CONTINUOUS MONTE CARLO GRAPH SEARCH

Anonymous authors
Paper under double-blind review

ABSTRACT

In many complex sequential decision making tasks, online planning is crucial for high-performance. For efficient online planning, Monte Carlo Tree Search (MCTS) employs a principled mechanism for trading off between exploration and exploitation. MCTS outperforms comparison methods in various discrete decision making domains such as Go, Chess, and Shogi. Following, extensions of MCTS to continuous domains have been proposed. However, the inherent high branching factor and the resulting explosion of search tree size is limiting existing methods. To solve this problem, this paper proposes Continuous Monte Carlo Graph Search (CMCGS), a novel extension of MCTS to online planning in environments with continuous state and action spaces. CMCGS takes advantage of the insight that, during planning, sharing the same action policy between several states can yield high performance. To implement this idea, at each time step CMCGS clusters similar states into a limited number of stochastic action bandit nodes, which produce a layered graph instead of an MCTS search tree. Experimental evaluation with limited sample budgets shows that CMCGS outperforms comparison methods in several complex continuous DeepMind Control Suite benchmarks and a 2D navigation task.

1 INTRODUCTION

Monte Carlo Tree Search (MCTS) is a well-known online planning algorithm for solving the decision making problem in discrete action spaces (Coulom, 2006; 2007). It has been shown that when a learned transition model is available, MCTS could be used for achieving super-human performance in various domains such as Atari, Go, Chess, and Shogi (Schrittwieser et al., 2020). That is why recent research has been trying to extend MCTS to environments with continuous state and action spaces (Hämäläinen et al., 2014; Rajamäki & Hämäläinen, 2018; Lee et al., 2020; Kim et al., 2020; Hubert et al., 2021).

There are several limitations in the current approaches for extending MCTS to environments with continuous states and actions. Approaches such as MCTS with progressive widening (Chaslot et al., 2008; Couëtoux et al., 2011), that build the search tree by discretizing the action space, do not scale up well to high-dimensional action spaces. On the other hand, learning-based approaches require a large number of samples and are also difficult to implement (Rajamäki & Hämäläinen, 2018; Hubert et al., 2021).

This paper presents Continuous Monte Carlo Graph Search (CMCGS), a novel extension of MCTS to the continuous control problem. Similar to MCTS, CMCGS employs an iterative mechanism for building a search graph that can the be used for sampling the next action. At each iteration, a series of operators is used to grow the search graph and update the information stored in the graph nodes. CMCGS uses state clustering and Gaussian action bandits to deal with challenges posed by continuous states and actions. Our experiments show that CMCGS demonstrates robust and superior performance compared to the baselines in several challenging benchmarks, including high-dimensional environments from DeepMind Control Suite (Tunyasuvunakool et al., 2020).

This paper makes the following contributions:

- We propose Continuous Monte Carlo Graph Search (CMCGS) algorithm, a straightforward extension to the popular MCTS algorithm for online planning in environments with continuous states and actions. CMCGS uses state clustering and stochastic action bandits to build a search graph, which can be used for controlling the agent.

- We evaluate the performance of CMCGS using several popular continuous control benchmarks and two novel environments with sparse rewards and requiring complex continuous control. The results show a clear advantage of CMCGS compared to the baselines.
- In order to share our findings with other researchers, we publish an open-source implementation of CMCGS on GitHub[1].

2 Related Work

Monte Carlo Tree Search is a successful decision-time planning method in discrete action space (Coulom, 2006; koc; Browne et al., 2012). It is a core component of the success of the computer Go game (Gelly & Silver, 2011; Silver et al., 2017; 2018). It also shows advantages in general game playing (Finnsson & Björnsson, 2008; Guo et al., 2014; Anthony et al., 2017; Grill et al., 2020). Several recent work combine MCTS with a learned dynamic model to lift the requirement of accessing dynamic models or simulators, making MCTS-based methods to be a general solution for decision making problems (Schrittwieser et al., 2020; Ye et al., 2021).

Online planning in continuous action spaces. In continuous action spaces, unlike discrete spaces, the action can be any real number within a pre-defined range. This property makes it challenging to apply MCTS in continuous action spaces. Instead, Cross-Entropy Method (CEM) is vastly used in the continuous action domain as an online planning method (Rubinstein, 1997; Rubinstein & Kroese, 2004; Weinstein & Littman, 2013; Chua et al., 2018; Hafner et al., 2019). Many methods combine CEM with a value function and/or policy in different ways to improve its performance (Negenborn et al., 2005; Lowrey et al., 2018; Bhardwaj et al., 2020; Hatch & Boots, 2021; Hansen et al., 2022). The main limitation of CEM compared to MCTS (and this work) is that CEM models the whole action trajectory using a single sampling distribution. This could be translated into the context of MCTS by using only one node at each layer of the search tree. This can limit the exploration capabilities of CEM in environments where there are several dissimilar ways for controlling the agent, for example, going around an obstacle using two different ways.

MCTS in continuous action spaces Since the success of MCTS in discrete action spaces, several attempts have been made to adopt it in the continuous action domain. To mitigate the requirement of enumerating all actions as in the discrete case, several work use progressive widening to increase the number of child actions of a node according to its number of visits (Coulom, 2007; Chaslot et al., 2008; Couëtoux et al., 2011; Moerland et al., 2018). Hamrick et al. (2020) directly split the action space into bins by factorizing across action dimensions (Tang & Agrawal, 2020). Together with a learned value function, policy and dynamic model, this method can successfully control a humanoid character with 21 action dimensions (Tassa et al., 2012). Besides these, a line of work extends MCTS to continuous action space via Hierarchical Optimistic Optimization (HOO) (Bubeck et al., 2008; Munos et al., 2014; Mao et al., 2020; Quinteiro et al., 2021). HOO hierarchically partitions the action space by building a binary tree to incrementally split the action space into smaller subspaces. Furthermore, several works grow the search tree based on sampling. Yee et al. (2016) use kernel regression to generalize the value estimation, thus only limited actions are sampled per node. Ahmad et al. (2020); Hubert et al. (2021) represent the policy using a neural network and sample from it when expanding the tree. By leveraging the learned policy, these methods achieve promising performance with limited search budgets. The main differences between this class of methods and our work is that our proposed algorithm builds a search graph instead of a search tree. Note that while we focus in this paper on the core search mechanism, our approach could also take advantage of a value function or policy learning similar to other approaches.

3 Preliminaries

Planning and search is arguably the most classical approach for optimal control (Fikes & Nilsson, 1971; Mordatch et al., 2012; Tassa et al., 2012). The idea of this approach is to start with a (rough) estimation of the action trajectory, and gradually improve it through model-based simulation of the environment. In the case of environments with complex dynamics, this is usually done online, i.e., at

[1]During the review process, the code is uploaded as the supplementary material. It will be published on GitHub after acceptance.

each timestep only a small part of the action trajectory is optimized and the first action in the best found trajectory is returned as the result. After executing the best found action, the whole process is repeated again, starting from the new visited state. This process is also known as closed-loop control, or model-predictive control (MPC). MPC has shown to be an effective approach in for optimal control in complex real-time environments (Samothrakis et al., 2014; Gaina et al., 2017; Babadi et al., 2018).

Monte Carlo Tree Search (MCTS) is one of the most popular MPC algorithms in time-sensitive environments such as video games (Perez et al., 2014; Holmgård et al., 2018). It was first introduced in 2006 (Coulom, 2006), when it demonstrated impressive performance in the game of Go (Coulom, 2007). The psuedocode of general MCTS is shown in Algorithm 1. It starts by initializing a single-node search tree using the current state s_t as the root, and grows the tree in an iterative manner, where at each iteration one node is added to the tree.

Algorithm 1 General Monte Carlo Tree Search (MCTS) algorithm

```
function MCTS(s_t)
    Let v_t be the root of the MCTS tree, with v_t.state = s_t.
    while within computational budget do
        v_s ← TREEPOLICY(v_t)
        r ← ROLLOUT(v_s)
        BACKUP(v_s, r)
    return Action from v_t to the best node
```

The iterative process of MCTS is comprised of the following four key steps:

1. **Selection (Tree Policy)**: In this step, the algorithm selects one of the tree nodes to be expanded next. This is done by starting from the root, and navigating to a child node until a node with at least one unexpanded child is visited. This is done using a selection criteria, whose goal is to balance the exploration-exploitation trade off.
2. **Expansion**: In this step, MCTS expands the search tree by adding a (randomly chosen) unvisited child to the selected node from the previous step.
3. **Simulation (Default Policy)**: After expanding the selected node, the newly added node is evaluated using a trajectory of random actions and computing the return.
4. **Backpropagation**: In the final step, the computed return is propagated backwards through the navigated nodes, updating the statistics stored in each node.

For a more detailed explanations of the MCTS algorithm, the reader is referred to (Browne et al., 2012).

4 METHOD

4.1 OVERVIEW

Figure 1 shows the core steps followed in one iteration of the proposed Continuous Monte Carlo Graph Search (CMCGS) algorithm. The idea behind each step is similar to its corresponding step in the original MCTS algorithm. CMCGS starts by initializing the search graph one node per layer, and continues by adding more nodes to the graph until the simulation budget is exhausted. Finally, the best action found in the root node of the graph is returned as the agent's next action. Important thing to note here is that, unlike MCTS, each node in the search graph of CMCGS corresponds to a cluster of visited states and has a replay memory for storing the experience tuples of states, actions, and accumulated rewards. Modelling clusters of visited states using nodes is the most fundamental building block of CMCGS that enables it to operate in environments with continuous states and actions using a relatively small search graph. We now explain the steps of the CMCGS algorithm shown in Figure 1.

(a) **Selection**: Starting from the root node, the selection mechanism is repeatedly applied to navigate through the search graph, until a leaf node or a terminal state is reached. This process is shown in Figure 1a. When visiting a node q, the node's policy $\pi_q(\mathbf{a})$ is used to sample and simulate a

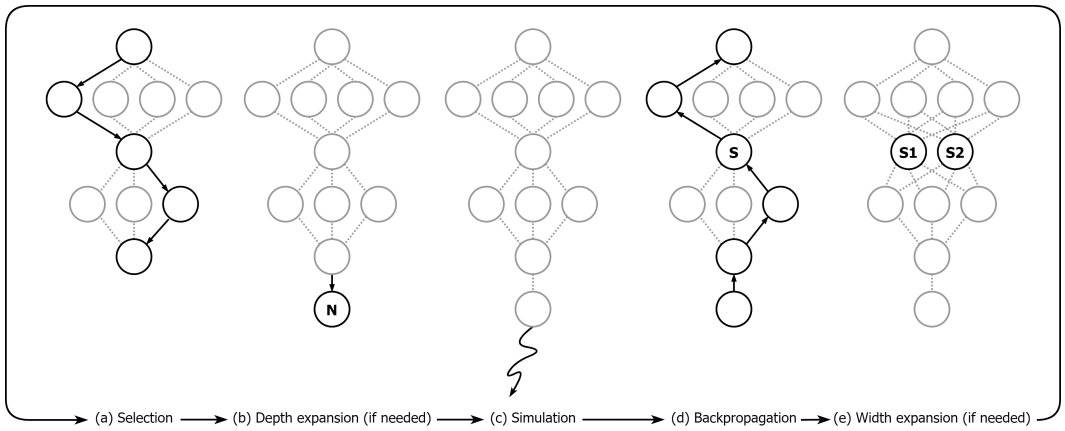


Figure 1: Core steps in one iteration of Continuous Monte Carlo Graph Search (CMCGS). a) Starting from the root node, the graph is navigated via action sampling and node selection until a leaf node is reached. b) If there is enough experience collected in the final layer of the graph, a new child node **N** is added to the previously-selected leaf node. c) A trajectory of random actions is simulated from the graph's leaf node to compute the accumulated reward. d) The computed accumulated reward is backpropagated through the selected nodes, updating their replay memories, policies, and node selection bandits. e) If a new cluster of experience data is found in a previous layer of the graph, all nodes in that layer are updated based on the new clustering information (in this example, the node **S** is split into two new nodes **S1** and **S2**).

random action $\mathbf{a}$, which updates the simulation state $\mathbf{s}$. The next node is then selected by finding the node that maximizes the likelihood of the state $\mathbf{s}$, i.e., $q \leftarrow \underset{q' \in \text{ the next layer}}{\arg\max} \; p_q\left(\mathbf{s}; \mu_{q'}, \sigma^2_{q'}\right)$.

(b) **Depth expansion (if needed)**: In the second step, CMCGS may add a new layer to the graph. This process is triggered if the graph has low depth or the replay memory of the graph's last layer has collected enough experience. Each new layer has initially a single node (as shown in Figure 1b).

(c) **Simulation**: Similar to MCTS, a trajectory of random actions is simulated starting from the last selected (leaf) node to update the accumulated reward. This is shown in Figure 1c.

(d) **Backpropagation**: The nodes visited during the selection step are updated by backpropagating the collected data (states, actions, and accumulated reward) through the graph and updating the replay memory of each visited node (shown in Figure 1d). This process also updates the policy and the state distribution of the nodes, if the corresponding replay memory contains enough experience.

(e) **Width expansion (if needed)**: The algorithm may increase the width of a graph layer if the visited states stored in that layer could be clustered into more clusters than the current ones. In this case, as shown in Figure 1e, for each new cluster a new node is added to the layer and all the collected experience tuples are re-assigned to their corresponding nodes. Each affected node will then update its policy and state distribution based on its fresh replay memory.

The pseudocode of the CMCGS algorithm is shown in Algorithm 2. Next we explain the implementation details of the algorithm.

4.2 Implementation Details

In order to implement CMCGS, one has to make a few design choices that affect the performance of the algorithm. Fortunately, most of these choices are intuitive and therefore easy to control, leading to a few hyperparameters in the algorithm. Here, we explain these details that are used throughout the paper for producing the results.

State Distribution at Each Node: Each node q in the search graph uses a Gaussian distribution $\mathcal{N}(\mu_q, \Sigma_q)$ to model the state distribution of its replay memory. The mean μ_q and covariance

Algorithm 2 Continuous Monte Carlo Graph Search

function CMCGS($\mathbf{s}_t$)
  Initialize the search graph with one node per layer and q_{root} as the root node in the first layer.
  while within computational budget **do**
    $\tau, r \leftarrow$ GRAPHPOLICY(s_t, q_{root}) ▷ $\tau \leftarrow \{\langle q_{t'}, s_{t'}, a_{t'}, s_{t'+1}\rangle\}_{t'=t}^{t+d-1}$
    $r \leftarrow$ ROLLOUT(s_d, r)
    BACKUP(τ, r)
  return best action found in the replay memory of the root node q_{root}
function GRAPHPOLICY($\mathbf{s}, q$)
  $\tau \leftarrow \emptyset$
  $r \leftarrow 0$
  while s is not terminal **and** not visited all graph layers **do**
    $a \sim \pi_q(a)$ ▷ Action bandit
    Apply action a and observe new state s' and reward r'
    $\tau \leftarrow \tau \cup \{\langle q, s, a, s'\rangle\}$
    $r \leftarrow r + r'$
    if s' is not terminal **and** we are visiting the last layer of the graph **then**
      Try adding a new layer with one node to the graph.
    if s' is terminal **or** we are visiting the last layer of the graph **then**
      break
    $q \leftarrow \underset{q' \in \text{ the next layer}}{\arg\max} \; p\left(s'; \mu_{q'}, \sigma^2_{q'}\right)$ ▷ Q bandit
    $s \leftarrow s'$
  return τ, r
function ROLLOUT(s, r)
  while within rollout budget **and** s is not terminal **do**
    Choose a uniformly at random
    Apply action a and update state s and accumulated episode return r
  return r
function BACKUP(τ, r)
  for each $\langle q_t, s_t, a_t, s_{t+1}\rangle \in \tau$ **do**
    Store the new experience $\langle s_t, a_t, r, s_{t+1}\rangle$ in the replay memory of q_t
    Compute c_t^*, the desired number of clusters in layer t
    if there is less than c_t^* clusters in layer t **then**
      Try to cluster the data in the replay memory of layer t into c_t^* clusters
      if clustering was successful **then**
        For each new cluster, add a new node to layer t of the graph
        Use the replay memory of each node q' to update its bandits
    if no new cluster has been found in layer t **then**
      Only update the bandits of the visited node q_t using its replay memory

matrix Σ_q of this distribution are estimated using the visited states in the replay memory of q. Our experiments showed no significant advantage of using the full covariance matrix. Therefore, for the sake of faster performance, we used diagonal covariance matrices throughout our experiments.

Policy (Action Bandit) at Each Node: A second Gaussian distribution (with a scalar variance) is used to model the policy $\pi_q(\mathbf{a})$ of each node q. The mean of this distribution is computed using the highest scoring experiences stored in the replay memory of the node q. This process is similar to how CEM updates its sampling distribution, with the difference that CMCGS uses a one-step action bandit at each node. The scalar variance of the policy is initialized at 0.5 and reduced to 0.15 as the size of the replay memory increases. The policy $\pi_q(\mathbf{a})$ is then used during the selection step to sample random actions.

Selection of Q Nodes: As explained in Section 4.1, during the selection step, CMCGS repeatedly samples random actions and navigates to a q node in the next layer. In this step, CMCGS navigates to the q node that maximizes the likelihood of the new visited state $\mathbf{s}$, i.e., $q \leftarrow \underset{q' \in \text{ the next layer}}{\arg\max} \; p_q\left(\mathbf{s}; \mu_{q'}, \sigma^2_{q'}\right)$.

Table 1: Hyperparameters used in the experiments

Parameter description	Value
Initial depth of the search graph	3
Maximum rollout length	5
Clustering algorithm	Agglomerative clustering
Ratio of elite samples to be used for updating the action bandits	0.1
Discount factor	1

Hyperparameters: Table 1 shows the list of hyperparameters used throughout our implementation. Note that, unlike MCTS, in CMCGS the graph is initialized with an initial number of layers (3 in our setup). In our experiments, this turned out to be an effective approach for better exploitation of visited experience tuples. For the clustering, we used Agglomerative Clustering provided in Scikit-learn library (Pedregosa et al., 2011).

5 EXPERIMENTS

We implemented CMCGS using Python, and the source code used throughout this paper is available on GitHub[2]. We compared CMCGS with (1) Monte Carlo Tree Search with Progressive Widening (MCTS-PW) (Chaslot et al., 2008; Couëtoux et al., 2011) and (2) Cross-Entropy Method (CEM) (Rubinstein & Kroese, 2004; Weinstein & Littman, 2013). To keep the experiments fair, we did not compare CMCGS against algorithms that employ learning of the value function or the policy, such as the recent work of Hubert et al. (2021). We used the following simulation environments throughout our experiments:

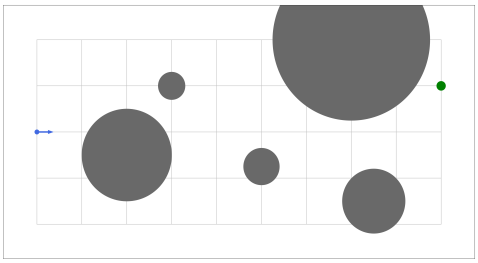
1. **2D Navigation**: We developed an environment for control of a particle such that it tries to reach the goal (shown in green in Figures 2a–2b) without colliding with the obstacles. We used two variations of this environment using circular and rectangular obstacles (shown in gray in Figures 2a and Figure 2b, respectively). In these environments, a new action is applied whenever the particle reaches a vertical gray line. We used a continuous reward that encouraged reaching the goal while penalizing high accelerations and collision with obstacles. The main challenge in these environments is that the agent has to repeatedly pick the direction for going around the obstacles.
2. **2D Reacher**: Since *2D Navigation* environments only have 1 Degree of Freedom (DOF), we also developed a variation of the classic reacher task for a 2D multi-link arm. In this environment, only a sparse binary reward is used for reaching the goal (shown in green in Figures 2c–2d), and there were some obstacles (shown in black) that blocked the path. In order to evaluate the scalability of CMCGS, we used two variations of this environment using 15- and 30-linked arms, as shown in Figure 2c and Figure 2d, respectively.
3. **DeepMind Control Suite**: We also used several popular environments provided by DeepMind Control Suite (Tassa et al., 2018) in our experiments. These environments pose a wide range of challenges for continuous control with low- and high-dimensional states and actions.

6 RESULTS

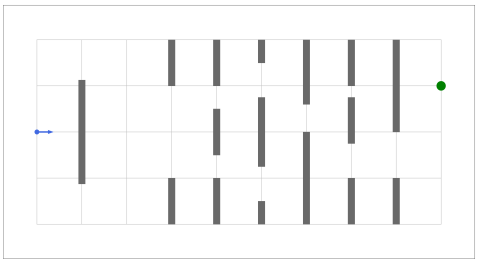
6.1 ABILITY IN EXPLORATION

Figure 3 shows how CMCGS explores the state and action space in the *2d-navigation-circles* task. As it can be seen in the figure, CMCGS demonstrates a good ability in exploration of the state space and, in particular, is able to find different ways for going around the obstacles. The supplementary material includes the videos for showing the full episode and how search graphs are generated in different steps of the *2d-navigation-circles* task.

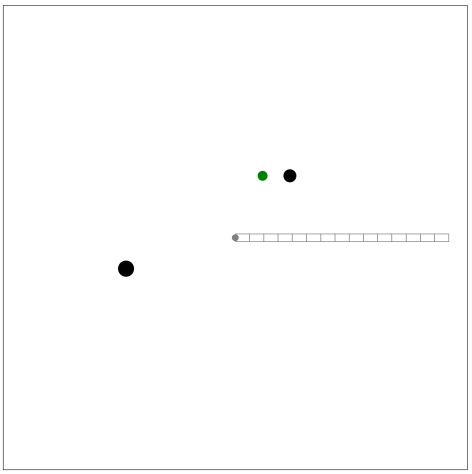
[2]During the review process, the code is uploaded as the supplementary material. It will be published on GitHub after acceptance.



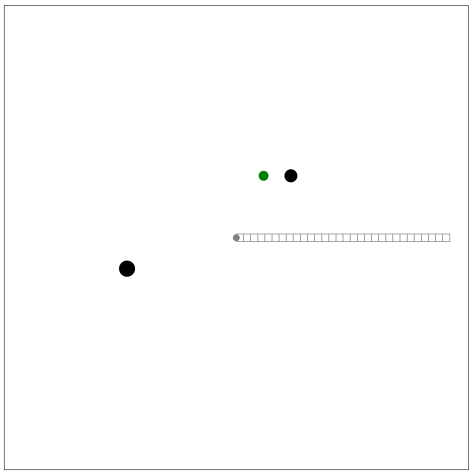
(a) 2D navigation task using circular obstacles (*2d-navigation-circles*)



(b) 2D navigation task using rectangular obstacles (*2d-navigation-boxes*)



(c) 2D reacher task using 15-linked arm (*2d-reacher-fifteen-poles*)



(d) 2D reacher task using 30-linked arm (*2d-reacher-thirty-poles*)

Figure 2: Developed environments that were used in the experiments

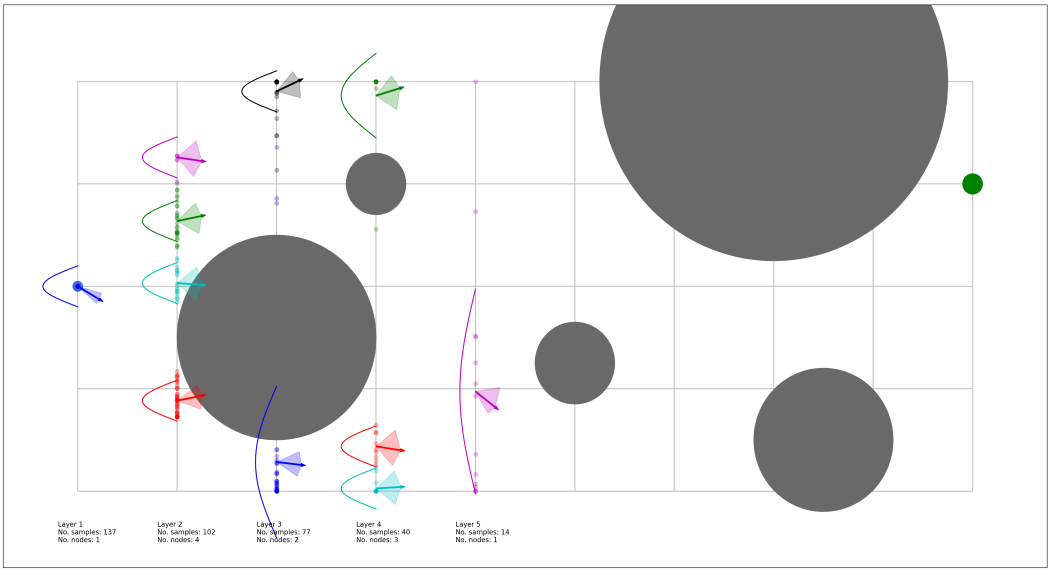


Figure 3: The search graph generated using CMCGS for the *2d-navigation-circles* task. The generated graph has 5 layers, where each node in the graph is shown using a different color and by projecting its state and action distributions on the navigation environment.

6.2 ROBUSTNESS

Figure 4 shows the reward plots for CMCGS and the tested baselines using different simulation budgets per control timestep. The plots show the mean and standard deviation of the undiscounted sum of rewards over the full episode, computed using 10 random seeds per each configuration. All experiments used the same set of hyperparameters, as shown in Table 1. The supplementary material includes example videos of how CMCGS works in different environments.

As it can be seen in Figure 4, CMCGS demonstrates the best performance compared to CEM and MCTS-PW. CEM achieves similar or slightly better performance in only a few environments such as *reacher-hard*, *walker-walk*, and *walker-run*. MCTS-PW is the weakest method in all environments except *2d-reacher-fifteen-poles* and *2d-reacher-thirty-poles*.

An important observation from Figure 4 is that CMCGS performs well even with small simulation budgets of less than 1000 timesteps. On the other hand, CEM requires at least 2000 timesteps to reach similar performance. This behavior can be easily seen in *ball_in_cup_catch*, *finger-spin*, *cheetah-run*, *walker-walk*, and *walker-run* environments.

Figure 4 also shows how the performance of each algorithm changes for different simulation budgets per timestep. As it can be seen in the plots, CMCGS and MCTS-PW demonstrate the least sensitivity to the simulation budget, which could be the result of Monte Carlo estimation of the return at each node.

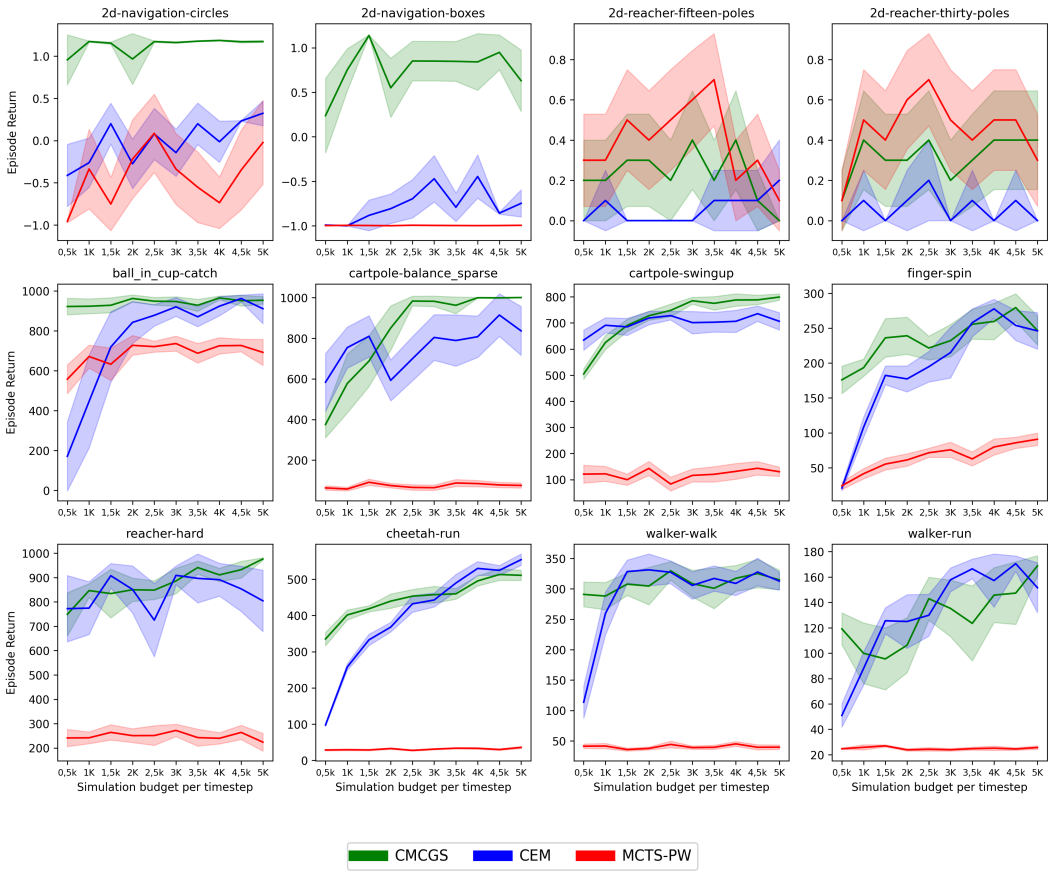


Figure 4: Reward plots for different simulation budgets per timestep. As explained in Section 5, *2d-X* environments (shown in the first row) have been developed by the authors, and the rest of the environments are from DeepMind Control Suite. The proposed CMCGS algorithm demonstrates superior performance in most of the tested environments, especially when using small simulation budgets of less than 1000 timesteps.

7 LIMITATIONS AND FUTURE WORK

In this section, we explain the main limitations of the proposed algorithm and how they could be addressed in future work.

Bootstrapping: In our current implementation, at each timestep CMCGS builds the search graph from scratch. It has been shown that in model-predictive control it is usually beneficial to use the best found trajectory from previous timestep(s) to bias the optimization process towards more promising trajectories. In CMCGS, this could be done by biasing the mean of the action policies of the search nodes towards old best actions. A more trivial approach would be to simply inject the old best trajectory into the replay memory of the search nodes.

Learning: It has been shown that MCTS could benefit from learned value functions or rollout policies that have been trained using pre-recorded datasets. We believe that this could be a trivial next step for improving the performance of CMCGS.

Arbitrary graph structures: Currently, CMCGS uses layered graph structures to explore the state space. This does not allow the search graph to re-use previous experience in environments where the agent can navigate back and forth between different states (such as searching through a maze). This limitation could be lifted by allowing the algorithm to build arbitrary graph structures (such as complete graphs) to better explore the structure of the state space.

8 CONCLUSION

In this paper, we proposed Continous Monte Carlo Graph Search (CMCGS), an extension of the popular MCTS algorithm for solving decision making problems with continuous state and action spaces. CMCGS builds up on the observation that different regions of the state space ask for different action bandits for estimating the value function. Based on this observation, CMCGS builds a layered search graph where at each layer the visited states are clustered into several stochastic action bandit nodes. This allows CMCGS to solve complex continuous control problems with small search graphs. Our experiments show that CMCGS outperforms popular benchmarks in several complex continuous environments, such as DeepMind Control Suite benchmarks. We believe that the proposed CMCGS algorithm could be a building block for a new family of Monte Carlo methods applied to decision making problems with continuous action spaces.

REFERENCES

Zaheen Ahmad, Levi Lelis, and Michael Bowling. Marginal utility for planning in continuous or large discrete action spaces. *Advances in Neural Information Processing Systems*, 33:1937–1946, 2020.

Thomas Anthony, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search. *Advances in Neural Information Processing Systems*, 30, 2017.

Amin Babadi, Kourosh Naderi, and Perttu Hämäläinen. Intelligent middle-level game control. In *Proceedings of IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 25–32. IEEE, 2018.

Mohak Bhardwaj, Ankur Handa, Dieter Fox, and Byron Boots. Information theoretic model predictive q-learning. In *Learning for Dynamics and Control*, pp. 840–850. PMLR, 2020.

Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

Sébastien Bubeck, Gilles Stoltz, Csaba Szepesvári, and Rémi Munos. Online optimization in x-armed bandits. *Advances in Neural Information Processing Systems*, 21, 2008.

Guillaume M JB Chaslot, Mark HM Winands, H Jaap van den Herik, Jos WHM Uiterwijk, and Bruno Bouzy. Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation*, 4(03):343–357, 2008.

Kurtland Chua, Roberto Calandra, Rowan McAllister, and Sergey Levine. Deep reinforcement learning in a handful of trials using probabilistic dynamics models. *Advances in neural information processing systems*, 31, 2018.

Adrien Couëtoux, Jean-Baptiste Hoock, Nataliya Sokolovska, Olivier Teytaud, and Nicolas Bonnard. Continuous upper confidence trees. In *International Conference on Learning and Intelligent Optimization*, pp. 433–445. Springer, 2011.

Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pp. 72–83. Springer, 2006.

Rémi Coulom. Computing "elo ratings" of move patterns in the game of go. *ICGA journal*, 30(4): 198–208, 2007.

Richard E Fikes and Nils J Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

Hilmar Finnsson and Yngvi Björnsson. Simulation-based approach to general game playing. In *Aaai*, volume 8, pp. 259–264, 2008.

Raluca D Gaina, Simon M Lucas, and Diego Pérez-Liébana. Population seeding techniques for rolling horizon evolution in general video game playing. In *Evolutionary Computation (CEC), 2017 IEEE Congress on*, pp. 1956–1963. IEEE, 2017.

Sylvain Gelly and David Silver. Monte-carlo tree search and rapid action value estimation in computer go. *Artificial Intelligence*, 175(11):1856–1875, 2011.

Jean-Bastien Grill, Florent Altché, Yunhao Tang, Thomas Hubert, Michal Valko, Ioannis Antonoglou, and Rémi Munos. Monte-carlo tree search as regularized policy optimization. In *International Conference on Machine Learning*, pp. 3769–3778. PMLR, 2020.

Xiaoxiao Guo, Satinder Singh, Honglak Lee, Richard L Lewis, and Xiaoshi Wang. Deep learning for real-time atari game play using offline monte-carlo tree search planning. *Advances in neural information processing systems*, 27, 2014.

Danijar Hafner, Timothy Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. Learning latent dynamics for planning from pixels. In *International conference on machine learning*, pp. 2555–2565. PMLR, 2019.

Perttu Hämäläinen, Sebastian Eriksson, Esa Tanskanen, Ville Kyrki, and Jaakko Lehtinen. Online motion synthesis using sequential monte carlo. *ACM Transactions on Graphics (TOG)*, 33(4):51, 2014.

Jessica B Hamrick, Abram L Friesen, Feryal Behbahani, Arthur Guez, Fabio Viola, Sims Witherspoon, Thomas Anthony, Lars Buesing, Petar Veličković, and Théophane Weber. On the role of planning in model-based deep reinforcement learning. *arXiv preprint arXiv:2011.04021*, 2020.

Nicklas Hansen, Xiaolong Wang, and Hao Su. Temporal difference learning for model predictive control. *arXiv preprint arXiv:2203.04955*, 2022.

Nathan Hatch and Byron Boots. The value of planning for infinite-horizon model predictive control. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 7372–7378. IEEE, 2021.

Christoffer Holmgård, Michael Cerny Green, Antonios Liapis, and Julian Togelius. Automated playtesting with procedural personas through mcts with evolved heuristics. *IEEE Transactions on Games*, 11(4):352–362, 2018.

Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Mohammadamin Barekatain, Simon Schmitt, and David Silver. Learning and planning in complex action spaces. In *International Conference on Machine Learning*, pp. 4476–4486. PMLR, 2021.

Beomjoon Kim, Kyungjae Lee, Sungbin Lim, Leslie Kaelbling, and Tomás Lozano-Pérez. Monte carlo tree search in continuous spaces using voronoi optimistic optimization with regret bounds. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pp. 9916–9924, 2020.

Jongmin Lee, Wonseok Jeon, Geon-Hyeong Kim, and Kee-Eung Kim. Monte-carlo tree search in continuous action spaces with value gradients. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pp. 4561–4568, 2020.

Kendall Lowrey, Aravind Rajeswaran, Sham Kakade, Emanuel Todorov, and Igor Mordatch. Plan online, learn offline: Efficient learning and exploration via model-based control. *arXiv preprint arXiv:1811.01848*, 12:2825–2830, 2018.

Weichao Mao, Kaiqing Zhang, Qiaomin Xie, and Tamer Basar. Poly-hoot: Monte-carlo planning in continuous space mdps with non-asymptotic analysis. *Advances in Neural Information Processing Systems*, 33:4549–4559, 2020.

Thomas M Moerland, Joost Broekens, Aske Plaat, and Catholijn M Jonker. A0c: Alpha zero in continuous action space. *arXiv preprint arXiv:1805.09613*, 2018.

Igor Mordatch, Emanuel Todorov, and Zoran Popović. Discovery of complex behaviors through contact-invariant optimization. *ACM Transactions on Graphics (TOG)*, 31(4):1–8, 2012.

Rémi Munos et al. From bandits to monte-carlo tree search: The optimistic principle applied to optimization and planning. *Foundations and Trends® in Machine Learning*, 7(1):1–129, 2014.

Rudy R Negenborn, Bart De Schutter, Marco A Wiering, and Hans Hellendoorn. Learning-based model predictive control for markov decision processes. *IFAC Proceedings Volumes*, 38(1): 354–359, 2005.

Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in Python. *Journal of machine Learning research*, 12:2825–2830, 2011.

Diego Perez, Spyridon Samothrakis, and Simon Lucas. Knowledge-based fast evolutionary mcts for general video game playing. In *2014 IEEE Conference on Computational Intelligence and Games*, pp. 1–8. IEEE, 2014.

Ricardo Quinteiro, Francisco S Melo, and Pedro A Santos. Limited depth bandit-based strategy for monte carlo planning in continuous action spaces. *arXiv preprint arXiv:2106.15594*, 2021.

Joose Julius Rajamäki and Perttu Hämäläinen. Continuous control monte carlo tree search informed by multiple experts. *IEEE transactions on visualization and computer graphics*, 2018.

Reuven Y Rubinstein. Optimization of computer simulation models with rare events. *European Journal of Operational Research*, 99(1):89–112, 1997.

Reuven Y Rubinstein and Dirk P Kroese. *The cross-entropy method: a unified approach to combinatorial optimization, Monte-Carlo simulation, and machine learning*, volume 133. Springer, 2004.

Spyridon Samothrakis, Samuel A Roberts, Diego Perez, and Simon M Lucas. Rolling horizon methods for games with continuous states and actions. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, pp. 1–8. IEEE, 2014.

Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419): 1140–1144, 2018.

Yunhao Tang and Shipra Agrawal. Discretizing continuous action space for on-policy optimization. In *Proceedings of the aaai conference on artificial intelligence*, volume 34, pp. 5981–5988, 2020.

Yuval Tassa, Tom Erez, and Emanuel Todorov. Synthesis and stabilization of complex behaviors through online trajectory optimization. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 4906–4913. IEEE, 2012.

Yuval Tassa, Yotam Doron, Alistair Muldal, Tom Erez, Yazhe Li, Diego de Las Casas, David Budden, Abbas Abdolmaleki, Josh Merel, Andrew Lefrancq, et al. Deepmind control suite. *arXiv preprint arXiv:1801.00690*, 2018.

Saran Tunyasuvunakool, Alistair Muldal, Yotam Doron, Siqi Liu, Steven Bohez, Josh Merel, Tom Erez, Timothy Lillicrap, Nicolas Heess, and Yuval Tassa. $\text{dm}_c ontrol : Softwareandtasksforcontinuouscontrol$. *Software Impacts*, $6 : 100022, 2020. ISSN2665 - 9638. doi : .$ URL https://www.sciencedirect.com/science/article/pii/S2665963820300099.

Ari Weinstein and Michael Littman. Open-loop planning in large-scale stochastic domains. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 27, pp. 1436–1442, 2013.

Weirui Ye, Shaohuai Liu, Thanard Kurutach, Pieter Abbeel, and Yang Gao. Mastering atari games with limited data. *Advances in Neural Information Processing Systems*, 34:25476–25488, 2021.

Timothy Yee, Viliam Lisỳ, and Michael H Bowling. Monte carlo tree search in continuous action spaces with execution uncertainty. In *IJCAI*, pp. 690–697, 2016.