit based Monte-Carlo Planning. European Conference on Machine Learning, ECML 2006 pp 282-293
3. Marco Lubosch, Martin Kunath, Herwig Winkler. Industrial scheduling with Monte Carlo tree search and machine learning. 51st CIRP Conference on Manufacturing Systems
4. Bernd Waschneck, André Reichstaller, Lenz Belzner, Thomas Altenmüller, Thomas Bauernhansl, Alexander Knapp, Andreas Kyek. Optimization of global production scheduling with deep reinforcement learning. 51st CIRP Conference on Manufacturing Systems
5. Frank Benda, Roland Braune, Karl F. Doerner, Richard F. Hartl. A machine learning approach for flow shop scheduling problems with alternative resources, sequence-dependent setup times, and blocking. OR Spectrum (2019) 41:871–893. https://doi.org/10.1007/s00291-019-00567-8
6. Chandrasekhar V. Ganduri. Rule Driven Job-Shop Scheduling Derived from Neural Networks Through Extraction. 2004 Thesis presented at College of Engineering and Technology of Ohio University
7. Yeou-Ren Shiue, Ken-Chuan Lee, Chao-Ton Su. Real-time scheduling for a smart factory using a reinforcement learning approach. https://doi.org/10.1016/j.cie.2018.03.039
8. Chaslot, Guillaume, de Jong, Steven, Takeshi-Saito, Jahn, Uiterwijk, Jos. Monte-Carlo Tree Search in Production Management Problems.
9. Tara Elizabeth, Thomas Jinkyu Koo, Somali Chaterji, Saurabh Bagchi. MINERVA: A Reinforcement Learning-based Technique for Optimal Scheduling and Bottleneck Detection in Distributed Factory Operations. School of Electrical and Computer Engineering - Purdue University
10. Sutton, Richard S., Barto, Andrew G. Reinforcement Learning - An Introduction. 2nd Edition. The MIT Press.
11. Pumperla, Max, Ferguson, Kevin. Deep Learning and the Game of Go. Manning.