Extensions of Monte Carlo Tree Search into Continuous Action Spaces

Risto Koverola

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 30.5.2022

Supervisor

Prof. Perttu Hämäläinen

Advisor

D.Sc. Aditya Acharya

Aalto University
School of Science

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Risto Koverola

Title Extensions of Monte Carlo Tree Search into Continuous Action Spaces

Degree programme Computer, Communication and Information Sciences

Major Computer Science **Code of major** SCI3042

Supervisor Prof. Perttu Hämäläinen

Advisor D.Sc. Aditya Acharya

Date 30.5.2022 **Number of pages** 40+1 **Language** English

Abstract

This thesis explored Monte Carlo Tree Search (MCTS) algorithm and its extensions to continuous action spaces. MCTS is a relevant search algorithm used in cutting edge AI design in discrete domains. However, its extensions to continuous action spaces have not been comprehensibly listed and categorized. Furthermore empirical comparison of their performance has been incomplete.

This thesis consisted of a literary survey and an empirical comparison of MCTS extensions. The aim was to find out how the different methods work, where they can be applied and how they compare to each other. In addition the performance of the extensions was compared against each other and other continuous non-MCTS algorithms.

The survey found twelve prominent MCTS extensions of varying complexity and performance. These extensions could be categorized into four general approaches: Manual/Informed discretization, Unpruning, Action optimization and Policy optimization. Three of these extension methods were benchmarked in the empirical study: Discrete Upper Confidence Bounds applied to Trees, Progressive Widening and Value-Gradient Upper Confidence Trees. In addition two baseline algorithms called Random Branching Depth and Covariance Matrix Adaptation Evolution Strategy (CMA-ES) were used to further evaluate the performance of MCTS methods.

The results of the performance study suggest that Progressive Widening is the best general purpose MCTS extension when using offline planning in continuous control settings. The results also seem to suggest that MCTS extensions could perform better than CMA-ES when action dimensionality and/or planning horizon depth increases. Further research could be conducted by including more extensions in the performance study. In addition online planning and time limited open loop planning performance also provides an interesting research direction when considering usage in real-time problems.

Keywords Monte Carlo tree search, MCTS, continuous action spaces, continuous control, UCT, continuous UCT, TLS, HOOT, VOOT, Progressive Widening, VSP-MCTS, KR-UCT, VG-UCT, A0C, FDI-MCTS, TPO, CMA-ES, benchmark

Aalto University
School of Science

Aalto-yliopisto, PL 11000, 00076 AALTO
www.aalto.fi
Diplomityön tiivistelmä

Tekijä Risto Koverola

Työn nimi Monte Carlo Tree Search algoritmin laajentaminen jatkuviin toiminta-avaruuksiin

Koulutusohjelma Computer, Communication and Information Sciences

Pääaine Computer Science **Pääaineen koodi** SCI3042

Työn valvoja Prof. Perttu Hämäläinen

Työn ohjaaja D.Sc. Aditya Acharya

Päivämäärä 30.5.2022 **Sivumäärä** 40+1 **Kieli** Englanti

Tiivistelmä

Tämä maisterintyö tutki Monte Carlo Tree Search (MCTS) hakualgoritmiä ja sen laajentamista jatkuviin toiminta-avaruuksiin. MCTS on suosittu hakualgoritmi, jolla on tekoälytutkimuksessa saavutettu osa parhaista tuloksista diskreeteissä toiminta-avaruuksissa, kuten lautapeleissä. MCTS on onnistuneesti laajennettu toimimaan myös jatkuvissa toiminta-avaruuksissa, mutta näitä laajennusmetodeja ei ole tyhjentävästi listattu tai kategorisoitu. Lisäksi empiirinen tutkimus metodien toimintakyvystä on puutteellista.

Tämä maisterityö koostuu kirjallisuuskatsauksesta sekä empiirisestä tutkimuksesta. Työn tarkoituksena on löytää miten metodit toimivat, missä niitä voi soveltaa ja miten ne vertautuvat toisiinsa. Tämän lisäksi laajennusmetodien toimintakykyä verrataan toisiinsa sekä myös muihin algoritmeihin, joita käytetään jatkuvissa toiminta-avaruuksissa.

Kirjallisuuskatsauksessa löydettiin kaksitoista laajennusmenetelmää, jotka voitiin kategorisoida neljään yleiseen lähestymistapaan: manuaalinen/tietoon perustuva diskretisointi, toiminta-avaruuden kasvatus, toimioptimointi sekä toimintaperiaateoptimointi. Empiiriseen tutkimukseen valittiin kolme menetelmää: Discrete Upper Confidence Bounds applied to Trees, Progressive Widening ja Value-Gradient Upper Confidence Trees. Lisäksi tutkimuksessa käytettiin kahta vertailualgoritmia, jotka eivät perustu MCTS algoritmiin: Covariance Matrix Adaptation Evolution Strategy (CMA-ES) sekä Random Branching Depth.

Tutkimuksen tulokset viittaavat siihen että Progressive Widening on paras MCTS laajennusmenetelmä yleiseen käyttöön jatkuvissa toiminta-avaruuksissa. Tulokset myös vihjaavat, että MCTS menetelmät saattavat toimia paremmin kuin CMA-ES, kun toiminta-avaruuden ja/tai suunnitteluhorisontin pituus kasvaa. Tulevaisuudessa olisi kiinnostavaa nähdä laajempaa tutkimusta eri menetelmien välillä. Esimerkiksi toimintakyky aikarajoitteisessa jatkuvassa toiminta-avaruudessa on tärkeää, kun halutaan toimia tosiaikaisten ongelmien parissa.

Avainsanat Monte Carlo tree search, MCTS, jatkuva toiminta-avaruus, UCT, continuous UCT, TLS, HOOT, VOOT, Progressive Widening, VSP-MCTS, KR-UCT, VG-UCT, A0C, FDI-MCTS, TPO, CMA-ES, vertaileva tutkimus

Contents

1 Introduction

Google's recent artificial intelligence known as AlphaZero plays board games: Chess, Shogi and Go. It utilizes a combination of machine learning and a search algorithm called Monte Carlo Tree Search (MCTS). With just 24 hours of training, the artificial intelligence learned to not only beat human experts of each game, but also current world champion programs in each game (Silver et al., 2017).

Monte Carlo Tree Search is a search algorithm utilized in sequential decision making. It can be used, for example, when designing an artificial intelligence for a board game. In board games, the player can at each point of the game take only a limited number of actions. This is known as a discrete action space. When the number of states the game can be in is limited in the same way, the state space is also discrete.

The MCTS algorithm and its variations have received considerable interest, because it can be used with large action and state spaces, especially when combined with other techniques. That is to say, it works well with games, where there are a large number of possible actions a player can take at each step of the game. Decision making in these kind of games is computationally intensive.

There are still many open questions in the field of MCTS research (Browne et al., 2012). One interesting problem is extending the MCTS algorithm to work in continuous action spaces, where the number of actions that can be taken at each step is limitless. This would allow the algorithm to not only work in board games, but also in continuous domains like decision making in the real world. While many extension methods have been invented, they have not been comprehensibly categorized and compared with each other. In addition comparison to other continuous non-MCTS methods is rare.

This thesis includes a literary overview of extensions of MCTS into continuous domains and also an empirical comparison of the methods. In addition the methods are compared to a state of the art non-sequential black box optimization method for continuous domains called Covariance Matrix Adaptation Evolution Strategy (CMA-ES), which can be applied to sequential problems by concatenating the sequential actions into one long vector of optimized variables. The thesis first presents an introduction to how the MCTS algorithm works in discrete action spaces. Then it explores the different existing methods for extending the MCTS algorithm to continuous action spaces. Finally the methods are compared to each other and the state of the art in an empirical study. The research questions are:

- How do the methods work?
- Where can the methods be applied?
- How do they compare to each other?
- How do they compare to non-sequential optimization of actions?

1.1 Search process

For the interest of the reader, the papers included in this thesis are listed already here in the introduction segment in Table 1. To find the methods for this thesis a literary survey was conducted using the Google Scholar search engine. Table 1 also shows used search terms and results to indicate how the search was conducted.

The papers that were excluded typically explored problems, which were not directly related to the problem explored in this thesis. These papers explored methods for dealing with stochastic transitions, performance improvement and concurrency. For the interest of the reader, some performance improvements and special variations are mentioned in this thesis, but they are not explored in depth.

1.2 Terminology and notation

The background and terminology for the MCTS algorithm is rooted in decision theory, game theory, Monte Carlo methods and bandit based methods (Browne et al., 2012). The following notation will be used in this thesis:

- S: Set of game states, with s_0 representing the initial state and terminal states are represented as s_t. The set of states S can be discrete or continuous.
- V: Set of nodes in the search trees, each node v_i having a corresponding state s_i. The state of a node must be acquirable in some way $s_i = s(v_i)$.
- A: Set of actions a. The set of actions associated with a decision making problem is known as its action space. The number of actions a in this set is the dimensionality of the action space. Actions can be discrete or continuous.
- A_t: A trajectory A_t is a list of actions $A_t = \{a_0, \ldots, a_t\}$ of length t, which are taken in sequence from a starting state s_0.
- R: Reward associated with a state s. Rewards can be for example numerical value 1 for win and -1 for loss or a continuous value. They must be acquirable in some way $R_i = R(s_i)$
- $f : (S, A) \to S$: State transition function, which given a state and action gives the new state.
- $\pi : A \to S$: Policy π is a mapping from states to actions. Given a state s, the policy can be queried for an action $a = \pi(s)$.

Search terms	Relevant papers
"Monte Carlo Tree Search" and "continuous" or "continuous action space"	• Cowling et al. (2013) • Fischer et al. (2015)
"Continuous Monte Carlo Tree Search"	• Couetoux et al. (2011a) • Van den Broeck and Driessens (2011) • Yee et al. (2016) • Couetoux (2013) • Auger et al. (2013)
"Monte Carlo Tree Search Continuous Control"	• Kocsis and Szepesvári (2006) • Rajamaki and Hamalainen (2018)
"Continuous MCTS"	• Moerland et al. (2018)
"Continuous Control MCTS"	• Lee et al. (2020) • Ba et al. (2019) • Yazdanbakhsh et al. (2019) • Kim et al. (2020)
Related and referenced articles in Continuous Upper Confidence Trees	• Couetoux et al. (2011b) • Rolet et al. (2009)
Related and referenced articles in Continuous Upper Confidence Trees with Polynomial Exploration – Consistency	• Mansley et al. (2011) • Weinstein and Littman (2012)
"Monte Carlo Tree Search"	• Browne et al. (2012)

Table 1: Papers included in the literary overview and associated search terms.

2 Monte Carlo Tree Search

The following section will examine the basic MCTS algorithm. The most popular variation of MCTS, which is also introduced here is known as Upper Confidence Bounds applied to Trees (UCT).

2.1 Monte Carlo Tree Search in discrete action spaces

The MCTS algorithm is an anytime algorithm that builds a search tree. It iterates on a search tree until a computational budget runs out. The computational budget can be limited by time, memory or iteration count. Once the iteration stops it returns a optimal choice. Typically more computational budget leads to better choices. (Browne et al., 2012)

The algorithm uses two policies for decision making: a default policy π_d and a tree policy π_t. A policy is simply a function, which given a state, outputs a resulting action. In other words, they make decisions based on the game state. The iteration itself happens in four steps: selection, expansion, simulation and backpropagation (Browne et al., 2012).

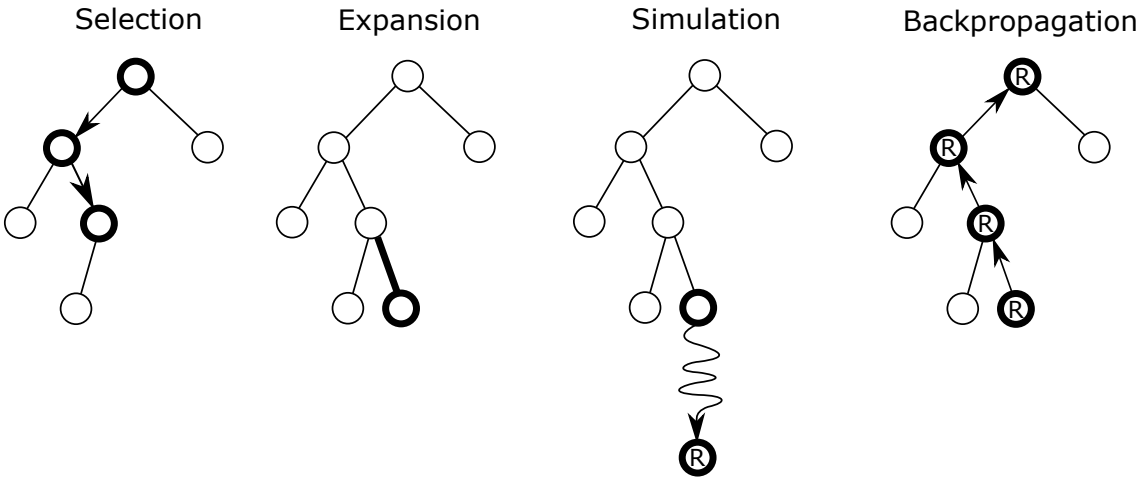

Figure 1: Phases of the MCTS algorithm

- Selection

  In the selection phase, the most urgent expandable child is selected from the search tree using the tree policy π_t. A child node is not expandable if it has already been expanded or it is a terminal node. The child selection is started at the root node and continues recursively to each child node.

- Expansion

  In the expansion phase the selected child node is expanded by adding one or several children nodes to it. These are the nodes which will be simulated forward in the next step.

- Simulation

  In the simulation phase the child nodes are simulated forward using a default policy π_d. A typical default policy is simply choosing random actions, so that the simulation is efficient and gives an approximation of what the possible rewards are starting from this node. Simulation can be done until a terminal state is reached. Alternatively it can be cut off at an intermediate state and the reward of that state can be evaluated using heuristics. The simulation or forward simulation is also known as a rollout.

- Backpropagation

  In the backpropagation step, the results of simulation are collected using the reward function. The reward values are then propagated back up the tree. This means updating the statistics of the parent nodes.

These four steps are typically repeated many times in an iteration. After the iteration has ended, the MCTS algorithm chooses the best child. In the basic MCTS algorithm this is the child with the highest expected reward.

The MCTS algorithm in pseudo code:

Algorithm 1 MCTS

$v_0 \leftarrow$ create root node v_0 with state s_0
while withing budget **do**
  $v_1 \leftarrow$ TreePolicy(v_0)
  $R \leftarrow$ DefaultPolicy(v_1)
  Backup(v_1, R)
end while
return $a \leftarrow$ BestChild(v_0)

After the best action at this point has been selected, it can be taken. The existing search tree can be discarded and a new one built from the next state. Alternatively the existing search tree can be used as basis for the next iteration.

2.2 Multi-Armed Bandits

To understand the most popular variation of MCTS, we need to briefly focus on a problem known as a multi-armed bandit or a k-armed bandit. A one-armed bandit is a term used for a slot machine, with a single lever. Each time the lever is pulled, a random reward R is given. The k-armed bandit is a theoretical problem, where instead of one lever, the slot machine has k levers. Some levers might be better than others, but the given rewards are random by nature. By pulling on the levers, how can we find the lever that gives the best rewards, without wasting effort on levers that appear not to yield good rewards? This is known as the exploitation-exploration problem (Browne et al., 2012).

It turns out that this problem can be solved using Upper Confidence Bounds (UCB) (Browne et al., 2012). An Upper Confidence Bound expresses the amount of

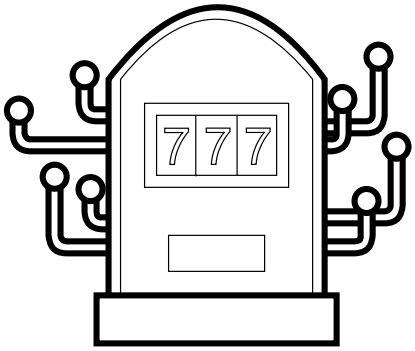

Figure 2: A multi-armed bandit

certainty we have in that the choice we are exploiting is indeed optimal. In other words an UCB is a value, which expresses, how much potential a choice has. Then if we play the arms with the most potential at each point, we will find the best arm in an optimal number of tries.

One simple way to calculate an Upper Confidence Bound is known as UCB1 (Browne et al., 2012).

$$UCB1 = R_j + 2C_p\sqrt{\frac{2\ln n}{n_j}} \tag{1}$$

Here R_j expresses the average reward given by the choice j. As the value of R_j increases the UCB gets higher and higher. This encourages exploitation of known good choices.

n is the number of choices made so far and n_j is the number of times choice j has been made. As choice j gets made more and more, the right term of the UCB equation gets smaller and smaller. This gives more weight to other choices than j and encourages exploration. The right hand term of UCB1 makes it explore each choice at least once. C_p is a constant parameter used for balancing between exploitation and exploration.

2.3 Upper Confidence Bounds applied to Trees

As stated before, the basic version of the MCTS algorithm always chooses the child with the highest expected reward, based on simulations. However, choosing an action at a given state is actually a problem very similar to a k-armed bandit. So instead of choosing the child that gives the highest reward, it would make sense to pick the child with the most potential or, in other words, the highest Upper Confidence Bound.

Upper Confidence Bounds applied to Trees (UCT) (also later referred to as Upper Confidence Trees) does this by replacing the tree policy with UCB1. One can imagine replacing each state with an k-armed bandit, the arms being the actions that can be taken. This makes the UCT algorithm explore new choices, while also exploiting known good choices. (Kocsis and Szepesvári, 2006)

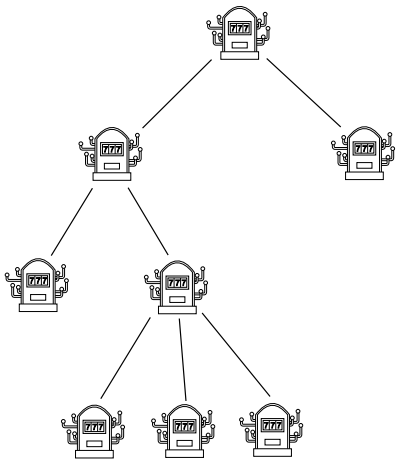

Figure 3: UCT algorithm

A popular extension to the UCT algorithm is Rapid Action Value Estimates (RAVE). RAVE is a all-moves-as-first heuristic, which assumes that all moves have some kind of inherent value associated with them regardless of when they are played (Gelly and Silver, 2011). In other words a bad action at late game probably is a bad action at the start too.

2.4 Limitations

The MCTS and UCT algorithm have important limitations. In particular the basic MCTS algorithm performs poorly when tree width or depth grow too large (Browne et al., 2012). In addition the UCT algorithm requires each action to be explored at least once as the tree is searched in UCB order. This makes the UCT algorithm degenerate into pure exploration as the size of the action space increases (Rolet et al., 2009). This is exactly what happens when moving towards continuous action spaces. RAVE also relies on the fact that each action can be sampled at least once and thus cannot be used in continuous action spaces (Couetoux et al., 2011a).

3 Literary Survey of MCTS Extensions

While the basic MCTS algorithm is not suitable for continuous domains, it can be modified and combined with other techniques to overcome its limitations. The following section will explore different approaches to extending the MCTS algorithm for decision making in continuous action spaces.

3.1 Manual/Informed discretization

The most intuitive way to extend the algorithm to work in continuous action spaces is to discretize the space into a finite one. We will first explore the use of manual discretization of action spaces. Then methods that use additional information to make better discretizations are explored. In essence, the following methods focus on how the splits in the action space are made.

3.1.1 Discrete Upper Confidence Bounds applied to Trees

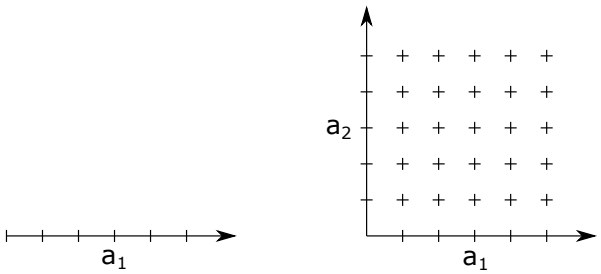

Figure 4: Exponential growth of discretized action space as dimensionality increases

As stated, the discretization of the action space can be done manually. While manual discretization can be applied on the standard MCTS algorithm, it is most often applied to the UCT algorithm. How coarse or fine grained the discretization is directly affects the size of the search tree (Fischer et al., 2015). Too fine discretization leads to a tree that is too large to explore, while too few can lead to an optimal strategy not being found (Van den Broeck and Driessens, 2011). Discretization can be done uniformly, randomly or with domain knowledge and it can directly have an effect on the performance of the algorithm. In this paper Discrete-UCT will refer to uniformly discretized UCT algorithm.

As the dimensionality of the action space grows, the number of samples increases exponentially. Too many samples make the action space too large to explore effectively and MCTS quickly degrades into exploration only. This renders linear discretization of the action space unusable in high dimensional cases (Mansley et al., 2011).

Therefore in practical applications using manual discretization, the dimensionality of the action space has to be kept relatively low. This can be done using abstractions, macro-actions and other tricks (Cowling et al., 2013). For controlling a car, for example, the action space can be abstracted into two continuous values: the speed of the wheels and the angle of the steering wheel. Then these two actions can be

discretized linearly to produce a search tree of appropriate size (Fischer et al., 2015). It would seem that manual discretization works best in low dimensional action spaces, where the action space can be discretized finely enough.

3.1.2 Tree Learning Search

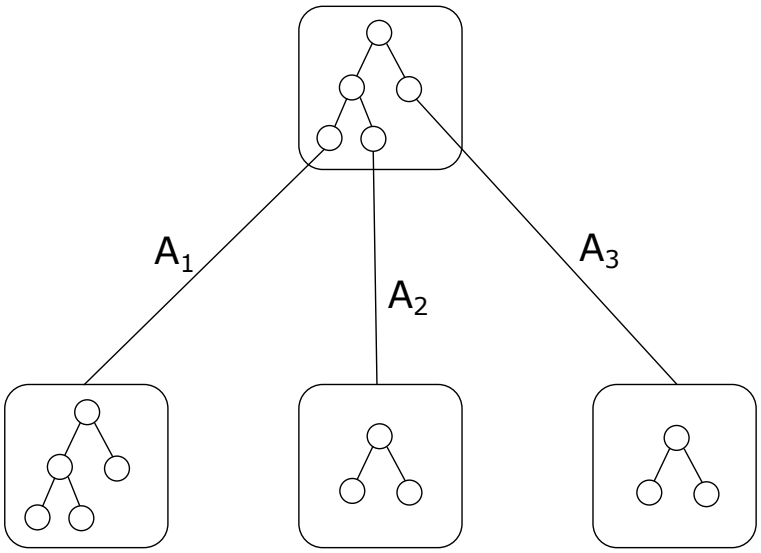

Figure 5: TLS algorithm uses regression trees at each state. Regression trees associate a range of actions A_i to each sub tree that further divides it.

To overcome the problems with manual discretization of the action space, more elegant methods can be used. In data stream mining a continuous stream of samples is used to build a regression tree to learn a model. One algorithm for building a regression tree is known as Very Fast Decision Tree learner (VFDT). MCTS can be used to provide the samples to a tree induction algorithm such as VFDT to build a regression tree. (Van den Broeck and Driessens, 2011). The VFDT algorithm uses a number of tests to find an optimal split in a continuous variable. As soon as a tests is found that splits the variable, a split is introduced and new nodes are added to the tree.

Tree Learning Search (TLS) is an extension to UCT. Instead of having k-armed bandits at each state, a regression tree is built on each state instead. The regression trees are used to divide the continuous action spaces into ranges of actions by making optimal split. As soon as an optimal split is found, the sub-optimal range is never sampled again. This causes some problems, when an internal node is split and an entire sub-tree of the MCTS tree is essentially discarded.

These problems cause the TLS algorithm to perform poorly. Experimental validation has shown that it works in low dimensional continuous domains. However, in more high dimensional and complex domains the standard implementation of TLS is outperformed by discretized standard MCTS algorithms. (Van den Broeck and Driessens, 2011)

3.1.3 Hierarchical Optimistic Optimization applied to Trees

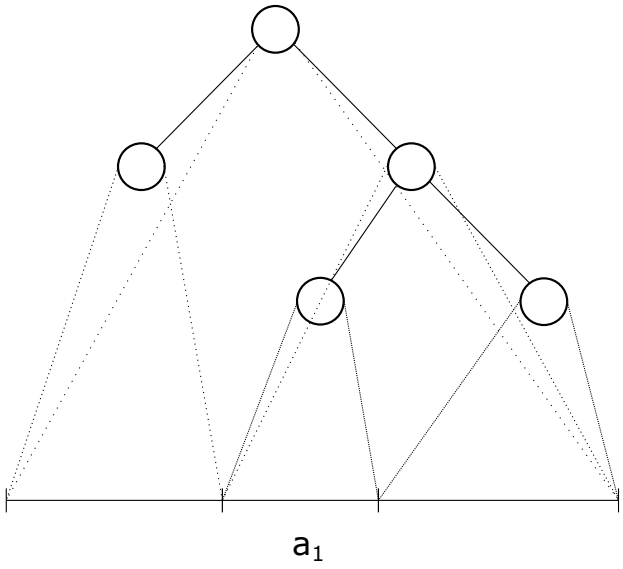

Figure 6: HOO decision tree

Hierarchical Optimistic Optimization (HOO) is a bandit algorithm that can be used for automatic discretization of an action space. It works by building a decision tree that divides the action space into ranges of actions (Mansley et al., 2011). The nodes of the decision tree represent ranges of actions in the action space and each time a new action is taken, the tree is searched for the most promising range of actions. Then a new action can be taken from the range. After taking an action the range is then further divided in to two and two new nodes are added to the decision tree.

To find the most promising ranges of actions, special B-values are calculated for each range. These values are calculated from a reward estimate and the B-values of the children nodes. Then selection of an action range starts at the root of the decision tree and continues to the leaf nodes of the trees, following the largest B-values. Unlike TLS, the algorithm can sample the same regions again, if the B-values are largest there.

Hierarchical Optimistic Optimization applied to Trees (HOOT) then expands on the UCT algorithm. It works by replacing the tree policy of the UCT algorithm with HOO. Instead of a k-armed bandit, each state has a HOO decision tree, which knows the best actions to take. Otherwise the algorithm is the same as standard UCT.

Building the HOO decision trees is computationally expensive (Mansley et al., 2011). However as the dimensions of the action space grow, the HOOT algorithm starts to outperform the linearly discretized UCT algorithm. This makes the HOOT algorithm more viable in high dimensional and continuous action spaces. However it requires that the action space is able to be sampled effectively. This means that it needs to be compact and have well defined boundaries (Couetoux, 2013). In addition HOOT only works in discrete state spaces (Weinstein and Littman, 2012)

Algorithm 2 Simple Progressive Widening Tree Policy

Input: start node v, alpha α
Output: action a_s
if $VisitCount(v)^{\alpha} > Children(v)$ **then**
  $a_s \leftarrow$ SampleAction(v)
  Actions(v) $\leftarrow a_s$
  Children(v) $\leftarrow$ Expand(v, a_s)
else
  $a_s \leftarrow$ UCB(v)
end if
return a_s

Extensions to HOOT such as Weighted HOOT (WHOOT) and Hierarchical Open-Loop Optimistic Planning (HOLOP) also work in continuous state spaces (Weinstein and Littman, 2012).

3.2 Unpruning

The second approach to extending to continuous action spaces is to use unpruning. In tree algorithms, it is common to prune computational trees for improved performance. For example in MCTS actions could be removed if they prove unsuccessful. Unpruning, however, is doing the opposite: start the search with only a limited number of actions and then add more actions as the search progresses. Instead of focusing on how the discretization is conducted, these methods focus on when the discretization should be conducted and elaborated on.

3.2.1 Simple Progressive Widening

Simple Progressive Widening (sometimes SPW, PW-UCT or just Progressive Widening) is an extension to the UCT algorithm. It extends the algorithm to work in continuous action spaces using unpruning. It is a modification to the selection method of the UCT algorithm, or in other words, to the tree policy.

In Simple Progressive Widening nodes keep track on how many actions have been explored at each iteration t and each state s has a corresponding a action set $A(s)_t$. Initially the explored action set is empty: $A(s)_0 = \emptyset$. Instead of always choosing the most promising child using Upper Confidence Bounds, it sometimes expands on the explored action set $A(s)_t$ of most visited nodes (Couetoux, 2013).

Now the most visited nodes get a progressively growing action space, while nodes with few visits only sample a few actions. The extension of the action space can be done by adding random actions or using domain knowledge for a specific order for actions to be added (Yee et al., 2016).

There is an important limitation to Simple Progressive Widening. It only works when an action a taken at state s leads only to finitely many states. When there is a degree of randomness such that taking action a at state s can lead to a continuous range of states, the algorithm does not work. Then the algorithm keeps adding

actions to the root node and does not grow the tree depth beyond 1 (Couetoux, 2013).

3.2.2 Double Progressive Widening

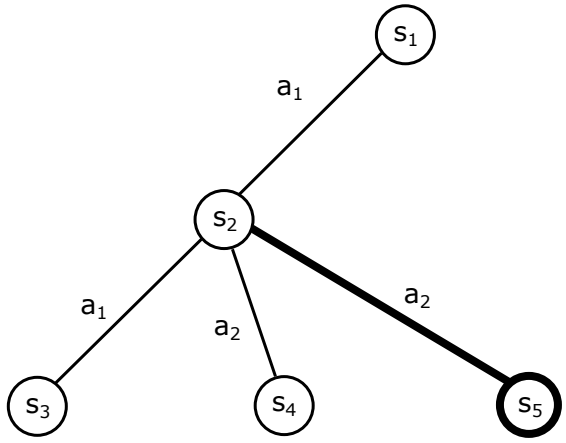

Figure 7: In Double Progressive Widening the same action can be sampled again to get a different outcome. In this case the action a_2 is sampled twice from state s_2 to get states s_4 and s_5

Double Progressive Widening is a further extension to Simple Progressive Widening. It makes the algorithm work with stochastic transitions in continuous state spaces (Couetoux, 2013). Much in the same way as actions are added to action sets, now state transitions are sampled more often as the nodes are visited more. This means adding more possible outcomes to an already sampled action in and already explored state as seen in Figure 7.

At the start, the algorithm works in the same way, an action is decided or a new action is added to the action set. After the action a has been decided, a second sampling is done to account for the randomness of the transitions from one state to the next.

Similarly to when unpruning actions, if a certain threshold is exceeded a new sample is added. In other words this already sampled action at this state (a, s) now creates a new child, with a different outcome. If the threshold is not exceeded one of the previous samples is selected. This is done in such a manner that the most probable child is selected.

The performance of Double Progressive Widening can be increased using Blind Value estimates. These can improve performance up to by a factor of 10 (Couetoux, 2013). Continuous Rapid Value Estimates (cRAVE) is an extension of RAVE to continuous action spaces (Couetoux et al., 2011b). cRAVE is an another heuristic that can improve the performance of progressive widening algorithms. It is also possible that in future work Blind Value estimates and cRAVE could be used in conjunction (Couetoux, 2013).

There is a further variation of the Double Progressive Widening algorithm known as Polynomial Upper Confidence Trees (PUCT) (Auger et al., 2013). In standard

Double Progressive Widening, the algorithm has logarithmic exploration, meaning that actions are added at increasingly rare intervals. PUCT algorithm uses polynomial exploration instead.

3.2.3 Voronoi Optimistic Optimization applied to Trees

Similarly to Hierarchical Optimistic Optimization, an optimization method called Voronoi partitioning can be used in conjunction with MCTS. Voronoi Optimistic Optimization applied to Trees (VOOT) (Kim et al., 2020) is in many ways similar to the previously mentioned HOOT algorithm.

However, instead of expanding on the basic UCT algorithm, VOOT is based on the Progressive Widening algorithm. While Progressive Widening samples new actions to be added to the action sets of the nodes at random, the VOOT algorithm uses Voronoi partitions for sampling these actions. The exploration-exploitation problem that is typically solved using UCB, is here solved instead by sampling the Voronoi partitions. In particular, the best partition is known and it can be sampled for exploitation, while exploration can be done by sampling the remaining Voronoi partitions.

VOOT was compared against Progressive Widening and many other non-MCTS decision making algorithms including CMA-ES and PPO (Kim et al., 2020). The comparison was done in function optimization domains. In addition the algorithms were compared in two realistic robot planning problems with sparse rewards, high dimensional action space and non-linearities. VOOT was shown to outperform both Progressive Widening and also all the other algorithms.

3.2.4 Monte Carlo Tree Search with Variable Simulation Periods

Variable Simulation Period Monte Carlo Tree Search (VSP-MCTS) (Ba et al., 2019) is an algorithm that combines both Progressive Widening and HOOT algorithms into a novel approach by utilizing variable simulation periods. Variable simulation periods here refer to the amount of time that can be used for MCTS simulation, before an action needs to be selected in online planning. What the paper finds is that in some environments it can be beneficial to continue with the same action for a longer period of time and instead of deciding on a new action, to use this time for more MCTS simulations. In other environments this can however be harmful, so the simulation period needs to be considered as a variable in the tree search.

This is done by using the Progressive Widening as the base of the algorithm. Each action is coupled with and associated simulation period τ, effectively adding another dimension to the search of the action space. In order to explore these pairs at each state of the MCTS tree, each node has an associated HOO tree, which is queried for the most promising action a and simulation period τ pair (a, τ).

The algorithm is shown to outperform both the Progressive Widening algorithm and Progressive Widening combined with HOOT. The comparison is done in two low-dimensional continuous Open AI Gym environments. The paper notes that VSP-MCTS performs poorly in environments with stochastic properties.

3.3 Action optimization

One way for successfully dividing a continuous action space is to utilize known information about the actions themselves. This information can be used to pick better actions or to optimize the actions further. In essence, the focus is on optimizing the actions themselves, even after they have been decided and taken.

3.3.1 Kernel Regression Upper Confidence Trees

Kernel regression is a method in mathematics for estimating the expectation of a random variable. In Kernel Regression UCT (KR-UCT), information is shared between similar actions by using kernel regression.

In it's simplest form, the expected reward r of a random action a_x could be the average of all the actions taken so far, weighted by the distance to the action a_x we are estimating. This can be written as $E(r|a_x)$. The weighting of the actions is done with a function, which is known as a kernel. Typically kernels are non-linear.

KR-UCT uses a modified UCB selection policy for its tree policy.

$$UCB_K = E(r|a_j) + 2C_p\sqrt{\frac{2\ln W(n)}{W(n_j)}} \tag{2}$$

The average reward of choice j is now represented by the estimated value $E(r|a_j)$. The number of choices n and number of times choice j has been made n_j are now weighted. Weighting of choices $W(n)$ is done with the kernel function.

Actions are added to nodes as they are visited more often, just like in Progressive Widening (Yee et al., 2016). Instead of initializing the explored action sets $A(s)_0$ to empty sets, some initial set of actions is required. This set can be initialized manually, by using domain knowledge. The algorithm can eventually improve on this initial set of actions by adding new ones to it.

KR-UCT can be used in continuous action and state spaces. In addition it works with stochastic transformations. When testing in a low dimensional continuous action and state space, KR-UCT outperforms both HOOT and progressive widening techniques, even when they are equipped with cRAVE heuristics (Yee et al., 2016).

3.3.2 Value-Gradient Upper Confidence Trees

Value-Gradient UCT (VG-UCT) (Lee et al., 2020) is an extension method that can be used on several UCT variants. It works by improving coarse grained discretization of actions by using a mathematical concept known as value gradients. Like with many other gradient based mathematical methods, the gradient of a function can be used to guide the search for finding the maximum of said function. However in complex domains basing the search purely on the gradient can guide the search to a local optimum, even when there are better solutions available.

The basic idea in VG-UCT is to first use UCT for searching the action space for promising branches in trajectory optimization. Then value gradients can be calculated for each of the actions in the nodes of a promising branch and the actions

can be slightly adjusted towards the direction of the gradient. This makes the algorithm exploit both UCT for coarse grained global search of the action space and value gradients for optimizing the best solutions locally - effectively negating the drawbacks of both methods.

In order to calculate value gradients, the planning objective is formulated as a sequence of policies π_0 ... π_T, which given a starting state s_0 yields the maximum cumulative reward. Then differentiating the planning objective with regard to state s_t and respective policy π_t yields the respective value gradient. Because these computations can be quite demanding and more importantly, because often times a simulators do not give access to the information required for calculating value gradients, they can be calculated with various approximations for the same results.

One problem with changing values of already taken actions is the possibility of invalidating the actions that come later in the branch. This could easily be done if for example the first action needed to be exactly correct for avoiding early termination in a learning problem. Therefore the adjustments to the actions need to be small. This problem can be examined mathematically by assuming Lipschitz continuity of the reward function and transition function. Then maximum allowable perturbations to actions can be calculated.

VG-UCT is shown to outperform Discrete-UCT, HOLOP and KR-UCT. In addition VG-UCT can be combined with kernel regression for VG-KR-UCT, which can yield an even higher performance. The testing was done with multiple MuJoCo learning problem environments with continuous action spaces with a moderately high action space dimensionality.

3.4 Policy optimization

Fourth approach to extending the search to continuous domains is to use machine learning. Instead of choosing children based on a simple metric or simulating them forward with random actions, more elegant strategies can be used for policies. Machine learning can be used to improve tree policies, so they work in large or continuous action spaces. In addition it can be used for sampling actions with improved default policies.

3.4.1 Continuous AlphaZero

It has been proven that AlphaZero works in truly enormous action spaces, like the one in Go (Silver et al., 2017). However, there seems to be only preliminary research on making it work on continuous action spaces. At the core of the discrete AlphaZero algorithm are two neural networks. The policy network π_p is used to select moves and it is used as a factor in the tree policy (Moerland et al., 2018). The value network π_v is used to evaluate board positions and can be used instead of the default policy to simulate games.

In the selection phase a modification of the basic UCB formula is used. The UCB term is scaled with the probability distribution of the actions that can be taken from this state. The idea is to take in data from the policy network π_p to make more

informed decisions on which nodes to expand. The expand phase itself works in the same way as standard MCTS.

In the simulation phase, instead of using a default policy to simulate an end result, the value network π_v is used instead. The value network is trained to give an estimate of the end result directly from the board position. The value network can be trained through self play. The backpropagation step is done in a similar fashion as standard MCTS.

Continuous AlphaZero (A0C) algorithm differs from the standard AlphaZero in three ways (Moerland et al., 2018). Firstly an extension method to the MCTS algorithm for continuous domains is needed. Secondly the algorithm has to deal with continuous probability distributions now. Thirdly the training methods need to be altered to work in continuous domains.

To use MCTS in continuous domains an already established method can be used. Progressive Widening can be used as a base for extending the AlphaZero algorithm to continuous domains (Moerland et al., 2018). Scaling continuous probability distributions in the selection phase can be problematic. One solution to this is to use the policy network to decide which actions to add in the Progressive Widening (Moerland et al., 2018). This effectively avoids the problem of scaling and has the same effect: the policy network steers the search. The training methods presented in the standard AlphaZero algorithm need to be changed. Policy network specification, policy target calculation and training loss need to be modified. Value network only needs a minor modification.

There has been little experimental validation of the A0C algorithm in continuous domains (Moerland et al., 2018). It seems that the algorithm can indeed learn to do continuous tasks. Experimental validation has been done using the inverted pendulum example, which is a low dimensional and simple continuous domain problem.

3.4.2 Fixed Depth Informed Monte Carlo Tree Search

Fixed Depth Informed Monte Carlo Tree Search (FDI-MCTS) is a quite heavily modified variant of standard MCTS (Rajamaki and Hamalainen, 2018). The core idea is to use machine learning to learn trajectories, which can be used to guide the search. A trajectory is simply a string of actions to be completed in sequence. Trajectories can be learned using neural networks from a state s, taking actions for T time steps. The amount of time steps taken T is also known as the planning horizon.

The algorithm uses multiple different trajectories utilizing for example machine learning models and previous iteration best trajectories. These trajectories are simulated in parallel and it means that the tree grows in every leaf node in each iteration cycle, unlike default MCTS.

The selection phase of the FDI-MCTS algorithm is quite simple. In each selection phase all the leaf nodes are expanded. Each node is expanded using the trajectory they have been using to this point. In other words each trajectory continues from the node it expanded in the previous time step.

In the expansion phase then, for each node an action is sampled from the trajectory they are using. This means that for example a machine learning trajectory, a new

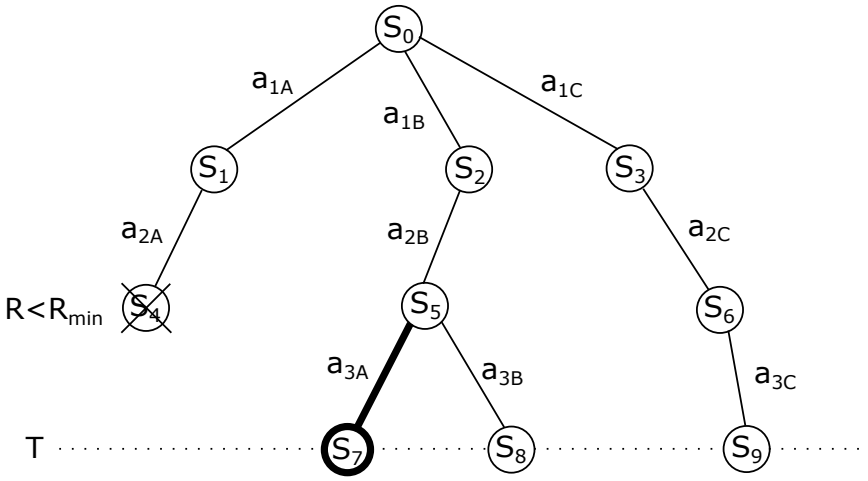

Figure 8: FDI-MCTS expands on each leaf node at every expansion phase. Rewards are calculated for each added state. At state S_4, trajectory $A = (a_{1A}, a_{2A}, a_{3A})$ is pruned and forked to continue from node S_5, before reaching the planning horizon T. Note that node S_5 was originally reached from trajectory B

action is picked by the neural network. This action is then simulated and a new child is added to the tree.

When the new node is added to the tree, a reward is calculated for it. This is done without an explicit simulation phase simulating it forward to a terminal state. It is instead done from the state directly using some function. These rewards can be accumulated through the trajectory or calculated for each state separately.

The back-propagation step is not done at all until the trajectories reach the planning horizon. Then if the rewards were not accumulated previously, they need to be backpropagated to sum up the rewards of the trajectory. When the rewards of a trajectory are summed up, a best trajectory can be picked by comparing the total rewards.

While the algorithm is iterating, a reward threshold can be used to prune bad trajectories before they reach the planning horizon. This means that if in the selection phase the reward is noticed to be under a certain threshold, the trajectory will be pruned. Then instead of continuing the trajectory from this node, it is continued from a node belonging to a more successful trajectory as seen in Figure 8. This ensures that all trajectories reach the planning horizon and there is always a selection of good trajectories to pick from.

After the algorithm has continued for T time steps and reached the planning horizon, the best trajectory can be picked. It is the one with the highest accumulated reward. Then the first action of this trajectory can be picked to be used and the algorithm can be restarted.

FDI-MCTS has been experimentally validated to work in high dimensional continuous action spaces (Rajamaki and Hamalainen, 2018). However, its performance has not been compared to other extension methods.

3.4.3 Tree search Policy Optimization

Tree search Policy Optimization (TPO) is an extension method based on the previously mentioned PUCT algorithm (Yazdanbakhsh et al., 2019). It uses reinforcement learning to allow the tree search to work in continuous action spaces. In particular it uses reinforcement learning to create a pre-trained tree policy for choosing which actions are to be taken at each node of the tree search. In return, the generated search tree can then be used to create high quality trajectories in the action space, allowing for the tree policy to be trained further using these trajectories as training samples. The reinforcement learning algorithm used here is called Proximal Policy Optimization (PPO)

The algorithm works in two stages: on-policy training and off-policy training. First the PPO policy is trained by generating trajectories through the action space. This is the on-policy part as it optimizes the policy based on the samples generated by the PPO policy itself. The authors found that this part is critical for the performance of the algorithm, as a pre-trained tree policy would help cut down on the branching factor of the tree search.

After the policy has been pre-trained, the tree search can begin. The tree search is guided by the PPO policy and it generates high quality trajectories. These trajectories can then be used to train the policy further. This is called off-policy optimization as these trajectories are generated by the tree search instead of the PPO algorithm itself. Using the samples generated by the MCTS is non-trivial and uses a newly defined loss function based on MCTS statistics as described by the authors. The authors also used a custom built distributed architecture to run the algorithm efficiently.

The authors find that the TPO algorithm performs well in many OpenAI gym and MuJoCo algorithms with a high dimensionality such as Ant-v2 and Humanoid-v2. The TPO algorithm was found to produce higher quality trajectories than pure PPO at the cost of more evaluations due to the MCTS component. However, the TPO algorithm performance was not compared to other MCTS extension methods.

3.5 Summary of literary survey

The literary survey found many prominent extension methods to the MCTS algorithm. In this overview they were categorized into: Manual/Informed Discretization, Unpruning, Action Optimization and Policy Optimization.

3.5.1 Findings

Discretization is the most intuitive way to extend the MCTS algorithm. Manual discretization is a commonly used way to extend into continuous domains, but it can only be used in specific cases and low dimensional action spaces. Tree Learning Search is a method to automate discretization, but it performs poorly, when tested in practice. HOOT and variations of HOOT are claimed to be the most effective methods in discretization algorithms and they can be used effectively in low dimensional continuous domains.

Unpruning methods present another way for approaching continuous domains. Progressive Widening methods are very effective and are shown outperform the HOOT algorithm in continuous domains. Voronoi optimization can be combined with Progressive Widening for an effective method, but performance remains unproven in continuous control settings. Monte Carlo Tree Search with Variable simulation periods adds the time the same action is taken as another dimension in the tree search. It is shown to outperform Progressive Widening and HOOT methods.

Action optimization methods focus on local optimization of actions. The KR-MCTS algorithm is even more effective than HOLOP and HOOT variations, but it not as commonly used. The more recent VG-UCT performs even better and can be combined with Kernel Regression. However, the performance has not been compared against unpruning methods such as Progressive Widening.

Machine learning is used heavily in policy optimization methods. A0C is an extension to the algorithm used in AlphaZero. A0C uses Progressive Widening as its base and builds further upon it using machine learning. However, A0C is still a novel algorithm and lacks experimental validation as it only works in single dimensional spaces as defined in the paper. FDI-MCTS is a machine learning based method which has been experimentally proven to work in high dimensional action spaces. TPO is another method that uses machine learning, which is shown to also perform in high dimensional continuous control problems.

3.5.2 Analysis

While there have been other studies on different MCTS algorithm variations such as (Browne et al., 2012), no studies comparing extension methods to continuous action spaces were found. The search process was conducted systematically and it is presented in section 1.2. It is possible that some methods were missed, but that would however suggest that the method is not prominent. The focus was on prominent methods that are in use. To the knowledge of the authors, MCTS extension methods have not been previously listed and categorized. Categorization presented here is, while being somewhat arbitrary, also a novel contribution to the body of MCTS knowledge.

4 Benchmarking MCTS Extensions

The empirical section of this thesis consists of benchmarking MCTS extensions against each other. The following section describes the reasoning in selecting algorithms to test and the experimental setup used for benchmarking them. The results of the study are also presented here.

4.1 Selection of algorithms

Based on the findings of the literary survey, a subset of the MCTS extensions was selected to be compared in a performance study.

4.1.1 MCTS Variants

The following criteria were used to choose the algorithms to be included in the study.

- Firstly, they would have to be representative and relevant of the body of MCTS research. The most prominent variant was chosen from each category of approaches to get a wide variety of differing extension methods. In addition the variants in the same categories had typically already been compared with each other.
- Secondly, the availability of code and complexity of the algorithms needed to be taken into consideration. It turned out that almost none of the variants introduced here had source code available, so considerable work would need to be allocated for implementation. Because many of the variants were quite involved in their implementation, especially variants based on machine learning, many could not be implemented based on the research papers alone in the given time frame.
- Thirdly, in order to guarantee a fair comparison between the algorithms, they had to work in general continuous control problems. Some extension methods discovered in the literary survey were developed with a specific problem in mind and therefore could not be run in a general environment against other algorithms.

Initially six algorithms were selected for the study. However, only three algorithms worked without problems in the final study. In particular the HOOT implementation was discarded, because of its complexity and vague description of implementation details in the original paper. The TPO algorithm was quite interesting for comparison, but included several machine learning components and a custom hardware setup as described in (Yazdanbakhsh et al., 2019), resulting in unreasonable amount of effort.

A0C was deemed easier to implement as source code could be found for the discrete version of the AlphaZero algorithm by original authors (Silver et al., 2017). The first approach modified this code to implement the A0C algorithm, but proved problematic through the use of deprecated TensorFlow library version. In addition a more recent approach to A0C implementation was found, but was limited to one

dimensional action spaces. Ultimately A0C was excluded as the original paper describes the algorithm only in one dimension and therefore extension to multiple dimensions was out of scope for the original paper. The algorithms included in final study are listed here:

- Discrete-UCT
- HOOT (not complete)
- Progressive Widening
- VG-UCT
- TPO (not complete)
- A0C (limited to one dimension, excluded)

4.1.2 Random Branching Depth

In addition to the algorithms above, two other algorithms were included in the study. The first baseline algorithm, which the MCTS variants are compared against is a simple novel algorithm referred here as the Random Branching Depth algorithm. It simply uses UCB to guide where Monte Carlo rollouts should be conducted from, without the need to discretize the search space. It was empirically validated that the algorithm performs reasonably well in trajectory optimization in continuous control tasks.

Because the Monte Carlo Tree Search methods were not originally designed for continuous control tasks, it raises the question whether expanding and iterating on a relatively shallow tree structure which guides the rollout simulations is worth the computational load or if the computational budget would be better used on deeper exploration of the state space with rollout simulations.

The algorithm is motivated by the fact that the MCTS tree search is guided by essentially the rollout simulation returns. Instead of starting the rollouts from a relatively shallow tree, they could just as effectively be started from paths taken by previous rollouts. By uniformly dividing the branches, instead of always starting from the MCTS tree, the ends of the search tree branches would then be more effectively guided.

The Random Branching Depth algorithm also constructs the search tree structure with much lower code complexity compared to more involved MCTS variants. Therefore it should provide an interesting comparison point for MCTS extension methods in continuous action spaces.

4.1.3 Covariance Matrix Adaptation Evolution Strategy

In order to benchmark the performance of the MCTS variants further, they were also compared to a state of the art decision making algorithm called Covariance Matrix Adaptation Evolution Strategy (CMA-ES). It is an evolutionary optimization

Algorithm 3 RandomBranchingDepth

$v_0 \leftarrow$ create root node v_0 with state s_0
run N initial rollouts from v_0 until termination or planning horizon
while withing budget **do**
    $d \leftarrow$ initial depth d between [1, MaxDepthSoFar]
    traverse the tree using UCB until at node v_d
    run rollout from v_d until termination or planning horizon
end while
return $a \leftarrow$ BestChild(v_0)

algorithm that can be used for a variety of purposes. In recent years it has been shown that CMA-ES can be used very effectively in trajectory optimization in continuous decision making problems, like in (Naderi et al., 2017).

While the CMA-ES algorithm on its own is not well suited for sequential decision making problems, the problem can be represented in such a form, where it can be optimized with the algorithm, just like any other function. The continuous control environments can be turned into a function f that take in general d dimensional input as a vector X_d and then return a reward $R = f(X_d)$. To make a continuous control problem fit in this description, the entire trajectory of actions can be concatenated into a single high-dimensional input vector and then given to the function f, where it will be evaluated in the optimization environment. The dimensionality of the problem represented in this manner increases based on the length of the trajectory or, in other words, the desired planning horizon. In addition a larger action dimensionality also increases the dimensionality of the problem.

As an example, given that we want a trajectory of t actions in a n dimensional continuous control problem, the dimensionality d of the problem will be $d = t * n$. Then the CMA-ES algorithm can be given a function $f()$ with the input X_d to minimize. One of the objectives of this performance study is to investigate whether MCTS algorithms are better at handling the rise in dimensionality, or whether one could get similar or better results with CMA-ES even at high problem dimensions.

4.2 Experimental setup

The benchmarking program and implementation specifics are discussed here.

4.2.1 Open AI Gym and MuJoCo

The performance of the algorithms is tested with the Open AI Gym library (Brockman et al., 2016), which can be used to run MuJoCo environments (Todorov et al., 2012). The MuJoCo physics engine comes with multiple pre-made continuous control tasks for research and development. Tasks vary from simple low-dimensional ones to complex high-dimensional ones. For example, the objective of the InvertedPendulum-v2 task is to balance a pole on a moving cart. It is a one dimensional continuous problem, as the control is only on how much force left or right is applied to the cart. An example of a high dimensional complex task is Humanoid-v2. In this environment the task is to make a 3d human ragdoll learn how to walk by applying torques to the joints of the doll. This task has a 17-dimensional continuous action space. The environments picked for the study and their dimensions are listed in Table 2.

4.2.2 Offline vs. Online planning

An important distinction in computational decision making can be made between offline and online planning. Planning offline means that the whole problem needs to be solved from the initial state. This means formulating a plan, or a trajectory through the action and state spaces of the learning problem all the way to an adequate

Environment	Action space dimensions
Ant-v2	8
HalfCheetah-v2	6
Walker2d-v2	6
Hopper-v2	3
Swimmer-v2	2
InvertedPendulum-v2	1

Table 2: MuJoCo environment action space dimensions

solution. Offline planning typically takes more computational resources than online planning, as the whole solution will need to be known before executing the solution.

Online planning means that planning can continue during the execution of the trajectory. This means that the final solution is not known before starting the execution. One can for example plan ahead a certain number of turns or a certain time horizon. Planning during execution typically means tighter time constraints on the planner, since execution will typically need to continue after a certain amount of time.

MCTS can be used for both offline and online planning. It is in particular suitable for online planning as it is a anytime algorithm. This means that the execution can be stopped at any point when the next part of the solution is needed, making it work with tight time constraints. On the other hand if offline planning is an option, the tree search can be continued indefinitely for better and better solutions. When MCTS is used for offline planning, instead of utilizing only the most promising action from the root node of the tree, the best actions from the best nodes can be followed all the way to leaf nodes to form a trajectory. Even the rollout can be included in this trajectory as it is known that it arrives at a high reward solution.

4.2.3 Hyperparameter optimization

Many of the algorithms have parameters that can be adjusted based on the nature of the target problem and environment. In order to provide a fair comparison, the parameters need to be adjusted to approximately optimal settings for all of the algorithms. To achieve this, the parameter space was searched with a simple grid search to find optimal parameters for each algorithm. What this means in practice is discretizing the parameter space and searching all the different combinations of parameters for the ones that give the largest rewards. The grid search was done in two MuJoCo environments (HalfCheetah-v2 and Hopper-v2), which were typical to the environments used in the actual study with a horizon of two seconds. The parameter spaces and best found values are listed for each algorithm in Table 3

4.2.4 Benchmarking program

For evaluating the performance of selected algorithms, a benchmarking program was written with Python, utilizing the Open AI gym and MuJoCo libraries. Since

Algorithm	Parameter space
Discrete-UCT	• Number of samples per dimension [**2**, 3]
Progressive Widening	• Alpha [0.01, 0.25, **0.5**, 0.75, 0.99]
VG-UCT	• Alpha [0.01, 0.25, 0.5, **0.75**, 0.99] • Delta [0.01, 0.1, 0.5, **1.0**] • Step size [0.01, 0.1, 0.5, **1.0**]
Random Branching Depth	• Number of initial rollouts [2, 4, **8**, 16, 32]
CMA-ES	• Population size multiplier [0.5, 1.0, **2.0**, 4.0]

Table 3: Relevant parameters for each algorithm were picked to be optimized with grid search. Best parameters found are marked in **bold**.

available code for the MCTS extensions was mostly not available, implementations of the algorithms were written from scratch based on the papers. For CMA-ES implementation the cma library for Python was used.

Comparing the offline planning performance of the algorithms provides more consistent results and also a fair comparison to other non-MCTS algorithms, which might not be suited for online planning. While standard MCTS variants only provide one action as output, the algorithms can be modified to provide a set of best sequential actions until a certain state, or in other words a trajectory.

The key performance metric used in this study is the return of the best trajectory found by each algorithm, i.e., the sum of rewards over an action trajectory. Best trajectory return R_b here means the summed intermediate rewards of the best trajectory from the start state to the planning horizon or terminal state. Given the best trajectory of the run $A_t = \{a_0, a_1, \ldots, a_t\}$, associated states $S = \{s_0, s_1, \ldots, s_t\}$, state transition function f and associated reward R, the best trajectory return R_b is defined as in Equation 3. For reliable reporting of results, the returns were averaged over 10 independent optimization runs with different random seeds.

$$R_b = \sum_{n=0}^{t} R(f(s_n, a_n)) \tag{3}$$

The performance of three different MCTS variants of increasing complexity was evaluated and also compared to the performance of the Random Branching Depth and CMA-ES algorithms. The testing was done on MuJoCo physics environments

Environment	Action replay amount	Tree depth (1s)	Tree depth (2s)	Tree depth (4s)
Ant-v2	2	20	40	80
HalfCheetah-v2	2	20	40	80
Walker2d-v2	12	125	250	500
Hopper-v2	12	125	250	500
Swimmer-v2	2	25	50	100
InvertedPendulum-v2	2	25	50	100

Table 4: The environment time step size directly affects the action replay amount used. The time step size and planning horizon depth determine the maximum tree depth.

with varying degree of dimensionality. Performance was tested with three different planning horizon depths of one, two and four seconds. The length of the planning horizon directly controlled the max depth of the tree search.

To account for the difference in the length of the time steps in the environments, the action sampling frequency was kept constant at 0.10. In other words, the action to take could be switched 10 times a second regardless of environment. If the time step of the environment was smaller than the sampling frequency, the previous action was executed multiple times until another action would be decided by the algorithm. In the program this amount of duplicate steps was controlled by a parameter called action replay amount and it was set automatically based on the environment time step size. The action replay and tree depths can be seen in Table 4.

4.3 Results

Results of the empirical study are presented here and evaluated both quantitatively and qualitatively.

4.3.1 Quantitative evaluation

A performance evaluation in a single environment can be seen in Figure 9. The best trajectory return of all algorithms increases as more simulations steps are taken and better trajectories are found. All algorithms are run 10 times and best trajectory returns are averaged. Standard deviation is also calculated and displayed. Typically growth is fast at first, but then slows as better trajectories become harder to find. In Figure 9, Discrete-UCT is the worst performing MCTS variant. VG-UCT outperforms Discrete-UCT slightly. CMA-ES and Random Branching Depth outperform these MCTS variants, but perhaps surprisingly, Progressive Widening is the best performing MCTS variant and overall algorithm in this setting.

The performance of the algorithms was affected significantly by the chosen planning horizon depth. This can be seen in Figure 10. For example, with horizon of 1 second, VG-UCT was the best performing MCTS variant, while with a 2 second

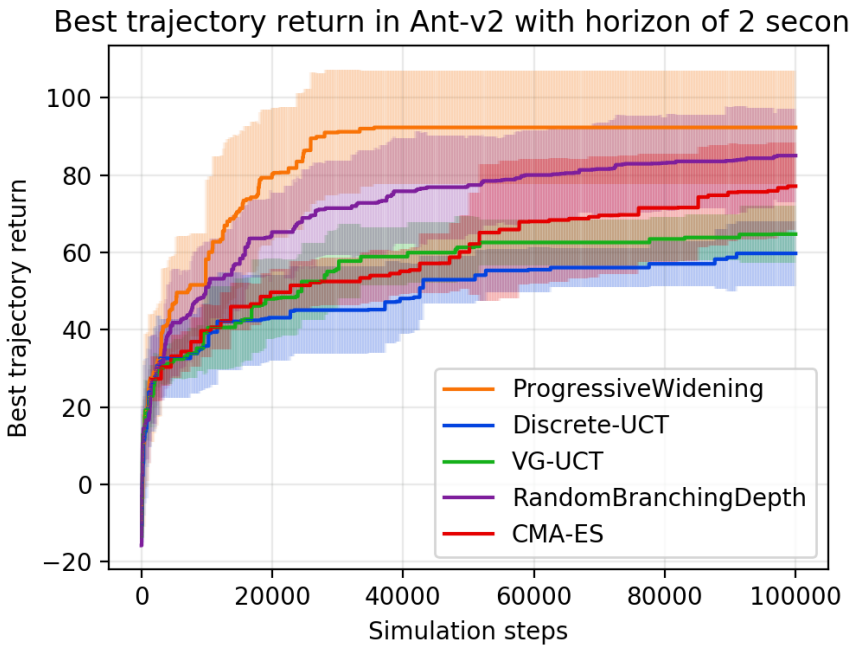

Figure 9: Best trajectory return mean and standard deviation as a function of the total simulation steps in the Ant-v2 environment.

horizon it was the worst performing one. It is noteworthy that the standard deviation, grew much larger on longer time horizons on all algorithms, but especially with the Discrete-UCT, Random Branching Depth and CMA-ES algorithms. The large variability on longer horizons could explain why the best performers could differ significantly from one problem setting to another.

The performance of the algorithms also varied significantly depending on the used environment as seen in Figure 11. Discrete-UCT had perhaps the most varied performance. In the Ant-v2 environment for example it was the weakest of the three MCTS variants. However, the Discrete-UCT performed quite well in HalfCheetah-v2, Walker2d-v2 and Swimmer-v2, where it performed the best of the MCTS variants. This is likely caused by the decrease in action space dimensionality as HalfCheetah-v2 and Walker2d-v2 have only 6 dimensions and Swimmer-v2 only 2 dimensions. In Hopper-v2 and InvertedPendulum-v2 environments the Discrete-UCT algorithm failed to output well-performing trajectories. This was most likely caused by the low sampling parameter of 2 samples per action dimension, which was determined by the hyperparameter optimization. In a single dimensional continuous environment like InvertedPendulum-v2 then, the algorithm has only two actions to pick from, leading to early termination as the pendulum quickly falls over and the algorithm fails to explore further from the root node, as it is already fully expanded.

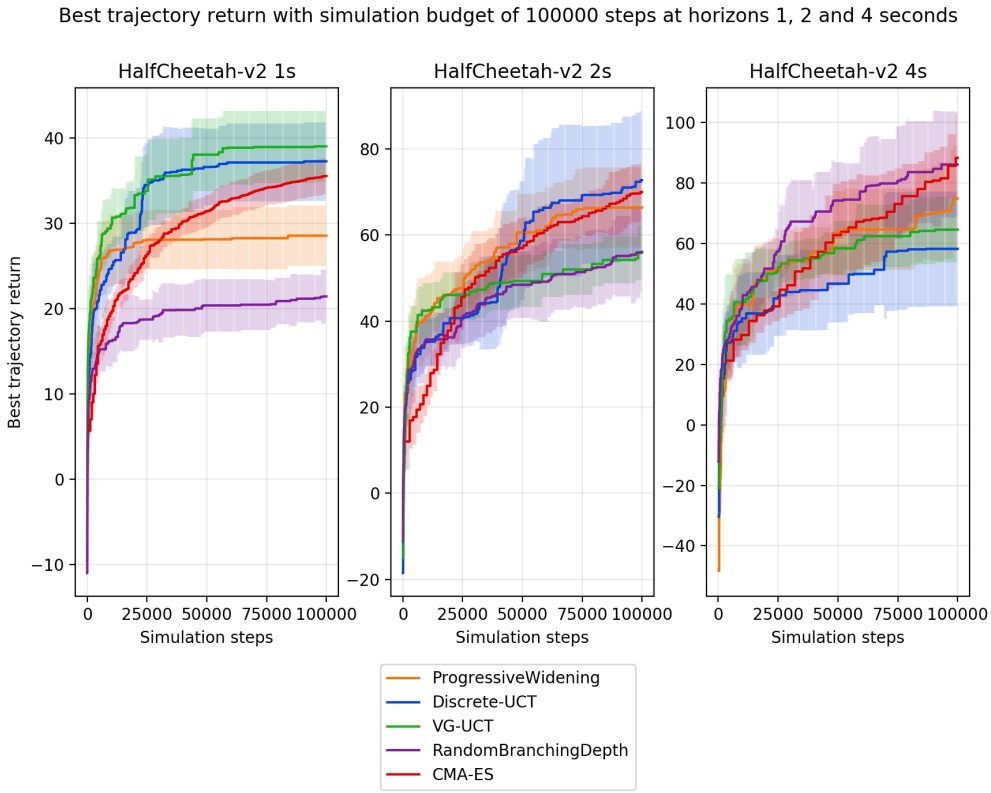

Figure 10: Algorithm performance in HalfCheetah-v2 environment with different planning horizons depths.

VG-UCT in most cases performed better than Discrete-UCT, outperforming it in environments Ant-v2, Hopper-v2, and InvertedPendulum-v2. This is consistent with results presented in the original paper (Lee et al., 2020). However, the VG-UCT algorithm was outperformed by all other variants in HalfCheetah-v2 and Walker2d-v2. The cause for this is unclear. The difference between worst performers in HalfCheetah-v2 in Figure 11 is slight and can be explained by randomness. It is also noteworthy that VG-UCT performs poorly both in Walker2d-v2 and Hopper-v2 environments, which both have a action replay value of 12. As the algorithm relies on slight adjustments of actions, it is possible that duplicating actions are the cause of its problems in these environments.

The Progressive Widening algorithm performed the best out of the three MCTS variants, by being the best variant in Ant-v2, Hopper-v2 and InvertedPendulum-v2 environments. Progressive Widening never failed to output well-performing trajectories in any of the tested environments, unlike Discrete-UCT for example.

Random Branching Depth algorithm performed quite well for being a simple algorithm. It outperformed all other algorithms in Walker2d-v2 and Hopper-v2 environments. Also noteworthy is the high variation in trajectory quality - particularly in these environments. It can be seen as the large standard deviation values in Figure 11.

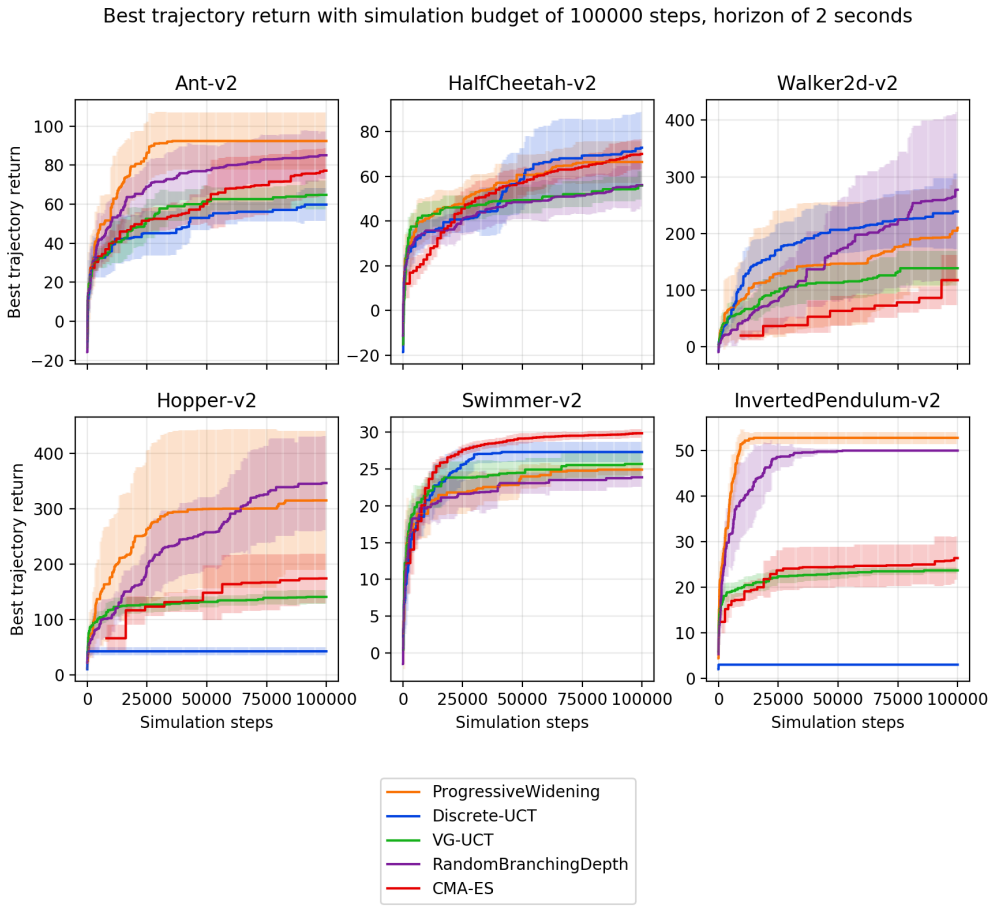

Figure 11: Performance of algorithm in all test environments.

CMA-ES algorithm performs well in many environments, but is the best performer only in Swimmer-v2 environment in Figure 11. Its performance is on par with other algorithms in most cases. Perhaps most surprising is its performance in the InvertedPendulum-v2, Walker2d-v2 and Hopper-v2 environments. All of these environments are prone to early termination and it possible that the CMA-ES algorithm was unable to adapt to this. In addition CMA-ES had a very low iteration count in Walker2d-v2 and Hopper-v2 environments, as a single function evaluation took 250 simulation steps at once. This was caused by the small time step size of the environment and the fact that CMA-ES only evaluates complete trajectories. This was a problem unique to CMA-ES, as MCTS variants would still iterate one action at a time even when tree depth was large.

When comparing MCTS performance across all environments and all time horizons (18 setups), Progressive Widening is the best MCTS variant, taking first place in nine of the setups. Discrete-UCT is the best performing MCTS variant in six setups while VG-UCT performed the best in only three setups. The performance in all setups is included in Appendix A.

4.3.2 Qualitative evaluation

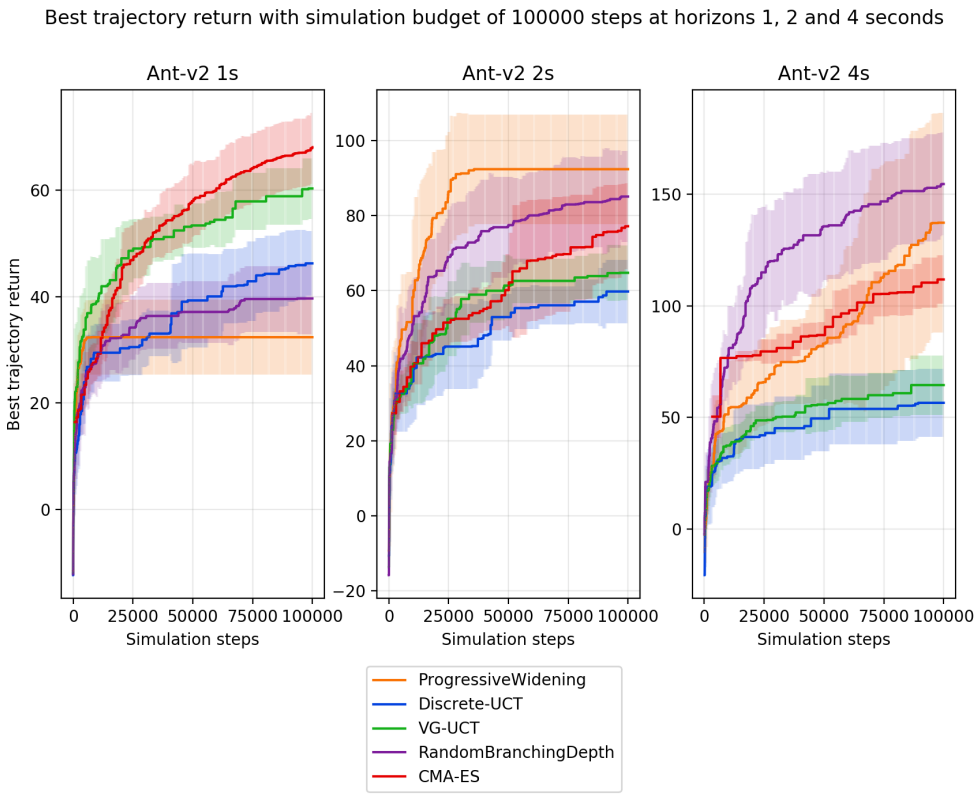

Figure 12: Algorithm performance in Ant-v2 environment with different planning horizons depths.

To determine the quality of solutions, found best trajectories could be replayed and inspected on the MuJoCo physics simulator. Here the trajectories on the Ant-v2 environment are inspected as it was the most high dimensional and complex environment. The objective of the Ant-v2 environment is to move a physics driven quadrupedal character to a single direction as far as possible by applying torques to its joints. Each leg is made with two joints, giving eight dimensions to the action space. The environment terminates if the body of the character touches the floor.

Discrete-UCT found an average best trajectory return of 46.2 units on horizon of 1 seconds, 59.8 units on 2 seconds and 56.5 units on 4 seconds. The found trajectories were chaotic with high degree of variability. On the shortest planning horizon the trajectories amounted to a jump towards a single direction. On longer horizons the trajectories played out as a series of jumps, with almost half of the trajectories ending with the character tipping over at the end.

VG-UCT acquired an average best trajectory return of 60.3 units on horizon of 1 seconds, 64.9 units on 2 seconds and 64.5 units on 4 seconds. The quality of the trajectories was similar, although there were less trajectories where the character fell over on long planning horizons.

Progressive Widening had an average best trajectory return of 32.4 units on horizon of 1 seconds, 92.3 units on 2 seconds and 137.3 units on 4 seconds. The trajectories were less chaotic on a shallow planning horizon than the other algorithms. On longer horizons the character tended to do similar small hops and even flips. However, unlike the other variants, the character could often recover and continue after flipping. The gait of the character was asymmetrical and random.

Random Branching Depth reached an average best trajectory return of 39.2 units on horizon of 1 seconds, 82.3 units on 2 seconds and 146.3 units on 4 seconds. Trajectory quality was similar to Progressive Widening, chaotic and employing flips, but often recovering afterwards. Similarly, there was no symmetric gait or strategy to moving the character.

CMA-ES had an average best trajectory return of 57.3 units on horizon of 1 seconds, 77.4 units on 2 seconds and 111.0 units on 4 seconds. The trajectory quality was similar to Progressive Widening and Random Branching Depth, but less chaotic and often slower, with finer movements. On shorter trajectories the character could recover from flipping where as on the longest horizon depth there was no flipping at all.

5 Conclusions

The findings of the empirical study are further evaluated and analyzed in the following section. Possible future research directions are also discussed.

5.1 Analysis of results

The empirical study measured the performance of three MCTS extension methods against two non-MCTS baseline algorithms. To the knowledge of the author, this is the only performance study which compares multiple extension methods in a general setting and the first study to compare the performance of the VG-UCT and Progressive Widening algorithms. The inclusion of more MCTS variants would have improved the informativeness of the study significantly, but even with only three variants, there were some novel findings. While the results of the empirical study were mostly to be expected, they were surprising in other aspects.

The study found that MCTS variants performed well in general continuous control settings when using offline planning. Surprisingly the best performing algorithm varied based on the used environment and length of the planning horizon. By counting the best performing algorithm across all environments and time horizons, it could be determined that Progressive Widening was the best general purpose MCTS extension in offline planning continuous control settings. While this metric does not take into account the absolute score difference between the algorithms, the amount of variance or convergence rate, it does provide a conclusive ranking based on the data collected. The results also suggest that MCTS variants can perform equally well or better than CMA-ES with long planning horizons and/or high dimensional action spaces, where evaluating a single trajectory at a time is computationally expensive. The performance of the Random Branching Depth algorithm also suggests that the search tree structure of MCTS is not as essential a part as the UCB metric used to select promising branches, when doing deep exploration of continuous control action spaces. This can be seen in the good performance of the Random Branching Depth algorithm.

The largest factor contributing to errors in performance calculation was implementation of the algorithms, as it was impossible to tell if the algorithms were bug free and implemented to the original authors specifications. This was somewhat mitigated by using class inheritance in algorithm implementation, as an error in a critical part would likely manifest in performance of all sub-class algorithms. The second largest error source is possibly hyperparameter optimization, as it is conducted with a limited grid search, using only two optimization environments. It is possible that fine tuning the hyperparameters further could change the results like for example in the case of Discrete-UCT algorithm. Third major source of error is randomness, which can be seen from the large standard deviation values. Random variation was accounted for by averaging the results of multiple runs in each environment and horizon combination, but variation between runs remained large. Validity of results was further evaluated by replicating the best found trajectories in the environments and the best trajectories were found to be valid and matched the best trajectory

return, which was graphed. Qualitative analysis was conducted to further validate that trajectories of higher score were indeed superior to low score ones.

Some findings of this study are corroborated by previous research, while others seem to deviate from established results, even though direct comparison is challenging. Discrete-UCT performed poorly is some environments due to too coarse discretization of the action space. This has been previously documented as a weakness of the method (Fischer et al., 2015) (Browne et al., 2012). Progressive Widening was sometimes outperformed by the simple Discrete-UCT algorithm, which was surprising. Progressive Widening has been previously benchmarked against Discrete-UCT in multidimensional continuous problem settings, where it beat Discrete-UCT (Couetoux, 2013). The problem settings in the original paper was custom built, and not directly comparable to MuJoCo problem settings, which could explain the difference. VG-UCT has been established previously to outperform Discrete-UCT in continuous control OpenAI Gyms environments similar to the ones used in this study (Lee et al., 2020). This is contradictory to what was found in the empirical study in most cases. Again the results are not directly comparable as the original paper was comparing stochastic online planning with a set planning time, instead of offline planning with limited simulation step budget. It follows that Discrete-UCT would most likely perform poorly in stochastic settings, as it does not account for random variation between state transitions.

5.2 Future research

The empirical study here was done to measure the performance of MCTS extension methods. While the study produced results, the study would be improved with the inclusion of more methods. In particular the methods based on machine learning are still lacking in experimental validation in standard problem environments. Findings of the study also seem to indicate that MCTS methods could potentially perform better than CMA-ES when dimensionality and depth of planning horizon increases and consequently evaluation of entire trajectories becomes more expensive. Further research would be needed to confirm this hypothesis. Online planning and time limited open loop planning performance also provides an interesting research direction when considering usage in real-time problems.

References

David Auger, Adrien Couetoux and Olivier Teytaud. Continuous upper confidence trees with polynomial exploration–consistency. *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 194–209. Springer, 2013.

Seydou Ba, Takuya Hiraoka, Takashi Onishi, Toru Nakata and Yoshimasa Tsuruoka. Monte carlo tree search with variable simulation periods for continuously running tasks. *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 416–423. IEEE, 2019.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012. Cited By :573.

Adrien Couetoux. *Monte Carlo tree search for continuous and stochastic sequential decision making problems*, 2013.

Adrien Couetoux, Mario Milone, Matyas Brendel, Hassan Doghmen, Michele Sebag and Olivier Teytaud. Continuous rapid action value estimates. *Asian Conference on Machine Learning*, pages 19–31, 2011a.

Adrien Couetoux, Mario Milone, Matyas Brendel, Hassan Doghmen, Michele Sebag and Olivier Teytaud. Continuous rapid action value estimates. *Asian Conference on Machine Learning*, pages 19–31, 2011b.

Peter I. Cowling, Michael Buro, Michal Bida, Adi Botea, Bruno Bouzy, Martin V. Butz, Philip Hingston, Héctor Muñoz-Avila, Dana Nau and Moshe Sipper. Search in real-time video games. *Dagstuhl Follow-Ups*, volume 6. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013.

Jacob Fischer, Nikolaj Falsted, Mathias Vielwerth, Julian Togelius and Sebastian Risi. Monte-carlo tree search for simulated car racing. *FDG*, 2015.

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

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. *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 9916–9924, 2020.

Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. *European conference on machine learning*, pages 282–293. Springer, 2006.

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

Christopher R. Mansley, Ari Weinstein and Michael L. Littman. Sample-based planning for continuous action markov decision processes. *ICAPS*, 2011.

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

Kourosh Naderi, Joose Rajamäki and Perttu Hämäläinen. Discovering and synthesizing humanoid climbing movements. *ACM Transactions on Graphics (TOG)*, 36 (4):1–11, 2017.

J. J. Rajamaki and P. Hamalainen. Continuous control monte carlo tree search informed by multiple experts. *IEEE Transactions on Visualization and Computer Graphics*, 2018. Article in Press.

Philippe Rolet, Michèle Sebag and Olivier Teytaud. Boosting active learning to optimality: A tractable monte-carlo, billiard-based algorithm. *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 302–317. Springer, 2009.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran and Thore Graepel. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

Emanuel Todorov, Tom Erez and Yuval Tassa. Mujoco: A physics engine for model-based control. *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

Guy Van den Broeck and Kurt Driessens. Automatic discretization of actions and states in monte-carlo tree search. *Proceedings of the ECML/PKDD 2011 Workshop on Machine Learning and Data Mining in and around Games*, pages 1–12, 2011.

Ari Weinstein and Michael L. Littman. Bandit-based planning and learning in continuous-action markov decision processes. *Twenty-Second International Conference on Automated Planning and Scheduling*, 2012.

Amir Yazdanbakhsh, Ebrahim Songhori, Robert Ormandi, Anna Goldie and Azalia Mirhoseini. Tpo: Tree search policy optimization for continuous action spaces. 2019.

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

A Appendix

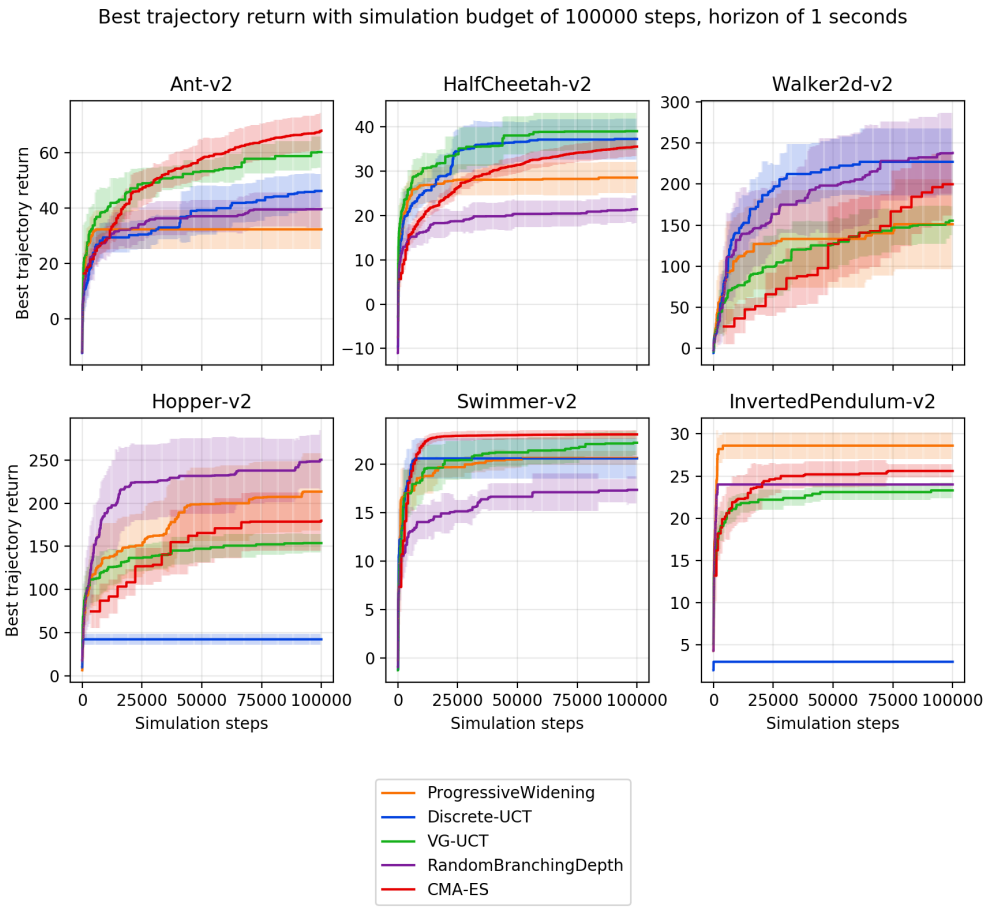

Figure A1: Performance of algorithm in all test environments 1s.

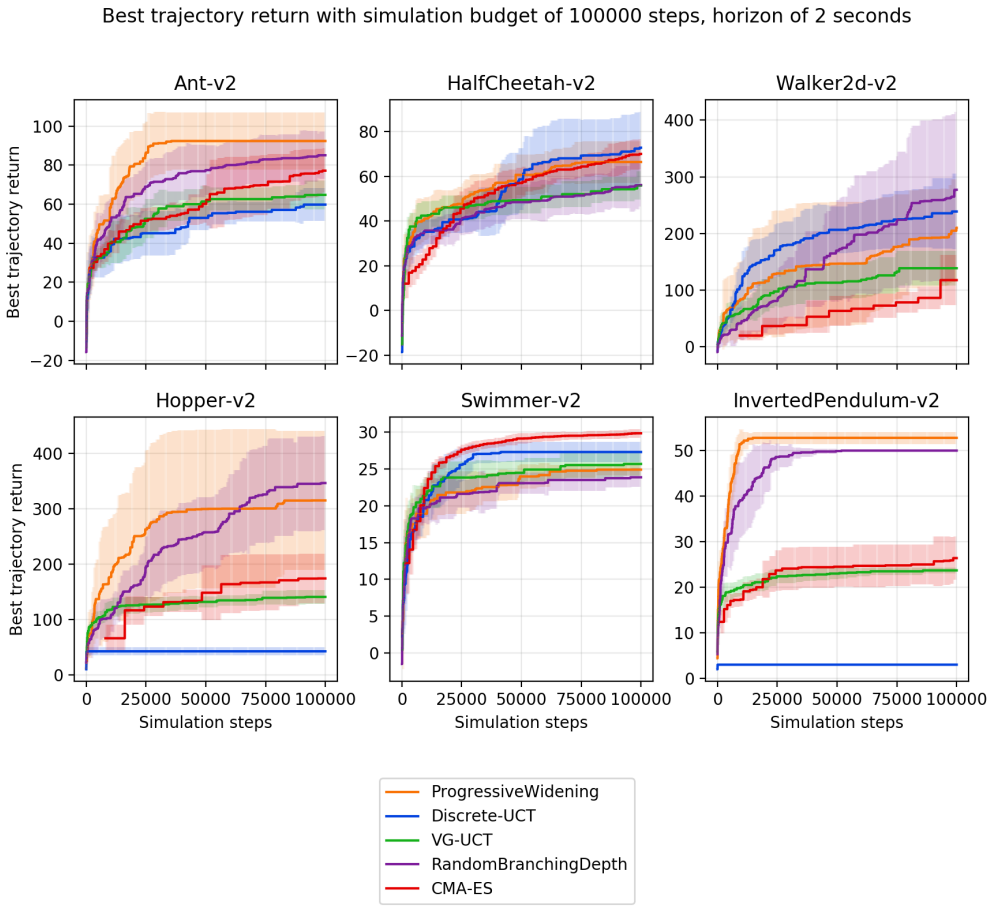

Figure A2: Performance of algorithm in all test environments 2s.

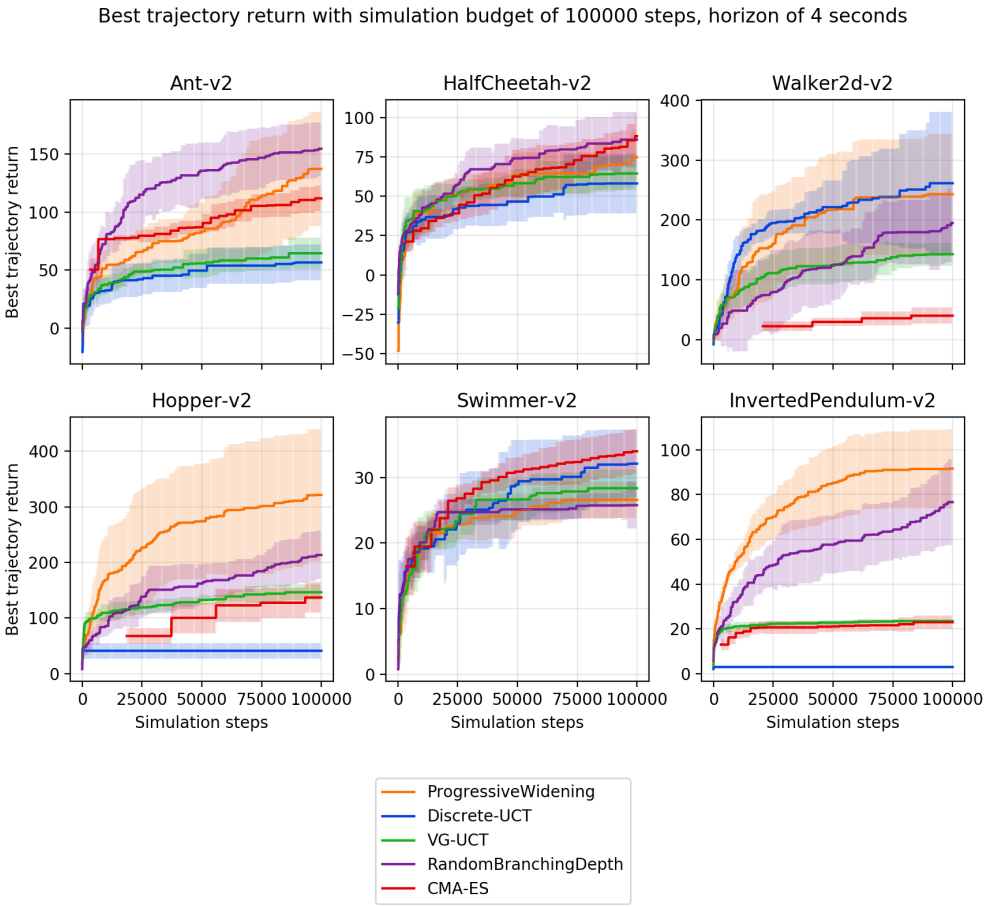

Figure A3: Performance of algorithm in all test environments 4s.