Observation Sensitive MCTS for Elevator Transportation - Milestone Report

Maximilian Rieger
max.rieger@tum.de

Tim Pfeifle
tim.pfeifle@tum.de

***Abstract*—We modified the Monte-Carlo-Tree-Search (MCTS) approach used in AlphaZero to incorporate observed rewards in every step, additionally to the value estimation of a neural network. We apply the modified algorithm to the task of elevator transportation as one representative of problems with rewards at every step that feed directly into the final reward.**

I. Introduction

Before evaluating our modifications for MCTS we implemented the elevator transportation environment. We choose this task as a representative of the family of tasks, that give useful rewards before the end of episodes. By modifying MCTS we could then apply the AlphaZero algorithm to a wide range of other tasks in this family. In the case of the elevator transportation where we optimize for minimal passenger-waiting-times, those step-wise rewards are the number of seconds a passenger waited, before arriving at the requested target.

II. Environment Implementation

Our environment consists of the following components:

House: We have one house, which contains a list of elevators, each with their own state. The house handles the elevator-requests (up/down) at each floor.

Elevator: This entity contains a set of passengers and the requested floors by this set. They have three possible actions *Up, Down, Up*. Elevators also have a maximum capacity and a time indicating when the current action is done.

Time: To choose the next action, the environment always selects the elevator with the lowest time, as this the one, which needs new control instructions first. We consider this lowest time as the time of the house. The time of the house elapses, when after controlling one elevator the new lowest time is higher. New elevator-requests are created by the PassengerGenerator when time elapses. An episode spans one day of elevator-requests.

Passenger: Passengers are only instantiated once they enter an elevator. Each passenger has a distribution over target floors on which he might leave. This makes sure, that no passenger leaves at a floor, for which no request exists and that all passengers have left the elevator after it stopped at all requested floors.

PassengerGenerator: The easiest way would be to generate passengers in each time step and let them have a fixed target floor and arrival time and just hide this information from the agent to simulate this hidden state space. In this project however we want to use a variant of the AlphaZero [1] algorithm, which uses simulation of the environment to search for good actions. If simulations are used with this hidden state space, information like the count of passengers waiting at a floor, or the count of passengers wanting to leave at a certain floor could leak through the observations of future time steps. We avoid this problem by sampling these values randomly at the time they are needed. This way, the agent can observe their distributions but cannot determine any exact values.

1) Generating Requests: When a house elapses time t, the PassengerGenerator samples new requests based on request rates for each floor. We sample the count of passengers appearing at each floor from a Poisson distribution. Then we sample a target floor for each passenger according to a predefined Categorical distribution over targets to decide whether the passenger presses the up or down button. We also sample an arrival time for each passenger. Having the arrival times and directions of each passenger, we can update the requests stored in our house entity.

2) Generating Entering Passengers: When an elevator opens at a floor at which a request signal is set, the PassengerGenerator generates actual passengers, which then can enter. Exact arrival times are computed by sampling the time between two passengers arriving from an Exponential distribution, so the count of arrived passengers will be Poisson distributed. The first arrival time is given by the timestamp of the requests. We also sample target floor distributions for each passenger, which are used to decide whether the passenger leaves the elevator when it stops at a certain floor.

III. Observation Sensitive MCTS

AlphaZero's version of MCTS considers observed rewards only at the end of episodes, which makes perfectly sense for games like go, where rewards only occur when the game ends. In contrast, we incorporate rewards observed while descending the search tree in the action-value. This helps to guide MCTS exploration especially at the beginning of episodes. In contrast to the function proposed in our project proposal we normalize the observed rewards not by the length of the action path, but rather by the time elapsed following the path (see Equation 1).

$$Q_{new}(s,a) = \frac{1}{N(s,a)} \sum_{s'} c_{obs} \cdot f_{norm}\left(\frac{r(\pi_{s,s'})}{\delta_{time}(\pi_{s,s'})}\right) + (1 - c_{obs}) \cdot v(s') \quad (1)$$

We have implemented our version of MCTS and for our first experiments we have mocked the neural network with a uniform-model to validate that our MCTS implementation works (see Fig. 1). The uniform-model outputs a constant value-prediction of 0 and a uniform policy over all valid actions.

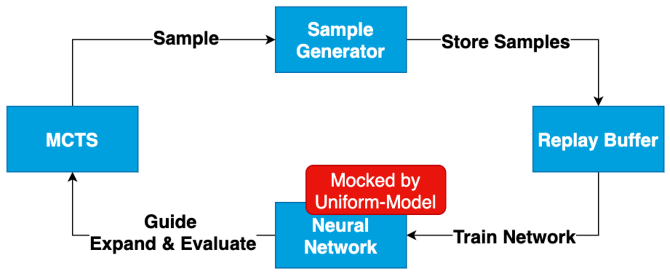

Fig. 1. Overview of the algorithm

IV. Experiments

Using our uniform model as a mock for the neural network, we evaluated our MCTS implementation by creating episodes following the MCTS policy. We used a low value (0.01) as the temperature parameter for this, which makes it very likely, that the action with the highest probability is chosen. As you can see in Figure 2 the MCTS policy is much better than random for houses with five or less floors. When looking closer at generated episodes for houses with three floors, we saw that the MCTS policy was acting mostly optimal.

With an increasing number of floors, the problem becomes a lot harder, as elevators are less likely to be at the correct floor by chance. Also reaching target floors takes much longer, which increases the required search depth. Hence it is expected that the MCTS performance degrades. Strangely, the performance did not only degrade, but became worse than the random policy. As we use squared waiting times, the linear normalization over time in Equation 1 is outweighed by the quadratic waiting times. We suspect that this causes the MCTS to favor short actions such as moving up or down over opening the doors at all. To confirm this suspicion, we looked at the generated actions and saw that those were dominated by the short actions Up and Down. Therefore passengers were simply never picked up. After switching to equal action times for all actions the observation sensitive MCTS outperformed Random Policy clearly (see Fig. 3).

V. Next Steps

Our first step will be to decide whether we keep equal times for all actions, or if it is possible to circumvent the domination by short-actions. We will then replace our uniform-model with a Neural-Network and train it with the MCTS samples from our replay buffer (see Fig. 1). To stabilize training and to help optimize even small improvements we plan to use ranked reward [2]. After training we will evaluate the performance against classic heuristic elevator algorithms.

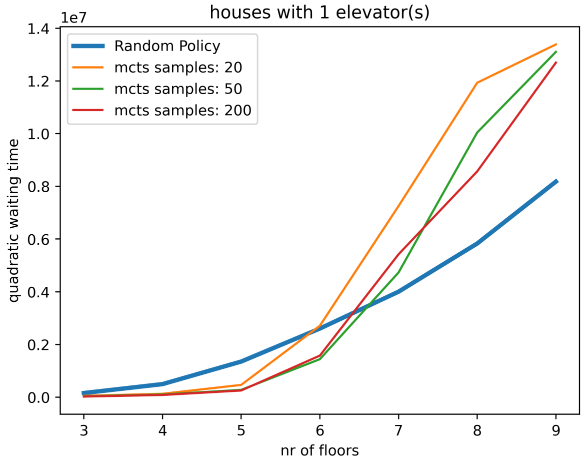

Fig. 2. MCTS results for houses with different amounts of floors (presentation version)

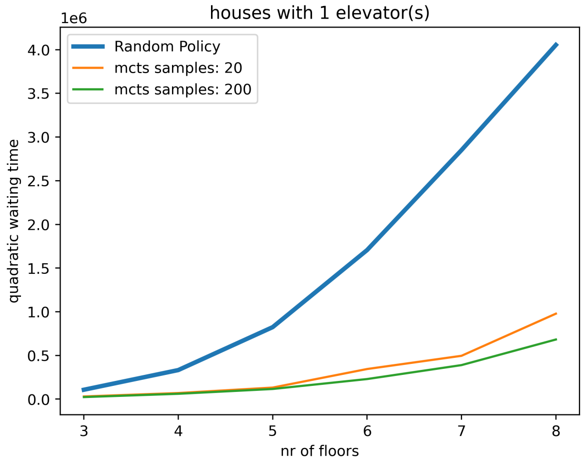

Fig. 3. MCTS results using the same time for all actions

References

[1] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, et al. “A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play”. In: *Science* 362.6419 (2018), pp. 1140–1144.

[2] A. Laterre, Y. Fu, M. K. Jabri, A.-S. Cohen, D. Kas, K. Hajjar, T. S. Dahl, A. Kerkeni, and K. Beguir. “Ranked reward: Enabling self-play reinforcement learning for combinatorial optimization”. In: *arXiv preprint arXiv:1807.01672* (2018).