MCTS for Universal Robot End Effector Positioning

Keona D'Souza
Ann and H.J. Smead Aerospace Engineering Sciences
University of Colorado Boulder
Boulder, Colorado, USA
keona.dsouza@colorado.edu

***Abstract*—Monte Carlo Tree Search (MCTS) is a popular searching algorithm that has been used with great success to play various board games and video games. This paper explores the use of MCTS for the application of positioning the end effector of a commonly used robot, the Universal Robot's UR10 robotic arm to reach a targeted position. The UR10 has six degrees of freedom and is capable of reaching distances of approximately a meter across. In order to evaluate the performance of the MCTS algorithm on an environment with such a large, continuous state and action space, the problem is formulated as a Markov Decision Process(MDP). Benchmark testing and Monte Carlo simulations of various implementations of the MCTS algorithm were performed. The environment was simulated through the use of OpenAI's Gym and other open source modules such as robo-gym, which provided a platform on which to test the capability of the algorithm to solve the problem in a realistic environment.**

***Index Terms*—MCTS, UR10, MDP, OpenAI, robo-gym**

I. Introduction

Robotic arms such as Universal Robot's UR10 are being deployed to numerous different workplaces to perform tasks such as automated welding, pick-and-place, palletizing, and others. At the core of all these tasks is the most basic functionality: moving the end-effector of the robotic arm to a targeted position. A classical solution for solving this task is using the known kinematics of the robot to generate paths in the environment. This solution, however, does not take into consideration the complications caused by human-robot interactions, which could have a large affect on the environment and potentially decrease the productivity of the robot and the workflow. Techniques such as Reinforcement Learning(RL) would make it possible to generate increasingly optimal, stable, and robust trajectories that allow the robots to perform in highly dynamic environments [8]

II. Background and Related Work

A. Motion Planning for Robotic Arms

Motion planning for complex manipulators such as robotic arms can be performed many different ways. Methods for generating trajectories can be as simple as picking points along the trajectory, then connecting the points using linear interpolation or splines. This requires some knowledge of where these points should land along the trajectory, and is generally done manually by the operators of the robot. Other methods include finding trajectories using the inverse kinematics of the robot, which allows one to find the necessary joint angles required for the end effector to be in a certain position. This solution is ideal for simple robots, but as the degrees of freedom begin increasing as does the complexity of solving the inverse kinematics problem. For robots with inverse kinematics that cannot be solved analytically, numerical methods can be used to generate solutions. Trajectories generated using inverse kinematics are widely, but they do not consider the environment around the robot as a factor. Algorithms that use inverse kinematics will need to supplement the inverse kinematics with other tools to perform obstacle avoidance or collision.

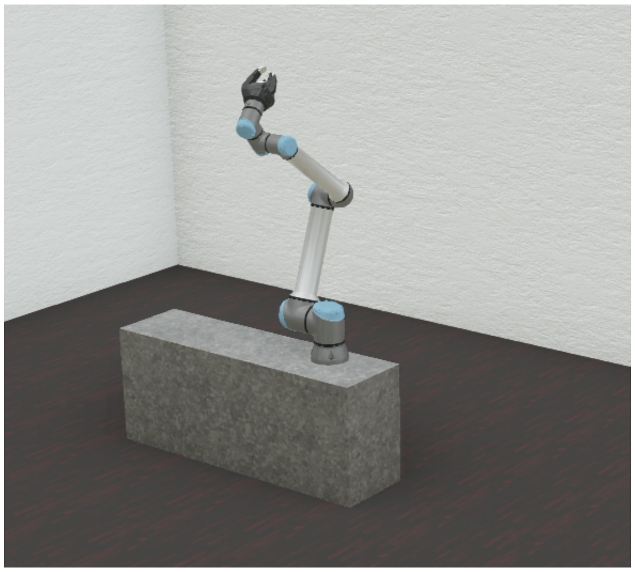


Fig. 1. UR10 Robot in simulated environment

Sampling-based motion planning algorithms are also widely used and implemented in the field of motion planning for mobile manipulators [15]. The Rapidly-exploring Random Tree(RRT) algorithm, along with it's many variants such as RRT*, is one such sampling-based algorithm that takes random samples in unexplored regions of the configuration space of the environment and creates a tree between samples. Once the space has been fully explored and the tree generated, trajectories can be generated by following nodes of the tree. Another popular algorithm is Probabilistic Roadmap(PRM), which is largely similar to RRT but will randomly sample the entire configuration space, not just the unexplored regions. The tree connects all valid samples in the space, and trajectories can then be found by searching along the tree. Often algorithms such as Dijkstra's's or A* are used to traverse the tree to find shortest paths.

A field growing in popularity for the use of mobile manipulator motion planning is reinforcement learning, which can allow robots to perform better in complex, dynamic systems such as ones with human-robot interactions [12]. Algorithms such as the ones mentioned above have the goal of generating trajectories to reach a desired tradition. Using RL, the robots can interact with the environment and be driven by performing tasks and gaining rewards, rather than simply following a path. Algorithms can be easily be trained on simulated environments before being executed on a real scenario, which allows for quicker development cycles and better performance [7].

B. Markov Decision Process

One of the fundamental components of reinforcement learning is the Markov Decision Process(MDP). MDPs are a method of defining and bounding a problem by it's states, actions, transition probabilities, and rewards [16]. Once the MDP has been formulated for an agent in an environment, the objective is to find the action that will maximize the possible rewards at that state. MDPs are commonly used to define problems such as board games, video games, economic scenarios, and many others. Robotics is one of the many applications that utilizes MDPs to perform tasks such as motion planning.
MDPs are defined by the following pieces of information:

1 S: The state space of the problem, or the available states the agent can be in.
2 A: The action space of the problem, or all available actions for the robot to take.
3 $T(s'|s,a)$: The probability distribution of transitioning from the current state s to the next state s' if action a is taken.
4 $R(s,a)$: The reward that can be obtained by taking action a while in state s.
5 γ: The discount factor, which allows for infinite horizon problems to be solved.

C. Monte Carlo Tree Search

Monte Carlo Tree Search, or MCTS, is a used commonly in MDPs to plan policies for an agent. The algorithm builds a tree through four main steps: node exploration, node expansion, rollout, and backpropogation [17]. First an action is selected using various different methods, such as an Epsilon-greedy policy that balances exploration and exploitation, or a policy based on the Upper Confidence Bound(UCB) exploration heuristic.

$$Q(s,a) + c\sqrt{\frac{logN(s)}{N(s,a)}} \tag{1}$$

The objective is to choose the action that will maximize the above equation. The second step is expansion of the chosen node into child nodes that correspond to the possible actions to be taken. Once the expansions of each node has been performed, a rollout is performed to find the reward. The reward found from the rollout can then be backpropogated up the nodes of the tree. This process is iterated over until a tree has been constructed that holds information on which action to take that will result in the highest reward, based on the current state.

III. Problem Formulation

The problem to be solved is the positioning of a Universal Robot's end effector to a target position through the use of MCTS. The specific robot, the UR10 has 6 degrees of freedom, with six controllable joints. In order to apply the MCTS algorithm, the problem must first be formulated as an MDP. The state space of the robot and the environment is extensive - including the current joint angle positions of each of the six joints, the current joint velocities of each of the six joints, the current position of the end effector, and the current velocity of the end effector. This makes for an extremely large state space, comprised of eighteen distinct states. The robot is controlled via position control of the joints, which means that the available actions include the entire range of possible angles for all six of the joints.
In order to bound the complexity of the problem for initial algorithm implementation, the state space is greatly reduced, and both state and action spaces are discretized with relatively coarse steps.

$$S = grid[EE_x, EE_y, EE_z, B_p, S_p, E_p, W_{1_p}, W_{2_p}, W_{3_p}] \tag{2}$$

, where EE refers to all possible combinations of the allowable range of x, y, z positions of the UR10's end effector and $B_p, S_p, E_p, W_{1_p}, W_{2_p}, W_{3_p}$ refers to the current joint angle positions. The UR10 robot has a reach of approximately 1.3m [10], which, which provides bounds for the position ranges. The joint angles are bound between 0 and 2π The state space is similarly defined:

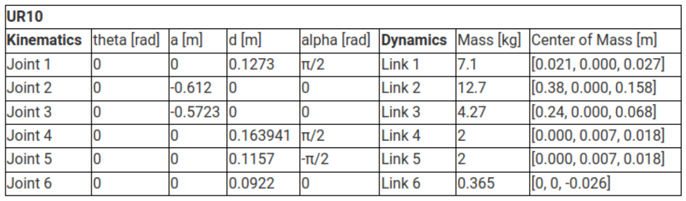
$$A = grid[B, S, E, W_1, W_2, W_3] \tag{3}$$

where B refers to the base joint, S refers to the shoulder joint, E refers to the elbow joint and W_1, W_2, W_3 refer to the three wrist joints, respectively. In order to further reduce the complexity of the problem, each joint has three possible actions: rotate in the counterclockwise direction by a set amount of degrees, rotate in the clockwise direction by a set amount of degrees, or to not rotate at all. The action space then represents a grid of all available joint action possibilities.

Discarding other information such as the joint velocities, as well as the velocity of the end effector allow only the necessary states for the problem to remain. However, the reduction of the state space will cause the policy generated by the algorithm to have degraded performance in simulated and real environments, due to the fact that the velocity and dynamic limits of the robot are not considered.

The transition function takes in the current state and an action, and returns the probability distribution of receiving the next state. In order to calculate the next state given that the action is known, forward kinematics can be used. Forward kinematics of a robot takes in the joint angles and outputs the end effector position. From the state, the current joint angles can be found. The action is then applied to the current joint angles, and the new joint angles are found. Then, the new joint angles are used along with the forward kinematics to find the new end effector position. Finally, the new state is found as a combination of the new end effector position and the new joint angles. The probability distribution is deterministic over this new state, but in reality sources of errors could cause the state to vary slightly from what is expected from the forward kinematics.

The forward kinematics were found using the existing Denavit–Hartenberg(DH) parameters for the UR10 robot [10], shown in Fig 2.



UR10							
Kinematics	theta [rad]	a [m]	d [m]	alpha [rad]	**Dynamics**	Mass [kg]	Center of Mass [m]
Joint 1	0	0	0.1273	π/2	Link 1	7.1	[0.021, 0.000, 0.027]
Joint 2	0	-0.612	0	0	Link 2	12.7	[0.38, 0.000, 0.158]
Joint 3	0	-0.5723	0	0	Link 3	4.27	[0.24, 0.000, 0.068]
Joint 4	0	0	0.163941	π/2	Link 4	2	[0.000, 0.007, 0.018]
Joint 5	0	0	0.1157	-π/2	Link 5	2	[0.000, 0.007, 0.018]
Joint 6	0	0	0.0922	0	Link 6	0.365	[0, 0, -0.026]

Fig. 2. DH Parameters for the UR10 Robot

The forward kinematics of the robot The reward is a function of both the state and the action. In order to calculate the reward, first the next state, s' must be found from the current state s and action a using the forward kinematics discussed above. Once the next state has been found, the euclidean distance between the target position and the next state is calculated using the following formula:

$$d = \sqrt{(t_x - ee_x)^2 + (t_y - ee_y)^2 + (t_z - ee_z)^2} \qquad (4)$$

where t_x, t_y, t_z represents the coordinates of the target and ee_x, ee_y, ee_z represents the coordinates of the end effector's next state, respectively. The reward function is then defined as:

$$R = \begin{cases} 200 & d \leq D_{min} \\ \frac{1}{d} & o.w. \end{cases}$$

where D_{min} is a defined distance threshold for being in the target state, which in this scenario is 0.01m away from the target in any direction. This reward function will produce a high reward if the next state is within the distance threshold. If the distance is not in the threshold, the function will still produce a reward that will punish large distances and reward small distance due to the inverse nature of the function.

The terminal condition of the MDP is calculated using the distance threshold discussed above, where the euclidean distance between the input state of the end effector and the target position is calculated, and checked whether it falls under the minimum distance threshold.

The final component of the MDP is the discount factor, which was chosen to be 0.99.

IV. Implementation

The implementation of this problem was split into two components:

1 Policy generation in Julia, which involved defining the MDP and running the MCTS algorithm to generate a policy to take
2 Simulation in an OpenAI gym environment, which involved using the policy generated from step 1 to perform a simulation.

A. MDP Creation

In order to define the MDP described in the section above, the QuickMDP package from [14] was used. The state space, action space, transition function, and reward function were all defined as detailed above. Although the state space and action space were both dense matrices, the reduction of the spaces discussed earlier allowed the size of the matrices to be manageable.

B. MCTS Algorithm development

The MCTS algorithm was designed and tested in two ways. The first way was using the pre-existing MCTS algorithm found from [14] to act as a baseline. The algorithm was trained with various different parameters, including the number of iterations, the depth of the tree, and the exploration constant. Ranges of number of iterations included 20, 100, 1000, and 10,000. Tree depths were tested at size 5, 10, and 20. The exploration constant was set to values of 1.0, 10.0, and 200.0. Monte Carlo analysis of the existing MCTS algorithm was also performed over 1000 runs of the MCTS algorithm.

Once the pre-exisiting MCTS algorithm was implemented and tested, a custom implementation of MCTS was developed. This algorithm was very similar to the one developed for the course. Different heuristic exploration policies were tested, including the UCB heuristic, using an epsilon-greedy policy, and using a custom heuristic policy that would find the action that minimized the distance between the current state and the target position. The performance of this algorithm was also benchmarked over a the same range of hyper parameters that was tested with the baseline MCTS algorithm. The performance of both algorithms, as well as visualizations of the trees will be displayed in the results section.

C. The robo-gym Environment

The simulation environment chosen was an open source extension of the commonly used OpenAI Gym environment, robo-gym [4] . The environment provided the full state space of the environment, including the joint angles and velocities, end effector position and velocity, target position and velocity, as well as transformations to put the end effector into the target frame. The action consisted of the six normalized joint angles. The environment provided a method of resetting the environment, as well as performing one step in the environment given an input action. The environment is available as a Python interface, so it was necessary to import the Julia module in Python to be able to use the policy developed in

Julia. Once the policy was developed and imported into the Python environment, a simulation was performed for a number of episodes.

V. Results

A. Average Rewards and Computation Time

Monte Carlo simulations were performed using a basic rollout to obtain the average rewards found from the MCTS algorithm. The hyper parameters chosen were a number of iterations equal to 10,000, an exploration constant of 200.0, and a tree depth of 20.

For the existing MCTS implementation, the average reward over 1000 runs was found to be 134.05, with each iteration taking approximately 16 seconds to run. The custom MCTS implementation using the epsilon-greedy exploration policy performed worse than the baseline, generating an average reward of 36.0 over 1000 runs. This policy took approximately 20 seconds to run through each rollout. Using the heuristic policy improved the average reward significantly, generated average rewards of 206.0 over 100 runs, and took approximately 11 seconds per rollout. This shows that the heuristic policy was consistently able to reach the terminal state of the end effector reaching the goal.

The computation time of all MCTS algorithm implementation at high iteration numbers is not feasible for actual implementation, and significant work would need to be done to improve the speeds for more complex state and action spaces.

B. Tree Visualization

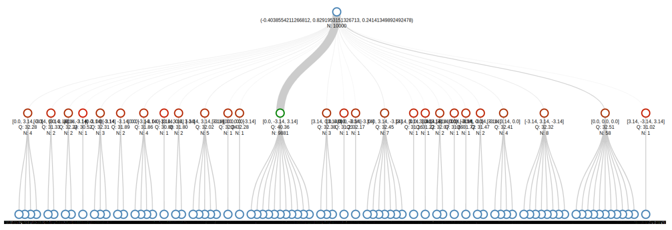


Fig. 3. MCTS Tree using existing baseline algorithm

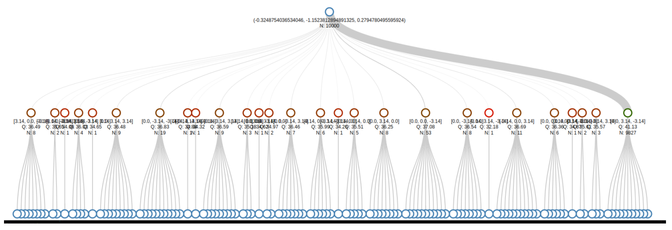


Fig. 4. MCTS Tree using custom algorithm and UCB heuristic policy

VI. Conclusion

In conclusion, the MCTS algorithm is a powerful tool for solving MDPs. Using certain hyper parameters, a heuristic exploration policy, and for a very reduced state and action space, it was able to find the target position and receive a high reward consistently. The advantage of MCTS is that it is an extremely straightforward algorithm to implement. However, the successful performance of the policy during the algorithm development did not translate to success in the simulation environment. This is likely due to the reduced state and action space, making the MDP inaccurate to the actual simulation. In order to fully represent the environment, the MDP would need to include the entire state space. Implementing such a large space is virtually impossible using MCTS, as the computational complexity and time required is infeasible. In the future, trying out different variations of MCTS, such as Double Progressing Widening (DPW) as well as trying out different Actor Critic Algorithms would likely produce more robust policies. In addition, utilizing machine leaning techniques such as deep learning and neural networks are a possible solution for solving this problem.

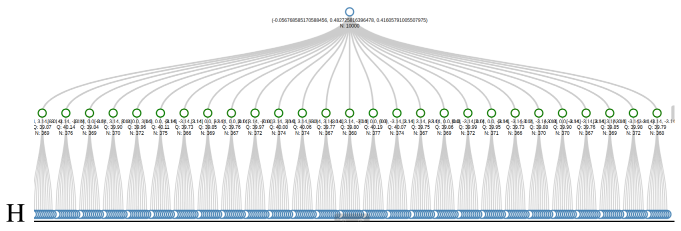


Fig. 5. MCTS Tree using custom algorithm and epsilon-greedy policy

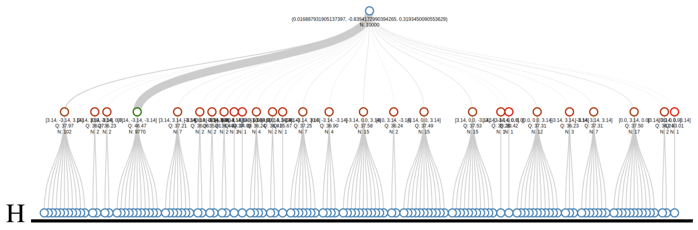


Fig. 6. MCTS Tree using custom algorithm and heuristic policy

Contribution and Release

The author is the sole contributor to the work presented in this paper. The author would like to acknowledge the use of existing code for the simulation environment, found at the link https://github.com/jr-robotics/robo-gym. The algorithms referenced in this paper were either developed from scratch with Python and Julia, using the references listed below, or were sourced from https://github.com/JuliaPOMDP/POMDPs.jl. The author grants permission for this report to be posted publicly.

References

[1] Malik, Aryslan, et al. "A Deep Reinforcement-Learning Approach for Inverse Kinematics Solution of a High Degree of Freedom Robotic Manipulator." Robotics, vol. 11, no. 2, 2 Apr. 2022, p. 44, https://doi.org/10.3390/robotics11020044. Accessed 12 Sept. 2022.

[2] Li, Kun, and Ke Wang. "Robot Arm Simulation Based on Model-Free Reinforcement Learning." 2021 IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA), 28 June 2021, https://doi.org/10.1109/icaica52286.2021.9498063. Accessed 8 May 2024.

[3] Sami Sellami, and Alexandr Klimchik. “A Deep Learning Based Robot Positioning Error Compensation.” 2021 International Conference “Nonlinearity, Information and Robotics” (NIR), 26 Aug. 2021, https://doi.org/10.1109/nir52917.2021.9666097. Accessed 1 Aug. 2023.https://ieeexplore.ieee.org/abstract/document/9666097

[4] Lucchi, Matteo, et al. “Robo-Gym – an Open Source Toolkit for Distributed Deep Reinforcement Learning on Real and Simulated Robots.” ArXiv (Cornell University), 24 Oct. 2020, https://doi.org/10.1109/iros45743.2020.9340956. Accessed 8 May 2024.

[5] Hawkins, Kelsey P. Analytic Inverse Kinematics for the Universal Robots UR-5/UR-10 Arms. 7 Dec. 2013.

[6] Fouad Bahrpeyma, Abishek Sunilkumar, and D. Reichelt, “Application of Reinforcement Learning to UR10 Positioning for Prioritized Multi-Step Inspection in NVIDIA Omniverse,” 2023 IEEE Symposium on Industrial Electronics & Applications (ISIEA), Jul. 2023, doi: https://doi.org/10.1109/isiea58478.2023.10212317.

[7] L. Eisinger, ”Deep Learning Based Motion Planning for Joint Position-controlled Robotic Manipulators,” Master’s thesis, Eindhoven University of Technology, Eindhoven, Netherlands, 2020. [Online]. Available: https://pure.tue.nl/ws/portalfiles/portal/205413496/1449273_Eisinger.pdf

[8] L. Leyendecker, M. Schmitz, Hans Aoyang Zhou, Vladimir Samsonov, Marius Rittstieg, and D. Lutticke, “Deep Reinforcement Learning for Robotic Control in High-Dexterity Assembly Tasks - A Reward Curriculum Approach,” International Journal of Semantic Computing, vol. Vol. 16, No. 03, pp. 381-402, no. 2022, Nov. 2021, doi: https://doi.org/10.1109/irc52146.2021.00012.

[9] P. Kurrek, M. Jocas, Firas Zoghlami, M. F. Stoelen, and V. Salehi, “Ai Motion Control – A Generic Approach to Develop Control Policies for Robotic Manipulation Tasks,” Proceedings of the ... International Conference on Engineering Design, vol. 1, no. 1, pp. 3561–3570, Jul. 2019, doi: https://doi.org/10.1017/dsi.2019.363.

[10] ”DH PARAMETERS FOR CALCULATIONS OF KINEMATICS AND DYNAMICS,” Universal Robots, [Online]. Available: https://www.universal-robots.com/articles/ur/application-installation/dh-parameters-for-calculations-of-kinematics-and-dynamics/

[11] Alberto Dalla Libera, Giulio Giacomuzzo, R. Carli, D. Nikovski, and D. Romeres, “Forward Dynamics Estimation from Data-Driven Inverse Dynamics Learning,” IFAC-PapersOnLine, vol. 56, no. 2, pp. 519–524, Jan. 2023, doi: https://doi.org/10.48550/arXiv.2307.05093

[12] N. M. Gomes, F. N. Martins, J. Lima, and H. Wörtche, “Reinforcement Learning for Collaborative Robots Pick-and-Place Applications: A Case Study,” Automation, vol. 3, no. 1, pp. 223–241, Mar. 2022, doi: https://doi.org/10.3390/automation3010011.

[13] Liu, S., Liu, P. (2021). A Review of Motion Planning Algorithms for Robotic Arm Systems. In: Chew, E., et al. RiTA 2020. Lecture Notes in Mechanical Engineering. Springer, Singapore. https://doi.org/10.1007/978-981-16-4803-8_7

[14] M. Egorov, Z. N. Sunberg, E. Balaban, T. A. Wheeler, J. K. Gupta, and M. J. Kochenderfer, ”POMDPs.jl: A Framework for Sequential Decision Making under Uncertainty,” Journal of Machine Learning Research, vol. 18, no. 26, pp. 1-5, 2017. [Online]. Available: http://jmlr.org/papers/v18/16-300.html

[15] Thushara Sandakalum and Marcelo H. Ang, Jr., ”Motion Planning for Mobile Manipulators—A Systematic Review,” Machines, vol. 10, no. 2, p. 97, Feb. 2022. [Online]. Available: https://www.mdpi.com/2075-1702/10/2/97

[16] C.C. White III and D.J. White, ”Markov decision processes,” European Journal of Operational Research, vol. 40, no. 1, pp. 59-60, Jan. 1989. [Online]. Available: https://doi.org/10.1016/0377-2217(89)90348-2

[17] Silver, D., Huang, A., Maddison, C. et al. Mastering the game of Go with deep neural networks and tree search. Nature 529, 484–489 (2016). https://doi.org/10.1038/nature16961