DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2017

High-Level Decision Making in Adversarial Environments using Behavior Trees and Monte Carlo Tree Search

BJÖRN DAGERMAN

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF COMPUTER SCIENCE AND COMMUNICATION

KTH Computer Science
and Communication

High-Level Decision Making in Adversarial Environments using Behavior Trees and Monte Carlo Tree Search

BJÖRN DAGERMAN

Master's Thesis at CSC
Supervisor: Petter Ögren
Examiner: Patric Jensfelt

Master's Thesis in Computer Science
Degree Programme in Computer Science and Engineering
Swedish Title: *En Algoritm för Beslutsfattande i Spelsituationer som Kombinerar Beteendeträd och Monte Carlo Tree Search*
Björn Dagerman, dagerman@kth.se
June 26, 2017

Abstract

Monte Carlo Tree Search (MCTS) has been shown to produce good results when used to model agent behaviors. However, one difficulty in MCTS is anticipating adversarial behaviors in multi-agent systems, based on incomplete information. In this study, we show how a heuristic prediction of adversarial behaviors can be added, using Behavior Trees. Furthermore, we perform experiments to test how a Behavior Tree Playout Policy compares to a Random Playout Policy. Doing so, we establish that a Behavior Tree Playout Policy can effectively be used to encode domain knowledge in MCTS, creating a competitive agent for the Adversarial Search problem in real-time environments.

Sammanfattning

Monte Carlo Tree Search (MCTS) har visat sig producera bra resultat när det använts för att modellera agenters beteenden. En svårighet i MCTS är att förutse motståndares beteenden i multiagentsystem, baserat på ofullständig information. I den här studien visar vi hur en heuristisk förutsägning kan tillföras MCTS med hjälp av Beteendeträd. Dessutom genomför vi experiment för att utvärdera hur en Beteendeträds-policy förhåller sig till en Random-policy. Som följd av detta konstaterar vi att en Beteendeträds-policy effektivt kan användas för att koda in domänkunskap i MCTS, vilket resulterar i en konkurenskraftig agent för Adversarial Search-problemet i realtidsmiljöer.

Contents

Chapter 1

Introduction

A difficult domain in Artificial Intelligence concerns multiple agents, with conflicting goals, competing against each other. This class of problems is called Adversarial Search (AS) problems, and is typically referred to as games. The reason AS problems are difficult is due to the often times exponential time complexity of finding optimal solutions. This is because games in this class often have a high branching factor, while also lasting many moves. For instance, the well-known game GO has a branching factor of 250, on average. Also, GO games typically last up to 400 moves. Hence, there are about 250^{400} game states in total, of which roughly 2.08×10^{170} are legal [10]. Searching through all these states is infeasible within reasonable time. Therefore, instead of trying to find optimal solutions, a common approach is to search for good *approximate* solutions. This typically involves not evaluating the entire search space, but instead introducing a heuristic evaluation function. Such a heuristic should give a good estimate of the utility of a given action, in much less time than computing an optimal answer.

Monte Carlo Tree Search (MCTS) has been shown to produce effective approximations of good solutions for GO [31]. GO can be classified as a two-player zero-sum game with perfect information. What this means is that at all times, each player can observe the complete game state, and given a terminal state, it is clear that either one player won, or the other. Other difficult games are real-time strategy games, such as StarCraft. In real-time environment, decisions must be made within a strict deadline, typically less than a second. Given a strict deadline, finding *fast* approximations becomes important. Moreover, MCTS has been shown to quickly find good solutions in real-time environments. The reason MCTS performs well, is that it simulates outcomes of games, choosing actions that are observed to perform well over many simulations. One component of MCTS is a playout policy. This policy, or strategy, dictates which actions agents select during simulations. Either these actions can be selected at random, or according to some strategy that encodes domain knowledge. Also, given that the number of moves to reach a terminal state is vast, a heuristic evaluation function can be used to only perform playouts to some maximum depth, where the function is used to evaluate the utility at the given depth. However, a problem with MCTS is that during playouts of limited depth, it is difficult to anticipate actions of adversarial agents.

Final State Machines (FSMs) have been used to model agent logic in AI, by expressing the different states an agent can have, and more specifically, the transitions between states. A problem with FSMs is that they do not scale well. That is, for large systems, being able to both reuse states to add new features, and to reuse transitions, is problematic. Behavior Trees (BTs) is an alternative to FSMs, which tries to solve this problem by encapsulating logic transparently, increasing the modularity of states. In BTs, states are self-contained and do not have any transitions to external states. Therefore, states are instead called behaviors. BTs are a new area of research, but initial results have shown that they make logic modular and easier to use [8].

In this study, we want to evaluate if heuristic predictions of adversarial actions can be added to MCTS using BTs, improving the quality of approximations. That is, can BTs be used as a playout policy by MCTS to find better solutions? This question is of interest for current research, in part because it may introduce new improvements that may improve the speed of which MCTS can find good solutions. Also, since BTs is a new area of research, it is useful to evaluate unexplored uses of BTs.

We denote an algorithm using Monte Carlo Tree Search with a Behavior Tree playout Policy as MCTSBTP. Ideally, such an algorithm would combine the attributes of both MCTS and BTs. This could enable easy modeling of agents, while providing good solutions to the AS problem. Thus, MCTSBTP could make it easier to express powerful agents, enabling modeling of more realistic agents with dynamic behaviors. Furthermore, MCTSBTP could be used in a wide range of areas where it is difficult to model the real world, requiring agent behaviors to be modular and easily updated or changed.

Being able to perform adversarial search, while maintaining modularity in agent behaviors, enables applications where it is difficult to predict all parameters of a mission. E.g., in Search and Rescue operations using UAVs. Using MCTSBTP makes it possible to adapt agents behaviors as unforeseen circumstances arise. As this could enable the autonomous systems to be more easily adapted to new situations, it could aid human decision makers explore various strategies by changing behavior parameters.

1.1 Problem

Our problem is to design (and implement) an algorithm for the real-time adversarial search problem using Monte Carlo Tree Search and Behavior Trees.

1.2 Objective and Hypothesis

Our objective is to show that BTs can be used to extend MCTS with heuristic predictions of adversarial actions.

Hypothesis: *The performance of MCTS can be improved by using a simple BT play-out policy, instead of a standard random policy, for real-time adversarial games.*

In order to achieve the objective, we will implement MCTS using both a random policy and a Behavior Tree policy. We will also implement a baseline team. The implementation will be tested in a well-known environment. To test the hypothesis, we will first play many games (enough that the results will be statistically significant) with MCTS, using a random policy, against the baseline team. Then, we will play many games of MCTS, using the BT policy, against the baseline. Given these observation, we can note and compare differences in performance. We can then evaluate how the number of iterations, and the search depth, is affected by the change in policies.

We will compare the policies to test the hypothesis. Computational time will be taken into consideration, to ensure the suggested solution is applicable in real-time domains.

1.3 Delimitation

Our work is based on research in MCTS and BTs (see Chapter 3). We only evaluate our method on one game (see Chapter 4 *Capture the Flag Pac-Man*), although previous research have shown this environment to be sound [3]. Although we can not prove the environment to be perfectly fair, we will take precautions to ensure it is as fair as possible. Lastly, our game has a Partial Observability property. How to best deal with this property is an open question and answering it is outside the scope of this study. Therefore, we will mostly disregard it. Although, we will observe how it affects the maximum playout depth.

1.4 Sustainability, Ethical, and Social Aspects

The proposed approach enables applications in domains where it is difficult to predict all parameters ahead of deployment. Such domains include, e.g., Disaster Relief, and Search and Rescue operations. Suppose a decision making and planning system, based on a black-box, was deployed. Then, in a domain where the state-of-the-art is constantly improving, such a black-box may be difficult to modify, given the desire to integrate newly discovered domain knowledge. This could result in systems that are either outdated or replaced. In such cases, an alternative system with modular logic, based on our proposed approach, could instead enable system engineers to extend existing Behavior Trees with new features, at a presumably lower cost than replacing the entire system. Furthermore, it could aid human decision makers explore a range of new strategies, at a faster rate, by experimenting using different Behavior Trees. In addition to the potential economic gain by using such a system, because Behavior Trees can be understood by humans—it is conceivable that an engineer could design and input new features to an existing Behavior Tree—at a faster rate than previously possible, in response to an unseen situation. Thus, allowing faster response times. Lastly, because the decision model is human readable, and the decisions found by MCTS can be traced, it could allow for introspection by human decision makers, e.g., in safety-critical domains.

To recapitulate, the modularity and ease of use of logic could potentially give the system a high level of maintainability. This could further enable easier expansions of the system, while diminishing the need for full system replacements. Such interactive properties, could have an immense value in operations such as Disaster Relief or Search and Rescue, where ultimately human lives are at risk. The proposed approach could potentially be used to improve human decision makers ability to handle unforeseen circumstances in such situations, which is of clear value to society.

Ethics in Artificial Intelligence has been a subject of much discussion. As this study concerns decision making and planning, it is appropriate that we briefly discuss some ethical considerations. The main concern that relates to this study, is that of the trustworthiness of the AI system. That is, when using an AI system, can we blindly trust that the system does what it is designed to? Regardless of how well the intentions of a systems engineer may be, for any complex system, there will likely be bugs. In the case of decision making and planning, e.g., using autonomous vehicles, a misbehaving agent may cause harm to the environment, other agents, or even a human. Although such scenarios must be avoided at all cost, we argue that it is not the responsibility of the underlying algorithm, such as MCTSBTP, to provide such capabilities. Rather, these factors should be considered at a higher level, i.e., when designing safety features, such that they are able to override the underlying decision making.

1.5 Summary of Contributions

This thesis presents four contributions, listed below. For each contribution, we give a brief description, followed by a reference to the chapters and sections where the results can be found.

- We show how high-level decisions can be made in real-time games using BTs and MCTS in Chapter 5, extending the work described in Chapter 3.
- We conclude that the performance of MCTS can be improved by using a simple BT playout policy, instead of a standard random policy, for real-time adversarial games, in the discussion in Chapter 7, based on the experiments described in Section 6.2, and results presented in Section 7.1 and discussed in 7.2.4.
- We conclude that the proposed approach is able to extend MCTS with heuristic predictions of adversarial actions, in Section 7.2.4, based on the results presented in Section 7.1.
- We give a suggestion for a new algorithm MCTSUT in Section 8.1, potentially making modeling of the environment easier.

Chapter 2

Background

This chapter introduces the necessary background theory required for the main body of this study. In Sections 2.1 and 2.2, a brief introduction to decision and game theory is presented. Section 2.3 introduces the multi-armed bandit problem, on which Monte Carlo Tree Search is based. Sections 2.1-2.3 give the required background to discuss Monte Carlo Tree Search in Section 2.4. Lastly, Section 2.5 presents Behavior Trees. We will revisit Sections 2.4 and 2.5 when discussing algorithm design considerations and our proposed approach, in Chapter 5.

2.1 Decision Theory

Decision Theory is a framework for decision problems that depend on some degree of uncertainty, combining probability theory and utility theory. Markov decision processes is used to tackle problems, for which the utility of a state directly follows from the sequence of decisions yielding that state.

2.1.1 Markov Decision Processes

In fully observable environments, sequential decision problems can be modeled using Markov Decision Processes (MDPs). In order to model overall decisions, the following is required:

- a set of states, $S = \{s_0, s_1, ..., s_n\}$,
- a set of actions, $A = \{a_0, a_1, ..., a_k\}$,
- a transition function P, where $P(s'|s, a)$ denotes the probability of reaching state s', given that action a is applied in state s,
- a reward function R.

The overall decisions made during a game can thus be expressed as a sequence of (state, action)-tuples, where the transitions for a given (state, action) to a new state is decided by a probability distribution.

Definition 2.1 (Policy) A *policy* $P : S \rightarrow A$ is mapping from each state in S to an action in A.

Typically, there exists many policies, and the goal is to find the policy that maximizes the reward function.

2.1.2 Partially Observable Markov Decision Processes

In case of every state not being fully observable, the above MDPs formulation needs to be extended with an observation model $O(s, o)$, which expresses the probability of making observation o in state s.

In short, the difference between Partially Observable Markov Decisions Processes (POMDPs) and MDPs can typically be seen by agents performing actions to gather information. That is, actions that are not necessarily good right now, but that will enable better decisions to be made in the future.

Exact planning in POMDPs that operate on all possible information states, or *beliefs*, suffer from a problem known as the *curse of dimensionality*. Meaning that, the dimensionality of a planning problem is directly related to the number of states [13]. Additionally, POMDPs may also have a problem where the number of belief-contingent plans increases exponentially with the planning horizon [28]. This is known as the *curse of history*. Littman showed that the propositional planning problem is NP-complete, whereas exact POMDP is known to be PSPACE-complete [18]. Therefore, domains with even small amounts of states, actions, and observations are computationally intractable.

2.2 Game Theory

Define a game as an environment containing one or more agents, each interacting to produce specified outcomes [3]. These interactions are subject to a set of rules. Though similar to Decision Theory, in games, agents need also consider other agents decisions. In order to extend Decision Theory to games, we need the following components:

- a set of states $S = \{s_0, s_1, ..., s_n\}$,
- a set of actions, $A = \{a_0, a_1, ..., a_k\}$,
- a transition function, $T : S \times A \rightarrow S$,
- a reward function R,
- a list of agents, declaring which agent should act in which state, B,
- the set of terminal states, $S_T \subseteq S$.

A typical game will be played as follows: the game starts in state s_0. For any state s_t, the agent to act—as defined by B—selects a valid action a. Given (s_t, a), the transition function T is applied to yield state s_{t+1}. This process is repeated until a terminal state $s \in S_T$ is reached. How an agent selects an action for a given state is defined by the agent's *policy*. In general, an agent's policy will select an action which has a high probability of yielding a successor state with high reward, given by R.

2.3 Bandit Based methods

In this section, we will give an introduction to Bandit Based methods, on which Monte Carlo Tree Search (Section 2.4) is based.

Given K multi-armed bandit slot machines, which machines should one play and in which order, to maximize the aggregate reward? Or, which is the optimal sequence of actions? This class of sequential decision problems are denoted *Bandit problems*. However, because the underlying reward distributions are unknown, finding optimal actions is difficult. Therefore, one typically relies on previous observations to predict future results. That is, formalizing a *policy* that determines which bandit to play, given the previous rewards. The problem is that this requires a balance between exploiting the action currently believed to be optimal, with exploring other actions. This balance is called the *exploitation-exploration dilemma*.

2.3.1 Regret

The loss in reward a player achieves, by not playing the optimal action, is called the player's *regret*. Lai and Robbins [16] introduced the technique for upper confidence bounds for asymptotic analysis of regret, obtaining a $O(\log n)$ lower bound on its growth. The regret after n plays can be expressed as:

$$\bar{R}_n = n\mu^* - \sum_{t=1}^{n} \mathbb{E}[\mu_{I_t}] \tag{2.1}$$

where μ^* is the best possible expected return, and $\mathbb{E}[\mu_{I_t}]$ the expected return for arm $I_t \in \{1, ..., K\}$. See [4] for more details and stronger results.

2.3.2 UCB1

Based on Agrawal's index-based policy [1], Auer showed that there exists a policy UCB1, achieving logarithmic regret uniformly over n, without any preliminary knowledge of the reward distributions. UCB1 works by playing machine j that maximizes $\bar{x}_j + \sqrt{\frac{2 \ln n}{n_j}}$, where $\bar{x}_j$ is the average reward obtained from machine j, n_j is the number of times machine j has been played, and n is the overall number of plays [2].

2.4 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a family of algorithms, based on the general idea that:

1. the true value of an action can be approximated using random simulations,
2. using these approximations, the policy can efficiently be adjusted to a best-first strategy [3].

MCTS maintains a partial game tree, starting with the empty tree and progressively expanding it following a *tree policy*. Each node in the tree represents a game state, and the edges from a parent to its children represents the available actions from the parent state. As the tree is expanded, observations of the reward of successor states are propagated following the parent path. Thus, as the tree grows, these estimated rewards become more accurate.

2.4.1 Algorithm

Given a computational budget (i.e., a maximum amount of time, memory, or iterations), MCTS iteratively builds a game tree. See Figure 2.1 for an illustration of one iteration of MCTS. In each iteration, the following steps are performed:

1. *Selection*: starting at the root, a child policy is recursively applied until a promising child is found. What constitutes a promising child is determined by the policy. However, it must have unvisited children, and be a non-terminal state.
2. *Expansion*: the available actions at the selected node are explored by an expansion policy, determining which node(s) are expanded. Expanding a child involves adding it to the tree, and marking it for simulation.
3. *Simulations*: the expanded node(s) are simulated following a default policy, yielding an observed reward for each node.
4. *Back propagation*: starting at each of the expanded nodes, the observed results are propagated up the tree, following the parent path until the root is reached.

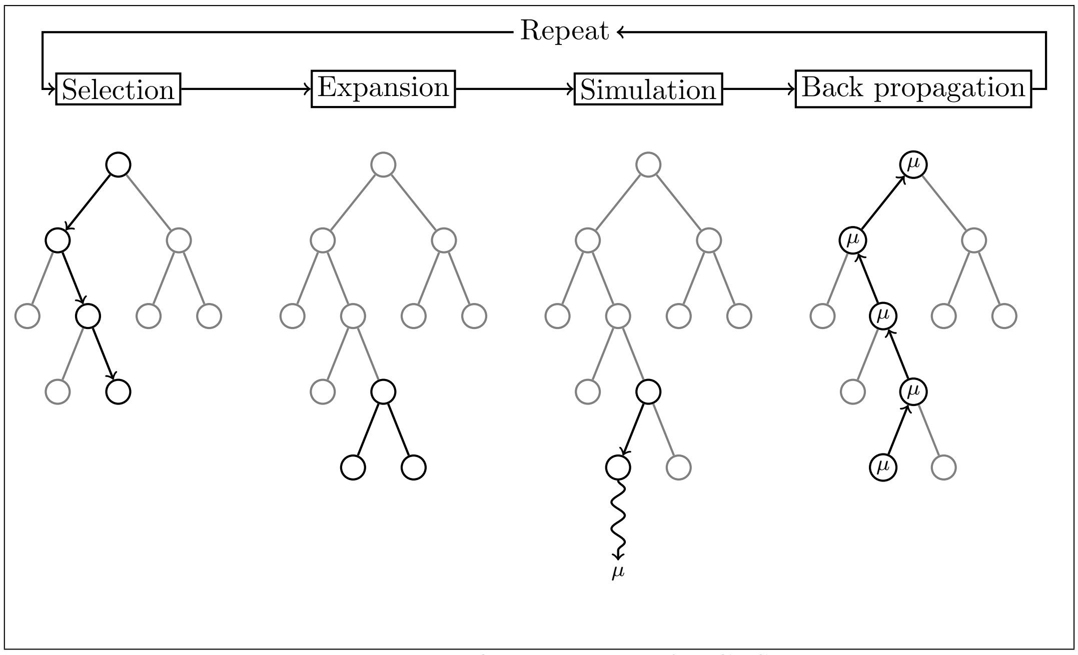

Figure 2.1: One iteration of MCTS.

The back propagation step does not follow a policy. Rather, it updates the node statistics. These statistics are then used in future selections and expansions. Upon reaching the threshold imposed by the computation budget, a valid action from the root node is selected following some selection rule. Such selection rules, as introduced by Schadd [30], based on the work of Chaslot et al. [6], and discussed by Browne et al. in [3], includes:

- *Max child*: select the child with the highest reward.
- *Robust child*: select the most visited child.
- *Secure child*: select the child that maximizes a lower confidence bound.

2.4.2 UCT

The Upper Confidence bound applied to Trees (UCT) is an adaption of UCB1 (Section 2.3.2). It is a popular approach to handling the *exploitation-exploration dilemma* in MCTS, initially proposed by Kocsis and Szepesvari [14, 15]. The idea is that the problem of picking good nodes during the selection and expansion steps, can be modeled s.t. every choice encodes an independent multi-armed bandit problem. Then, the node that should be chosen is one that maximizes equation 2.2:

$$UCT = \frac{w_i}{n_i} + c\sqrt{\frac{\ln t}{n_i}} \tag{2.2}$$

where w_i is the number of wins after the i-th move, n_i the number of simulations after the i-th move, t the total number of simulations for the considered node, and c is the exploration parameter.

2.5 Behavior Trees

Following the success of high-level decisions and behaviors displayed by AI agents in Halo 2 [11], the gaming industry took an interest in adapting the demonstrated techniques [5, 12]. Primarily, this corresponds to Behavior Trees—having been established as a more modular alternative to finite state machines [9].

To describe Behavior Trees (BTs), we will adhere to the definition used by Millington and Funge in [23]. The main difference between a Hierarchical State Machine and a BT, is that BTs are constructed of *tasks* instead of states. A task can either be an action or a conditional check, or it can be defined as a sub-tree of other tasks. By having a common interface for all tasks, it becomes possible to model higher level behaviors using sub-trees. Furthermore, knowledge of the implementation details of each behavior (sub-tree) is not necessary when constructing the tree. Thus, it is possible to model complex behaviors as compositions of sub-trees.

In order to evaluate a BT, a signal (or tick), is pushed through the tree starting at the root. In order to explain how this tick is propagated through the tree, we will first explain the different types of nodes (tasks). Each node can report exactly one of three states: success, failure, or running. The four most essential nodes are:

- *Selector*: a non-leaf node where its children are evaluated in order (left to right). As soon as one child succeeds, so does the *selector*. If no child succeeds, the *selector* fails. A *selector* is considered running if any child is running.

- *Sequence*: a non-leaf node that succeeds iff all its children succeeds. A *sequence* is considered running if any child is running.

- *Action*: a leaf node that succeeds upon completing the action it describes. An *action task* is considered running while the described action is still being evaluated. It fails if the action can not be completed.

- *Condition*: a leaf node that succeeds and fails if the described condition is true or false, respectively.

An illustration of a BT, consisting of the above types of tasks, is shown in Figure 2.2. More details, and more types of tasks (i.e., parallel and decorator) can be found in [23].

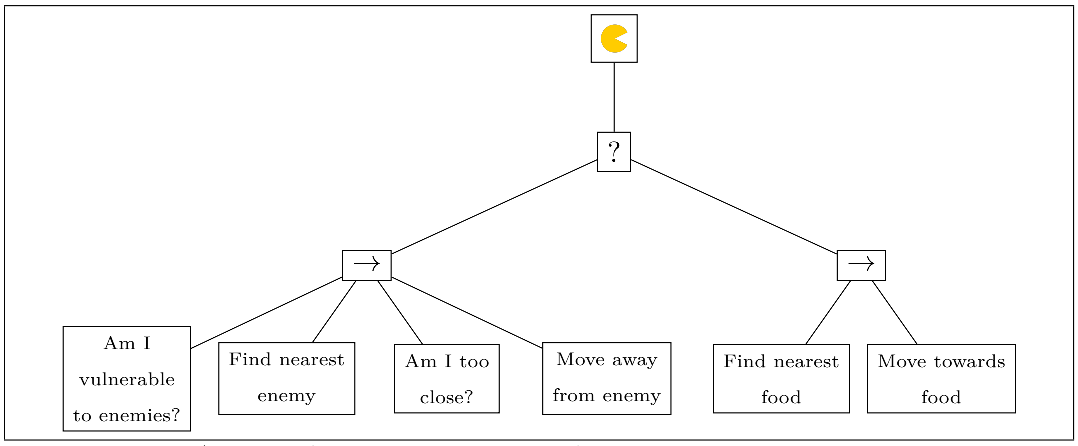

Figure 2.2: A simplified Behavior Tree of Pac-Mans high-level decisions in the game Pac-Man. "→" represents a left to right *sequence*. The "?" represents a *selector*. Leaves with descriptions that are questions represent *conditions*. Leaves with descriptions that are statements represent *actions*. That is, the left sub-tree describes the behavior of running away from close enemies. The right sub-tree describes eating the closest food. The overall strategy encoded in the BT is to try to eat the closest food, unless an enemy is nearby, in which case Pac-Man should run away. See Section 6.2.3 for the actual BT used during the experiments.

For every tick, start at the root and traverse the tree as described by the *control nodes* (selectors and sequences). Upon reaching a leaf (action or condition), said task is evaluated for one time step (one tick), and its status (success, failure, or running) is reported back to its parent, which in turn signals its parent, etc. Thus, the results are back propagated to the root. The typical evaluation of a BT will consist of ticking the root either until it signals *success* or *failure*, or until some computational time budget is exhausted.

Chapter 3

Related Work

In this chapter, we give an overview of the related work that has been done in the domains of Behavior Trees (BTs) and Monte Carlo Tree Search (MCTS). We present select articles for BTs and MCTS in Sections 3.1 and 3.2, respectively. Although many advancements have been made in machine learning using these techniques, our focus is on real-time planning and decision making. As such, we mainly present results related to this topic.

3.1 Behavior Trees

Weber et al. present idioms for reactive planning in multi-scale games in [34]. That is, games where agents are required to make both high-level planning decisions, while also controlling individual agents during battle. They present idioms for handling this in the real-time game of StarCraft. These include reactively building an Active Behavior Tree, that is gradually populated by goals, inspired by Loyalls work in "Believable agents: building interactive personalities" [19]. These techniques relate to our study, as it concerns the problem with Behavior Trees commonly only handling one agent, which is problematic when scaling to multi-agent systems. However, the authors make no effort in generalizing their results beyond StarCraft. Furthermore, their evaluation is against the standard StarCraft bots, which to our knowledge, do not provide an accurate representation of competitive AI agents. Marzinotto et al. further discusses the problem of cooperation in multi-agent systems, using separate Behavior Trees, in [21]. They suggest introducing a special *Decorator* node that handles synchronization between different agents' Behavior Trees.

In On-Line Case-Based Planning (OLCBP), planning at different levels suffers from problems in reactivity. For instance, not being able to interrupt execution of tasks as the parameters of the world changes, making the actions being evaluated no longer desirable. Palma et al. show how a *Tactical Layer* can be constructed in [26], using a database of Behavior Trees for low-level tasks. That is, they propose a system where high-level decisions are formed by combining low-level BT decisions. Although their high-level planning is based on learning and thus different from ours, their overall system design may be applicable to our system. However, they provide no empirical performance analysis.

In [17], Lim et al. use Behavior Trees to build a bot for the real-time game DEFCON. The authors implemented BTs for low-level actions, then combined these BTs randomly to build an agent that maintains an above 50% win-rate against the game's default bot. They argue that this shows that it may be possible to automate the process of building agents. A problem with their approach is that they need to provide fitness functions for each behavior, in order to guide the combinations. How these functions should be defined is unclear. However, is is clear that these functions may sometimes be problematic to define in certain environments. We hypothesize that our approach of using MCTS simulations could instead be used to combine the behaviors.

3.1.1 Summary

There is a synchronization problem if Behavior Trees are separated for individual agents in multi-agent systems [34]. This problem can be addressed by extending BTs with new node types [21]. We will further discuss this problem in Chapter 5 *Proposed Approach*.

BTs are commonly used for low-level tasks. However, how to best use BTs for high-level decision making is still an open question. Previous work have explored different learning or stochastic processes. To our knowledge, there have been no attempts in using simulation-based methods for high-level decision making with BTs.

3.2 Monte Carlo Tree Search

Samothrakis et al. discuss how the search depth can be limited to deal with games that have no clear terminal winning state [29]. They extend previous research in MCTS to N-player games by extending it to Max^n trees. Max^n is a procedure to find the equilibrium point of the values of the evaluation function in N-player games [20]. Lastly, they evaluate their approach by playing against other bots. As our study involves multi-agent systems, this work is of clear interest.

In [27], Pepels and Winands present a bot for the real-time game Ms. Pac-Man, using MCTS to guide its decisions. The authors claim to use MCTS to find optimal moves at each turn. However, they provide no proof of optimality. Nonetheless, they provide a detailed description of their implementation, and the considerations that must be taken to adapt MCTS to Ms. Pac-Man. They show significant improvements while playing against competitive bots, concluding that the MCTS framework can be used to create strong agents for Ms. Pac-Man. This result relates to our study, as we want to use MCTS in real-time games. One potential problem with their approach is that they use highly formalized strategies to encode domain knowledge. This raises the question of whether these results for MCTS can be reproduced without finely tuned strategies, i.e., using BTs.

A new Monte Carlo algorithm, POMCP, is proposed in [32]. POMCP is able to efficiently plan in large POMDPs (Section 2.1.2), without needing an explicit probability distribution. Recall the *curses of dimensionality and history.* The authors claim to break both these curses using Monte Carlo sampling. This article is of interest as it discusses problems with POMDPs, while showing empirical data of performance limitation, if the partial observability is not handled with care.

3.2.1 Summary

In games with no clear terminal winning state, heuristic evaluations can be made at a limited search depth [29]. MCTS can find good solutions in real-time games, if highly formalized strategies are applied during playouts [27]. Lastly, not handling partial observable properties may impose performance limitations [32].

Both [27] and [29] disallow Pac-Man to reverse direction when expanding the tree. This attempt at reducing the branching factor by limiting moves is questionable, especially in [27], where the authors claim to produce optimal paths. This seems unlikely as they prune part of the search space, without showing that it cannot possibly lead to better solutions.

Chapter 4

Capture the Flag Pac-Man

In this chapter, we give an introduction to the game Capture the Flag Pac-Man in Section 4.1, followed by a classification in Section 4.2. Lastly, we motivate why this game is of interest for our study, in Section 4.3.

4.1 Environment Introduction

In Capture the Flag Pac-Man (CTF Pac-Man), each player controls $\frac{N}{2}$ agents, constituting a team. The map is a maze with reflection symmetry. Each team has a unique home and away field of the map. Agents' starting locations are fixed in their home field. An agent on its home field is considered a defender. An agent on the away field is considered an attacker. The map is populated by pellets, or *food*. When an attacker touches food on the away field, the agent picks it up. If a defender touches an attacker, the attacker dies, dropping all collected food, before respawning at its start location. If an attacker returns to its home field carrying $m > 0$ food, the agent trades these food for m points. For a game with $\frac{N}{2}$ agents per team, the game ends when only $\frac{N}{2}$ food remains on either player's home field, or after a set time has elapsed. That is, it is not a winning strategy to stand still on some arbitrarily selected location. The winner of the game is the player with most points. If both players have the same amount of points, the game is considered a draw. See Figure 4.1 for an illustration of the game.

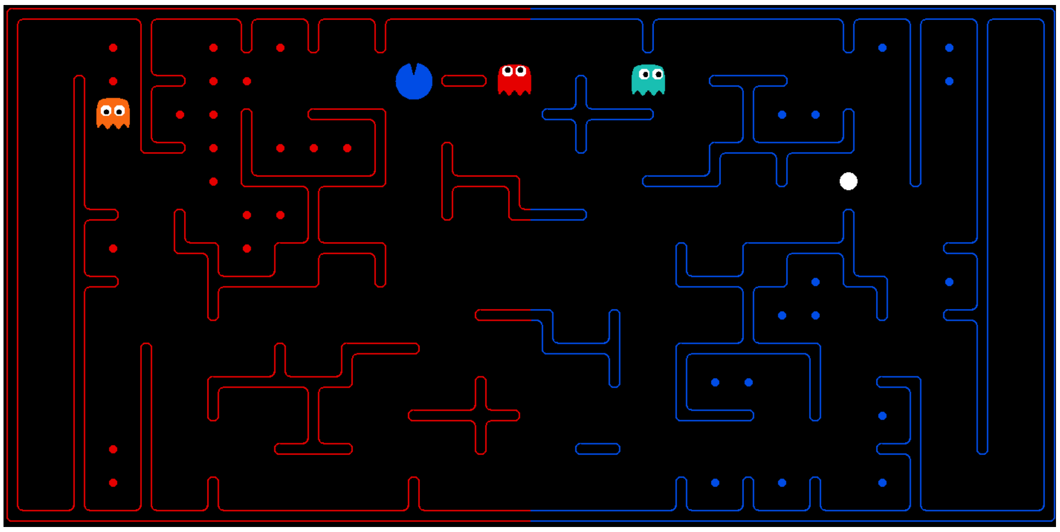

Figure 4.1: An illustration of the game CTF Pac-Man. The *Red* team is represented by the orange and red agents, while the blue and teal agents belong to the *Blue* team. As both of the *Red* agents are defending (their position is on their half of the map), they are ghosts. The blue agent, on the other hand, is a Pac-Man, as it is on the opponent's side of the map.

4.2 Environment Classification

- *Partial Observable*: Agents do not have complete knowledge of the world. That is, their sensors can only perfectly detect opponents that are within 5 units of distance. For an agent agt_1 in state $s \in S$ observing opponent agt_2, with $dist(agt_1, agt_2|s) > 5$, an observation $o = O(agt_2, s)$, is given instead of a true (x, y)-location. This observation o contains a perceived distance $5 < dist(agt_1, agt_2|o) < \infty$, following a given discrete probability distribution.

- *Multi-agent*: A game is played by N agents, with each team consisting of $\frac{N}{2}$ agents that work together to defend and attack the adversaries.

- *Stochastic*: Because agents' sensors can not perfectly detect opponents at all times, the state transition function is dependent on probabilistic features of the environment and adversaries.

- *Sequential*: Every action changes the game state. Denote the action applied in state s_k as a_k, and the resulting state $s_{k+1} = s_k(a_k)$. Then, any state s_n can be expressed as a sequence of the n actions $\{a_0, \ldots, a_n\}$ applied to states $\{s_0, \ldots, s_n\}$, respectively.

- *Semi-dynamic*: There is a fixed time to make decisions. During this time, agents are given a guarantee that the environment will not change.

- *Discrete*: There is a finite amount of states and state transitions.

4.3 Environment Motivation

A majority of research in real-time games is split between the two games StarCraft, and Pac-Man, making either of these environments interesting. The problem with planning and decision making in StarCraft is that of integrating reasoning at multiple levels of abstraction [25]. That is, in order to apply MCTS to StarCraft, the use of abstraction is required [24]. Thus, in order to use StarCraft, the scope of our subject would have to be expanded. Therefore, we select Pac-Man. Capture the Flag Pac-Man is selected because, unlike traditional Pac-Man, it has a terminal winning state, making evaluation easier. Also, relevant studies exist that have evaluated the use of UCT in Pac-Man, e.g., [27, 29], giving a foundation for continued work.

Chapter 5

Proposed Approach

In this chapter, we start by extending MCTS with a Behavior Tree Playout Policy (MCTSBTP) and adapting the algorithm to CTF Pac-Man, as described in Sections 5.1.2 and 5.1.1, respectively. A detailed discussion of the evaluation functions is given in Section 5.2. Section 5.3 discusses the design decisions related to the Partial Observability property of CTF Pac-Man. Lastly, Section 5.4 briefly discusses how MCTS can be extended to cooperative environments.

5.1 Policies and Domain Knowledge

5.1.1 Selection and expansion

During the selection step, either a child node is selected at random, if the child's visit count, n_c is under a threshold T. Based on the suggestion by Pepels and Winands [27], we select $T = 3$ and $C = 1.5$. For $n_c \geq T$ apply UCT (see Section 2.4.2), i.e., select the child that maximizes equation 5.1:

$$n_s + C\sqrt{\frac{\ln n_c}{n_p}} \tag{5.1}$$

where n_s is the current score of the considered node, n_p the visit count of the current node, and n_c the visit count of child c. Both T and the exploration constant C, are naturally domain dependent and may require experimentation to find.

The visit threshold T is used due to the Partial Observable, and Stochastic properties of CTF Pac-Man (Section 4.2), to ensure the robustness of the selected path. That is, because an otherwise safe node can evaluate to a loss due to the above-mentioned factors, applying a threshold ensures that this risk becomes negligible.

5.1.2 Behavior Tree Playout Policy

Given the Behavior Tree B_i that perfectly expresses the i-th agent's decision model, tick B_i to get an action a. Apply a for agent i. Repeat for each agent at depth d, and increment $d = d + 1$. Repeat until the maximum depth is reached. Naturally, in some settings, one may not have a perfect BT for each agent, or the BT may simply be an approximation. In such case, either:

1. select some BT B_j to be used as an approximation for agent i. Tick B_j to get an action a_{bt}. Let a_r be a randomly selected action, from the set of valid actions for agent i in the current game state. Set $a = a_{bt}$ with probability p and $a = a_r$ with $1 - p$,

2. or, in addition to the above strategy, after each iteration of MCTS, note the actual move performed by the agent. Now, as BTs are modular and easily changed, given a set of observations, it is conceivable one could adjust the parameters of B_j to better match the actual observations, but dynamically improving BTs based on observations is an open question.

5.2 Evaluation Function

Ideally, simulations should run until a terminal game state is reached, provided that such a state has a clear winner or loser. Suppose each simulation is represented by the sequence of states $\{s_0, s_1, ..., s_{d-1}, s_d\}$, obtained by repeatedly applying the playout policy until depth d. Given a set of such sequences of states, representing these simulations, the search tree can be constructed (following UCT)—such that it reveals which states tend to lead to victories—and which lead to losses. Then, decisions can directly be based on this tree, and we can properly apply, e.g., minmax decisions. However, there are two main problems with this approach: firstly, the search depth of CTF Pac-Man is more than 1000 moves. Secondly, it is a game of imperfect information. Meaning that the error of a simulation will increase as the depth of the simulation increases (see Section 5.3).

As previously discussed, one approach to these problems is to instead stop simulations at some maximum depth. When stopping a simulation, an evaluation of the current game state is required, to determine how desirable it is. This is commonly achieved by defining a heuristic evaluation function, which given a game state outputs a numerical value, where the magnitude of that value corresponds to the utility of the given state. Clearly, when determining the utility of some arbitrary non-terminal game state, looking at the current score may be misleading, as a winning strategy in CTF Pac-Man may involve picking up enough pellets to win without having to return to the away field a second time. Thus, it is of limited consequence what the intermediate score is. Instead, we will present two alternative heuristic evaluation functions in Sections 5.2.1 (Basic Heuristic) and 5.2.2 (Improved Heuristic). These heuristics will be referenced as the *basic* and the *improved* heuristic in the following chapters.

5.2.1 Basic Heuristic

An observation of a game state consists of a number of food locations, and the locations of the various agents. Given this information, the basic heuristic expresses the utility of the given state as a linear combination of the amount of remaining food, and the distance to the closest food. Each feature is weighted such that its always better to eat a food, than to move close to it and stop.

5.2.2 Improved Heuristic

We hypothesize that using an overly simplified heuristic may cause poor performance. As an improvement to the basic heuristic, the improved heuristic adds an additional weighted feature representing the distance to the closest non-scared ghost. The features are weighted as in the basic heuristic, with the addition that it is never a good idea to die.

5.3 Observing enemies

As discussed in Section 2.1.2, and by Silver and Veness in [32], dealing with imperfect information is problematic. It is outside the scope of this study to find good ways of dealing with the Partial Observability property. Instead, we will make the assumption that what we can not observe (with high probability) does not exist. That is, when doing simulations, we will only simulate agents for which the probability of us having correct information—regarding their position at the start of the simulation—is high. At every depth of a simulation, the current game state is a direct result of the actions selected in previous states. Meaning that, if a simulation starts in game state s, we have an observation of s which we will base our simulation on. Denote this observation as s^o. While simulating at an increased depth, it is not possible to extract new observations that would reveal more information about s than s^o.

Naturally, this will have consequences when the simulation depth increases, as agents that have incorrectly been assumed to not exist, may eventually move close enough to other agents, such that what is perceived to be a good decision may not match the actual reality. It should be noted that this problem is by no means specific to MCTSBTP. Therefore, we suggest that if needed, a more sophisticated approach is found by exploring current research.

5.4 Cooperation between agents

As mentioned in Section 3.1.1, there is a synchronization problem if BTs are separated for individual agents in multi-agent systems [34]. However, MCTS can be extended to handle agent cooperation. Thus, cooperation aspects can be shifted from the BTs to the MCTS algorithm. One way to extend MCTS to cooperation problems, is by using a two-stage selection strategy. Additionally, node values are extended to contain both a team utility value, in addition to the agent utility value. Then, when making selections, for a node that is not fully explored, select the child with the highest team mean utility. Otherwise, select the node with the highest agent mean utility. It should be noted that the simulations should be tested for Pareto Optimality, to evaluate team performance. That is, the situation where an agent cannot improve its own utility without degrading the utility of the team. Cheng and Carver proposes a pocket algorithm in [7] to test for Pareto Optimality in MCTS. Pareto Optimality in cooperative MCTS is further discussed by Wang and Sebag in [33].

Chapter 6

Evaluation

We hypothesize that *the performance of MCTS can be improved by using a simple BT playout policy, instead of a standard random policy, for real-time adversarial games.* Using an experimental methodology, it is possible to investigate this hypothesis. In this chapter, we first disclose our testing environment in Section 6.1, followed by a presentation of the various experiments in Sections 6.2.1 - 6.2.3.

6.1 Statistical Significance

As the game is probabilistic, it may be difficult to distinguish between what is a feature of the algorithm, and what is random noise. Therefore, a large enough number of games should be played for each test, such that conclusions can be drawn statistically using large sample sizes.

In order to play a significant amount of games for each test in reasonable time, up to 48 games were played in parallel, distributed on two servers each having 2x6 cores with Hyper-Threading (24 threads per server). For each test, independent instances of the game were spawned using the same input. After evaluating an instance, the samples output was saved. After gathering up to 200 samples for each test, a statistical analysis was performed on the collected data.

6.2 Experiments

6.2.1 Establishing Fairness

To simplify further experiments, it is preferable to establish some degree of fairness in the testing environment (the game), and in the baseline strategy. Given that the game and baseline can be considered fair, a team's start location can be fixed to either side of the map, thus decreasing the number of variables to be tested. CTF Pac-Man utilizes a completely mirrored environment for both teams. That is, the layout of both sides of the map are mirrored, including agent spawn locations, and pellet/food locations. However, although a real-time game, agents select actions in a sequence of game states. Meaning, agent agt_0 will first select an action for game state S. Next, a successor state S' of S is generated by applying the action selected by agt_0. Now, agent agt_1 will select an action for state S'. Agents transform the game state in a fixed order. This order is determined at startup as shown in Algorithm 1.

Algorithm 1: Agent Order Selection. $Random[0, 1]$ refers to a pseudorandom number generator yielding floating point values between 0 and 1 uniformly. popb refers to a function that removes the back element in a collection and returns it.

```
1 redAgents ← $[ragt_0, ragt_1]$
2 blueAgents ← $[bagt_0, bagt_1]$
3 team1 ← redAgents $if$ $Random[0, 1] < 0.5$ $else$ blueAgents
4 team2 ← blueAgents $if$ team1 $is$ redAgents $else$ redAgents
5 order ← [team1.$popb()$, team2.$popb()$, team1.$popb()$, team2.$popb()$]
```

Definition 6.1 (Perfect strategy) Consider a finished game to be the complete sequence of actions—selected by each agent—at every time step. We say that a strategy ST is a *perfect strategy* iff, for every time step, actions are selected by ST such that in the terminal state, the maximum possible score, is always achieved (given the opponents' moves).

Definition 6.2 (Perfectly fair game) For a game to be *perfectly fair*, it must hold that the chance of winning—using a *perfect strategy*—is exactly equal, regardless of the order in which the agents move.

There is no known *perfect strategy* for CTF Pac-Man that is computationally traceable. Furthermore, this study concerns approximations. Therefore, we will not try to show that CTF Pac-Man is a *perfectly fair game*, but instead that it is *reasonably fair*.

Definition 6.3 (Non-trivial Strategy) We say that a strategy is a *non-trivial strategy*, if it can maintain a win rate of at least 99% against a strategy that selects actions at random.

Definition 6.4 (Reasonably fair game) We say that a game is a *reasonably fair game*, if there exists a *non-trivial strategy* that, when played against itself, maintains a mean score s, with $0 \leq abs(s) < \epsilon$, for some small ϵ.

Define a team selecting actions at random as RandomTeam. Let BTTeam be a team that selects actions according to a simple Behavior Tree. In order to test that the game is *reasonably fair*, we first test BTTeam against RandomTeam. It is expected that BTTeam will maintain a win rate of at least 99%. Given that

BTTeam is a *non-trivial strategy*, we will sample BTTeam against BTTeam. The range of the score for one sample is $[-20, 20]$. For a *reasonably fair game*, the scores should be normally distributed around zero. That is, the true mean is expected to be to zero. Based on this we can perform a one sample $t-test$. The discussed data is insufficient to decisively accept the null hypothesis ($\mu = 0$). However, we can give a confidence interval for the actual mean.

6.2.2 Establishing a baseline

Next, we will establish a baseline, using a random policy, which we will later use to evaluate our Behavior Tree policy. We start with the collected data and samples from the tests in Section 6.2.1 for BTTeam against BTTeam. Next, we play MCTSRP against BTTeam. Given these two groups of data, a 2 sided $t-test$ can be performed to test the null hypothesis (that the two groups are equal). Given that the null hypothesis gets rejected—as expected—we have a baseline for how MCTS, using a random playout policy, compares to BTTeam. As MCTSRP depends on many variables (tree depth, playout depth, UCT constant, number of iterations), it is conceivable that some combinations of variables may perform similarly to BTTeam. As such, a set of combinations may need to be tested.

6.2.3 Behavior Tree Playout Policy

To test your main hypothesis, we will test the Behavior Tree playout policy (MCTSBTP) against BTTeam. The collected data can then be compared to the baseline established in Section 6.2.2. Doing so, we can establish the performance difference in using the BT policy instead of a random policy, for playouts. The expected result is a significant improvement in win rate.

Regarding playouts, it should be noted that they are traditionally performed until a terminal state is reached [25]. However, since our game is partially observable—consisting of probabilistic properties—an observed game state may contain an error compared to the actual game state. This error is propagated as the observed game state is expanded in the simulations. Also, we only have an approximate model of our adversaries behaviors. Meaning, as simulations increase in depth, the error is likely to increase. Implying that, at some depth, the error will reach a threshold—where increasing the depth will yield an observed score for the simulation—that is significantly different to the actual score. As the selection policy only can take into account simulated scores, this may cause selections of nodes that are increasingly different from what they are perceived to be. In other words, the risk of picking a node, that decreases the score that the algorithm tries to maximize, increases. Thus, decreasing the overall win rate. Furthermore, as the error is not directly observable, it is difficult to know when to stop a simulation. Therefore, we will limit the depth of playouts. That is, we want to find a combination of iterations and depth, such that the depth is deep enough that Pac-Man does not commit suicide by walking into dead end alleys, while shallow enough such that the error caused by partial observability is negligible. And, we want to perform enough iterations to find moves that are globally good (not just locally).

Furthermore, as an objective of this study is to make modeling of agents easier, it is preferable if the BTs do not have to be very complex. Therefore, a simple BT (Figure 6.1) will be used in the experiments for MCTSBTP.

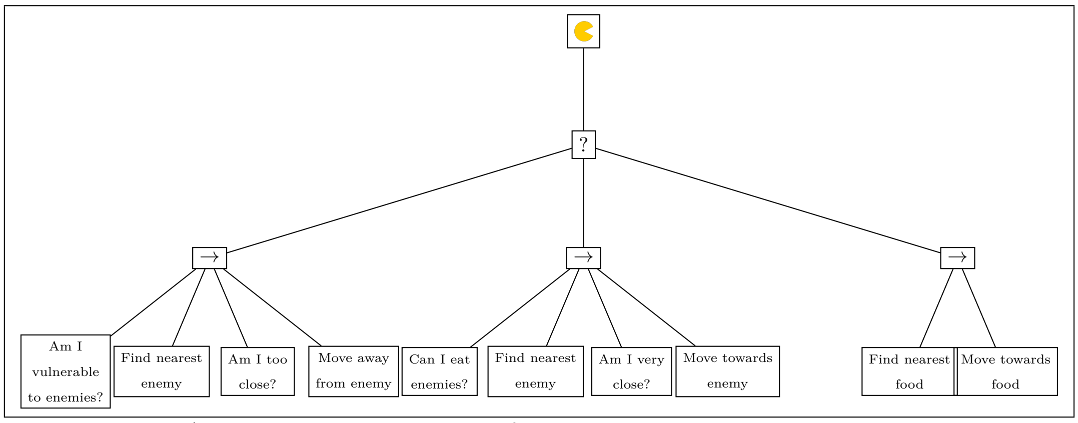

Figure 6.1: A simple Behavior Tree of Pac-Mans high-level decisions. The overall strategy encoded in the BT is to try to eat the closest food, unless an enemy is nearby. If so, Pac-Man should either try to eat it, if he has recently found a powerup and the enemy is very close, otherwise he should run away.

Chapter 7

Results and Discussion

In this chapter, we start by giving a presentation of the results of the three categories of experiments: Establishing Fairness, Establishing a Baseline, and the Behavior Tree Playout Policy, in Sections 7.1.1, 7.1.2, and 7.1.3, respectively.

The results are discussed in Section 7.2. Each category is discussed on its own in Sections 7.2.1, 7.2.2, and 7.2.3. Lastly, the combined results are discussed in Section 7.2.4.

7.1 Results

7.1.1 Establishing Fairness

This section presents the results of the experiments performed to establish fairness. That is, that the blue and red sides have an equal opportunity of winning. Observations gathered from gameplay are presented in Tables 7.1 - 7.6. Lastly, Table 7.7 shows a *t*-test that summarizes, and tests, the observations.

Table 7.1. BTTeam (red side) vs. RandomTeam, average score. The observed results after 50 games with BTTeam starting on the red side and RandomTeam on the blue side. The hypothesized mean score of 18 means that the team starting on the red side (BTTeam) is expected to win all games. The observed result was a mean of 18, as hypothesized.

Hypothesised mean	$\bar{x}$	Standard derivation	Sample size
18	18	0	50

Table 7.2. BTTeam (red side) vs. RandomTeam, win/loss distribution. The percentage of wins for the teams starting on the red and blue side of the map, respectively. BTTeam started on the red side. RandomTeam started on the blue side. BTTeam won all games.

Red wins (%)	Blue wins (%)	Ties (%)	Sample size
100	0	0	50

Table 7.3. RandomTeam (red side) vs. BTTeam, average score. The observed results after 50 games with RandomTeam starting on the red side and BTTeam on the blue side. The hypothesized mean score of -18 means that the team starting on the red side (RandomTeam) is expected to lose all games. The observed result was a mean of -18, as hypothesized.

Hypothesised mean	$\bar{x}$	Standard derivation	Sample size
-18	-18	0	50

Table 7.4. RandomTeam (red side) vs. BTTeam, win/loss distribution. The percentage of wins for the teams starting on the red and blue side of the map, respectively. RandomTeam started on the red side. BTTeam started on the blue side. BTTeam won all games.

Red wins (%)	Blue wins (%)	Ties (%)	Sample size
0	100	0	50

Table 7.5. BTTeam (red side) vs. BTTeam, scores. The observed results after 200 games with BTTeam playing against itself. The hypothesized mean score of 0 means that both sides are expected to score similarly, and win as many games. Although the observed mean is close to zero ($5.5 \cdot 10^{-2}$), both the minimum and maximum scores are -18 and 18, respectively. Consequently, resulting in a standard deviation of 7.76.

Hypothesised mean	$\bar{x}$	min	max	SD	Sample size
0	$5.5 \cdot 10^{-2}$	-18	18	7.76	200

Table 7.6. BTTeam (red side) vs. BTTeam, win/loss distribution. The percentage of wins for the teams starting on the red and blue side of the map, respectively. BTTeam played both the red and blue sides. The difference in win rates is within 1%.

Red wins (%)	Blue wins (%)	Ties (%)	Sample size
35	36	29	200

Table 7.7. BTTeam (red side) vs. BTTeam, *t*-test. A one sample *t*-test of the observed results of BTTeam vs. BTTeam over 200 games. The two-tailed *P*-value equals 0.92. Thus, we fail to reject the null hypothesis at a 95% confidence level. The 95% confidence interval is (-1.02, 1.13).

μ_0	$\bar{x}$	SD	SEM	*P*-VALUE	CI	n	CL (%)
0	$5.5 \cdot 10^{-2}$	7.76	0.55	0.92	-1.02–1.13	200	95

7.1.2 Establishing a Baseline

This section presents the results of the experiments performed to establish a baseline. All tests are performed for Monte Carlo Tree Search with a Random playout Policy (MCTSRP), playing against a team of agents making decisions based on Behavior Trees (BTTeam). Figures 7.1 and 7.2 shows the distribution of wins, losses, and ties, as the playout depth and the number of iterations increases. We denote the configuration of MCTSRP that performs n iterations, each with a playout depth of d, as MCTSRP(n,d). The scores of the best-observed configuration, MCTSRP(640,4), is shown in Table 7.8.

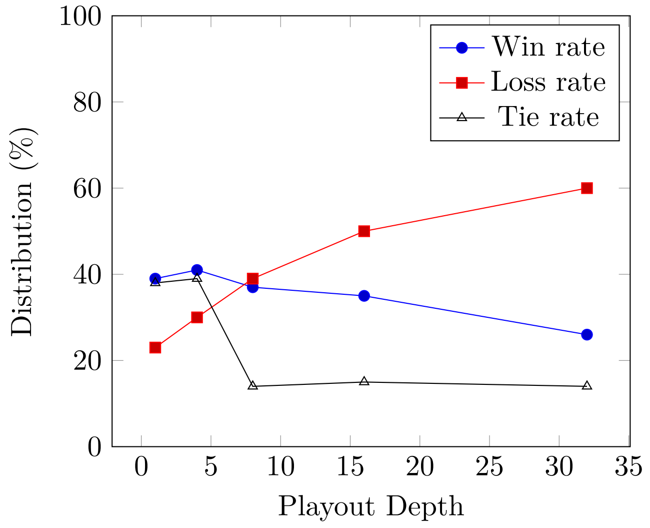

Figure 7.1: MCTSRP(160,_) vs. BTTeam, win/loss distribution. Win, loss, and tie rates for MCTSRP, when played against BTTeam over 200 games. The number of iterations used by MCTSRP was fixed at 160. The playout depth (1-32) is shown on the x-axis. The observed result is a maximum win rate at depth 4. As the depth increases past this point, the win rate decreases, while the loss rate increases, see Section 7.2.2.

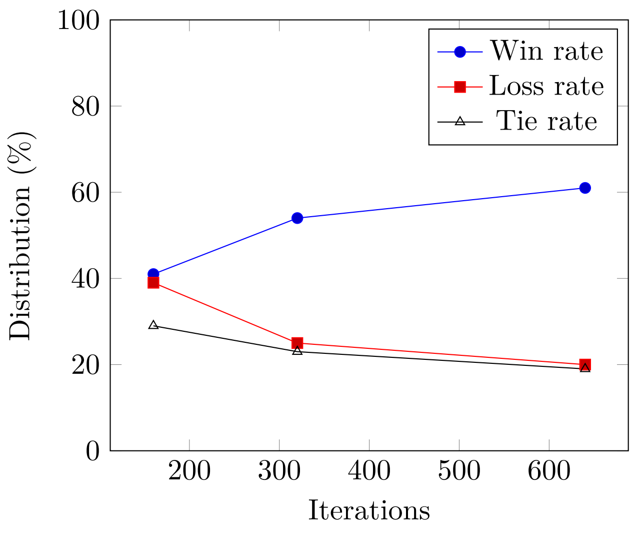

Figure 7.2: MCTSRP(_,4) vs. BTTeam, win/loss distribution. Win, loss, and tie rates for MCTSRP, when played against BTTeam over 200 games. The number of iterations (160-640) used by MCTSRP is shown on the x-axis. The search depth was fixed at 4. The results show a growth in win rate up to 320 iterations. Although the win rate increases after this point, the rate at which it increases is significantly lower.

Table 7.8. MCTSRP(640,4) vs. BTTeam, observed results. The average score and the distribution of wins, losses, and ties observed over 200 games between MCTSRP(640,4) and BTTeam. The results show a significantly favored win rate (61%) for MCTSRP. Nevertheless, the average score is -0.21.

Average Score	Wins (%)	Losses (%)	Ties (%)	Sample size
−0.21	61	20	19	200

7.1.3 Behavior Tree Playout Policy

This section presents the results of the experiments performed to test Monte Carlo Tree Search using a Behavior Tree Playout Policy (MCTSBTP). The adversary is BTTeam for all tests. Figure 7.3 shows win rates using a basic heuristic. Figures 7.4 and 7.5 shows how the distribution of wins, losses and ties changes, as the number of iterations, and playout depth increases, for MCTSBTP. Figure 7.6 shows how the average score of MCTSBTP changes as the number of iterations increases. Table 7.9 presents the scores of the best-observed configuration, MCTSBTP(320,4). Lastly, Table 7.10 presents the observed changes in scores, by using the Behavior Tree Policy, instead of the Random Policy, for MCTS.

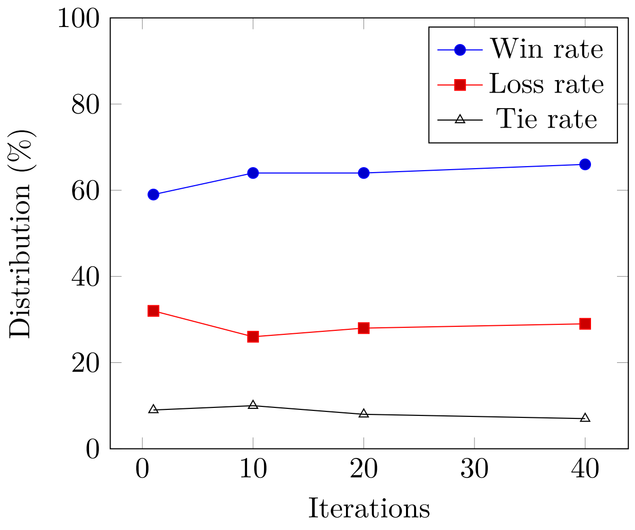

Figure 7.3: MCTSBTP(_,1) vs. BTTeam (initial heuristic), win/loss distribution. Win, loss, and tie rates for MCTSBTP, when played against BTTeam over 200 games. The number of iterations (1-40) used by MCTSBTP is shown on the x-axis. The search depth was fixed at 1. The observed result is a static distribution of wins, losses, and ties, as the number of iterations increases beyond 10, when using the initial heuristic.

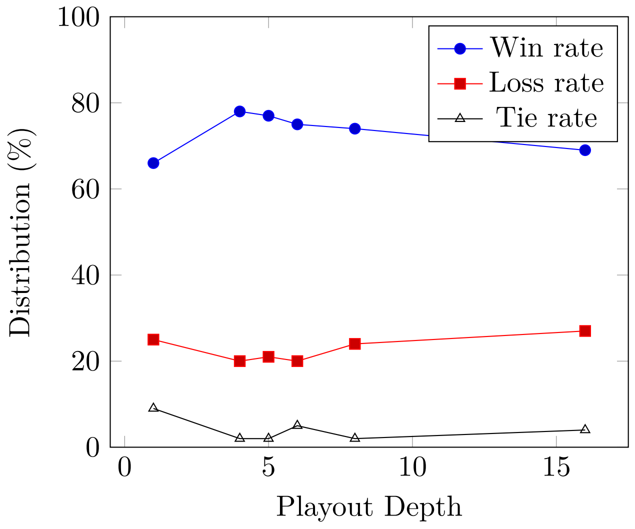

Figure 7.4: MCTSBTP(8,_) vs. BTTeam, win/loss distribution. Win, loss, and tie rates for MCTSBTP, when played against BTTeam over 200 games. The number of iterations used by MCTSBTP was fixed at 8. The playout depth (1-16) is shown on the x-axis. The observed result is an increase in win rate until playout depth 4. After which point, the win rate decreases while the loss rate increases.

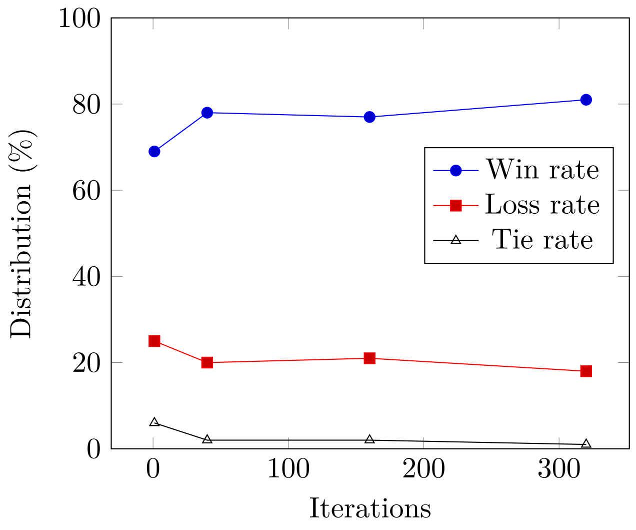

Figure 7.5: MCTSBTP(_,4) vs. BTTeam, win/loss distribution. Win, loss, and tie rates for MCTSBTP, when played against BTTeam over 200 games. The number of iterations (40-340) used by MCTSBTP is shown on the x-axis. The search depth was fixed at 4. The observed result is a significant increase in win rate for the first 40 iterations. After which point, the win rate slowly increases until 320 iterations, while the loss rate slowly decreases.

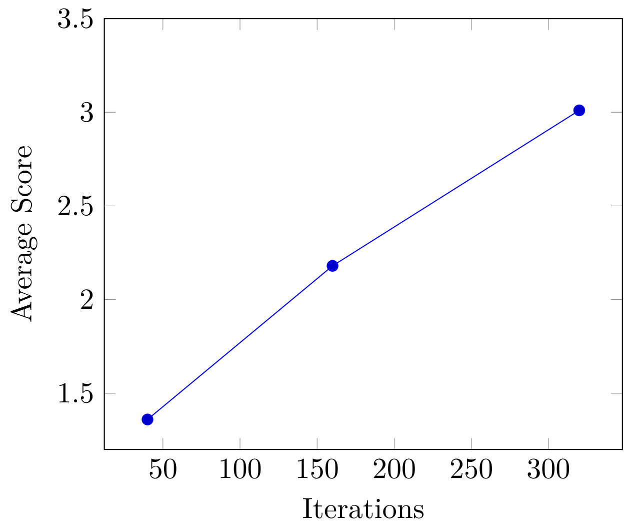

Figure 7.6: MCTSBTP(_,4) vs. BTTeam, average score. The Average Score for MCTSBTP, when played against BTTeam over 200 games. The number of iterations (40-340) used by MCTSBTP is shown on the x-axis. The search depth was fixed at 4. The observed result is a steadily growing increase in average score, as the number of iterations increases.

Table 7.9. MCTSBTP(320,4) vs. BTTeam, win/loss distribution. The average score and the distribution of wins, losses, and ties observed over 200 games between MCTSBTP(320,4) and BTTeam. The results show a significantly favored win rate (81%) for MCTSBTP. Furthermore, the average score of 3.01 is significantly greater than 0.

Average Score	Wins (%)	Losses (%)	Ties (%)	Sample size
3.01	81	18	1	200

Table 7.10. MCTSBTP(320,4) and MCTSRP(640,4), result differences. The observed changes in scores, by running MCTSBTP(320,4), instead of MCTSRP(640,4). The changes are presented as the signed difference, calculated by subtracting MCTSRP's observations from MCTSBTP's. The results show a significant favor in the average score (an increase of 3.22) and win rate (81% instead of 61%) for MCTSBTP. The number of tied games is significantly lower (1% instead of 19%). The loss rate is similar at 18% and 20%, comparatively.

ΔScore	ΔWins (%)	ΔLosses (%)	ΔTies (%)
3.22	20	-2	-18

7.2 Discussion

7.2.1 Establishing Fairness

BTTeam vs. RandomTeam

Tables 7.1 and 7.2 shows that BTTeam maintains a 100% win rate over 100 games, when playing against RandomTeam. Each game scoring 18 or -18, when playing on the red and blue side, respectively. This is the lowest score that results in a terminal winning state for any amount of remaining game time, on the given layout. Therefore, we conclude BTTeam to be a *non-trivial strategy* by definition 6.3.

BTTeam vs. BTTeam

Table 7.6 shows that the sample mean is close to zero ($5.5 \cdot 10^{-2}$) for BTTeam vs. BTTeam. However, the minimum and maximum scores are both -18 and 18, respectively, contributing to a large sample variance. The standard deviation is 7.76. As a result, as can be seen in Table 7.7, we fail to reject the null hypothesis $\mu_0 = 0$ at the 95% confidence level. However, we can establish a 95% confidence interval of (-1.02, 1.13). The win rate after 200 samples is 35% for the red team and 36% for the blue team.

We conclude that the above observations do not show a significant bias towards one side of the map for BTTeam. Thus, in conjunction with the conclusion that BTTeam is a *non-trivial strategy*, we conclude that CTF Pac-Man is *relatively fair* by definition 6.4. Therefore, we omit testing agents for both sides in future tests and will simply refer to the agents by their names.

7.2.2 Establishing a Baseline

Figure 7.1 shows that the maximum win rate is achieved at playout depth 4. To analyze this, we need to consider the Partial Observability (PO) property of CTF Pac-Man. For any given game state observed by an agent, there is a difference between the state seen by that agent, and the actual game state, e.g., as illustrated in Figure 7.7. Furthermore, when doing simulations, an agent can not request new observations. Meaning, all explored successor states in a simulation are expanded from the original observation. Now, because the initial observation contains an error, this error is propagated to all successor states. Thus, as the depth of playouts increase, so does the probability that the error will cause our agent to make an incorrect decision.

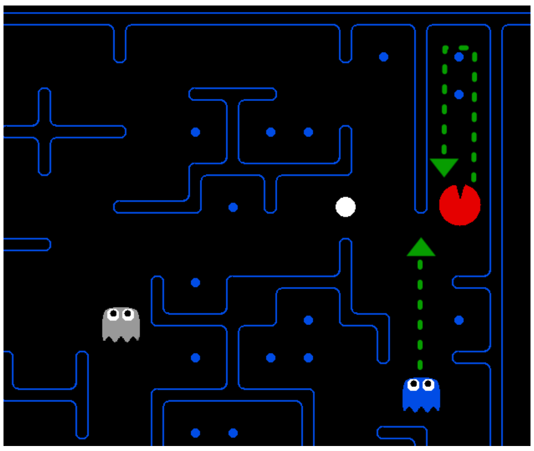
Figure 7.7: Suppose Pac-Man makes an observation of a ghost being a distance of 15 tiles away, illustrated by the gray ghost. However, suppose the ghost is actually 6 tiles away, illustrated by the blue ghost. Now, a simulation might cause Pac-Man to think he can enter the alley, collect the food, and get out, giving the simulation a high score. Whereas in actuality, Pac-Man would not be able to get out.

Consequently, it is not possible to perfectly predict where the ghosts will be, so increasing the depth increases the risk that a ghost will be in Pac-Mans path in the actual successor state, whereas it is not in the simulation. The maximum distance of which Pac-Man can make accurate observations is 5. Figure 7.1 shows that increasing the playout depth past this decreases the win rate. As playouts are performed from leaves, and not the root, the maximum playout depth that results in a possibility that Pac-Man moves a distance of at most 5 (per iteration) is 4. Therefore, we will fix the playout depth to 4.

Figure 7.2 shows the win rate increases rapidly with the number of iterations, up until 320. After which, the win rate increases slowly until 640, which is the largest number of iterations we may run within our time budget. It is expected that the win rate may increase more. However, there is a limit to how high it may go, consider the PO property. Also, since the desirability comparison between two states is performed by a heuristic, this further limits the maximum achievable win rate. Thus, the win rate is expected to increase (at a diminished rate), until reaching the upper bound imposed by these properties.

Table 7.8 shows a win rate of 61%, while maintaining a loss rate of 20%. Although a significantly higher win rate than loss rate is retained, the average score is -0.21. Indicating that, even though wins are more frequent than losses, the magnitude of the score in MCTSRP's victories is significantly smaller than in its defeats. This is not entirely unexpected for a random policy. As the map is mirrored, there are exactly two cases in which a game can be lost: Pac-Man either died, or found a longer path than the adversary Pac-Man. We assume the given heuristic is balanced such that Pac-Man does not die unnecessarily. Then, the prior case is negligible. Furthermore, it is intuitively obvious that a random agent may sometimes explore paths that are bad. Most times, exploring bad paths will not matter, as other configurations will be evaluated in following iterations, and the algorithm can recover. However, it is conceivable that, given a limited number of iterations and search depth, it may not be possible to recover from all configurations.

7.2.3 Behavior Tree Playout Policy

Initial Heuristic

A reader may expect the results of MCTSBTP(1,1) vs. BTTeam to be similar to those of BTTeam vs. BTTeam, provided that they both use the same Behavior Tree. The observed results show a significant increase in win rate, 59%, as compared to 36%. To explain this, we need to consider how one iteration of MCTSBTP differs from simply doing one evaluation of the Behavior Tree. In the expansion step of MCTS, all children are added to the tree. That is, when performing one iteration, up to five children may be evaluated (one for each possible move). Then, for each child, a playout of depth 1 consists of performing the given action, followed by running the BT once. That is, a series of exactly two actions are performed for each child, as compared to exactly one action for BTTeam vs. BTTeam. Furthermore, it should be noted that while the increase in win rate is significant, the decrease in loss rate is not. Meaning, the increase in win rate is the result of a significantly lower amount of games resulting in draws.

Figure 7.3 shows win rate as a function of the number of iterations. The expected result is an increase in win rate as the number of iterations grows. The observed result is a static win rate. We hypothesize this to be the result of the heuristic used being too simple. That is, if the heuristic can not be used to indicate which of two game states is better, then increasing the number of iterations will not yield a better result. Meaning, the heuristic applies an upper bound on the maximum achievable score. We test this hypothesis by repeating the tests using an improved heuristic (see Section 5.2).

Improved Heuristic

Following the use of the initial heuristic in the first set of MCTSBTP tests (Figure 7.3), the improved heuristic was used in all subsequent tests. Using the improved heuristic, the win rate and average score both increases as the number of iterations increases (Figures 7.5 and 7.6). Therefore, we conclude the quality of the heuristic to be of importance.

Figure 7.4 shows the distribution of wins, losses, and ties, as the playout depth changes. As with MCTSRP, the maximum win rate is achieved at depth 4. And again, the win rate decreases as the playout depth is increased beyond this point. This behavior is expected, as both MCTSBTP and MCTSRP use the same heuristic. More importantly, the PO property still holds. Thus, we confirm our previous observation. And again fix the playout depth at 4.

Figure 7.5 shows the distribution of wins, losses, and ties, as the number of iterations changes. The maximum number of iterations allowed by our time limit is 320. A fast-growing increase in win rate is observed during the first 40 iterations, after which point the growth is slow. Again, this behavior is comparable to what was observed for MCTSRP.

7.2.4 Comparison of MCTSRP and MCTSBTP

Table 7.10 shows the difference in results between MCTS using the proposed Behavior Tree Playout Policy, and a traditional Random Policy. The number of iterations (and to some extent the search depth) that is explorable, given the computational time budget, is significantly larger for MCTSRP than MCTSBTP. Nonetheless, the experimental results show a clear increase in both score and win rate for MCTSBTP. Since the selection, expansion and back propagation steps are the same for both approaches, the difference must stem from the difference in Playout Policy. Thus, we confirm our hypothesis that Monte Carlo Tree Search can be improved using Behavior Trees for high-level decisions.

Next, consider an instance of CTF Pac-Man without any ghosts. In such an instance, finding an optimal path to collect all pellets for all maps, can be reduced to the Traveling Salesman Problem. However, because the actual maps in CTF Pac-Man contains choke points and are generally small, the practical difficulty in finding close-to-optimal paths is significantly reduced. Therefore, we conjecture that MCTSRP is permitted enough iterations to find a close-to-optimal path to pellets, if the adversary ghosts were to be disregarded. Consequently, the difference in score must mean that MCTSBTP is better at avoiding dropping pellets than MCTSRP. As the only way of dropping pellets is to die by being tagged by an opposing ghost, we conclude that MCTSBTP is able to better predict the movement of adversarial agents. Thus, we conclude that MCTSBTP is able to extend MCTS with heuristic predictions of adversarial actions.

Chapter 8

Conclusions and Further Work

In this study, we have shown how Monte Carlo Tree Search can be combined with Behavior Trees, resulting in the proposed MCTSBTP algorithm described in Chapter 5. Our experimental results show that MCTSBTP can achieve both a higher win rate and mean score (Section 7.1.3), than MCTS using a Random Policy, given the same computational time budget. Furthermore, we have concluded that MCTSBTP can add heuristic predictions of adversarial actions to MCTS, following the discussion in Section 7.2.4. By combining MCTS and BTs, the proposed algorithm both includes the tree search element of Monte Carlo Tree Search, finding globally good action, while maintaining the ease of use, easy modeling, and maintainability of Behavior Trees.

8.1 Further Work

In the proposed algorithm, for any given environment, two inputs are required: a behavior tree that models the agents' behaviors in said environment, and a heuristic which, given a game state, returns a numeric value. This numeric value should have a direct correlation between the magnitude of the value and how desirable the game state is. How the heuristic is designed depends on the environment. However, in the general case, it is conceivable that it combines different features of the game state and assigns weights to each considered feature. Merrill discusses how to build utility decisions into existing Behavior Trees in [22]. Denote such a Behavior Tree—with utility decisions—as a Utility Tree. Now, consider a type of Utility Tree, similar to a regular Behavior Tree, but with an additional numerical value associated to nodes, which indicates how desirable a descendant leaf of that node is. Here, leaves correspond to available actions. Given that a heuristic is designed by looking at features and associating weights, it is directly correlatable to building utilities in such a UT. As such, the heuristic could instead be encoded into the BT, resulting in a UT. Given a UT, the proposed algorithm can be modified as shown in algorithm

2.

Algorithm 2: Suggested modified algorithm, MCTSUT.

```
itr $\leftarrow 0$
while itr $< N$ do
    1. Selection: as before.
    2. Expansion: do not expand the selected node. Instead, consider this
       node as the child to be evaluated in step 3. Denote this child with $C$.
    3. Simulation: start by walking from the root to the selected child $C$.
    depth $\leftarrow 0$
    while depth $< M$ do
        a. Select the descendant $D$ of $C$, where $D$ is the leaf with the highest
           score. If there are no leaves, let $D = C$.
        b. Starting in $D$, evaluate $UT$.
        c. Add each child state of $D$.
        d. Back propagate up to $C$.
        depth $\leftarrow$depth $+1$
    4. Back propagation: as before.
    itr $\leftarrow$itr $+1$
```

As expansions are now handled in the simulation step, if the simulation depth is greater than 1, simulations will cause nodes to be expanded in a depth-first manner. Therefore, the exploration vs. exploitation constant—used by UCT (or similar) in the selection step—should be reduced by the inverse of the simulation depth, such that the breadth of the tree is also explored.

Arguably, such an algorithm, which encodes the heuristic in the BT, greatly increases the ease of which an environment can be modeled. We suggest this as future work. This would entail comparing the modified algorithm to the one presented in this study. Lastly, how closely the design of such an algorithm should mirror Algorithm 2 needs further correctness and computational complexity arguments.

Bibliography

[1] Rajeev Agrawal. Sample mean based index policies by o (log n) regret for the multi-armed bandit problem. *Advances in Applied Probability*, 27(04):1054–1078, 1995.

[2] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

[3] 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.

[4] Sébastien Bubeck, Nicolo Cesa-Bianchi, et al. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *Foundations and Trends in Machine Learning*, 5(1):1–122, 2012.

[5] Alex Champandard. Understanding behavior trees. *AiGameDev.com.* Last visited June 2017.

[6] Guillaume M.J-B. Chaslot, Mark H.M. Winands, H.J van den Henrik, Jos W.H.M. Uiterwijk, and Bruno Bouzy. Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation*, 4(03):343–357, 2008.

[7] CheeChian Cheng and Norman Carver. Cooperative games with monte carlo tree search. In *Proceedings on the International Conference on Artificial Intelligence (ICAI)*, page 99. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2015.

[8] M. Colledanchise and P. Ögren. How behavior trees modularize hybrid control systems and generalize sequential behavior compositions, the subsumption architecture, and decision trees. *IEEE Transactions on Robotics*, 33(2):372–389, April 2017.

[9] Michele Colledanchise and Petter Ögren. How behavior trees modularize robustness and safety in hybrid systems. In *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, pages 1482–1488. IEEE, 2014.

[10] Jan Kristian Haugland. On-line encyclopedia of integer sequences, a094777, 2004.

[11] Damian Isla. Handling complexity in the halo 2 ai. In *Game developers conference*, volume 12, 2005.

[12] Damian Isla. Halo 3-building a better battle. In *Game Developers Conference*, 2008.

[13] Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1):99–134, 1998.

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

[15] Levente Kocsis, Csaba Szepesvári, and Jan Willemson. Improved monte-carlo search. *Univ. Tartu, Estonia, Tech. Rep*, 1, 2006.

[16] Tze Leung Lai and Herbert Robbins. Asymptotically efficient adaptive allocation rules. *Advances in applied mathematics*, 6(1):4–22, 1985.

[17] Chong-U Lim, Robin Baumgarten, and Simon Colton. Evolving behaviour trees for the commercial game defcon. In *European Conference on the Applications of Evolutionary Computation*, pages 100–110. Springer, 2010.

[18] Michael Lederman Littman. *Algorithms for sequential decision making.* PhD thesis, Brown University, 1996.

[19] A Bryan Loyall. *Believable agents: building interactive personalities.* PhD thesis, Mitsubishi Electric Research Laboratories, 1997.

[20] Carol Luckhart and Keki B Irani. An algorithmic solution of n-person games. In *AAAI*, volume 86, pages 158–162, 1986.

[21] Alejandro Marzinotto, Michele Colledanchise, Christian Smith, and Petter Ögren. Towards a unified behavior trees framework for robot control. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pages 5420–5427. IEEE, 2014.

[22] Bill Merrill. Building utility decisions into your existing behavior tree. *Game AI Pro: Collected Wisdom of Game AI Professionals*, page 127, 2013.

[23] Ian Millington and John Funge. *Artificial intelligence for games*. CRC Press, 2016.

[24] Santiago Ontanón. The combinatorial multi-armed bandit problem and its application to real-time strategy games. In *Proceedings of the Ninth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pages 58–64. AAAI Press, 2013.

[25] Santiago Ontanón, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill, and Mike Preuss. A survey of real-time strategy game ai research and competition in starcraft. *IEEE Transactions on Computational Intelligence and AI in games*, 5(4):293–311, 2013.

[26] Ricardo Palma, Pedro Antonio González-Calero, Marco Antonio Gómez-Martín, and Pedro Pablo Gómez-Martín. Extending case-based planning with behavior trees. In *Twenty-Fourth International FLAIRS Conference*, 2011.

[27] Tom Pepels and Mark HM Winands. Enhancements for monte-carlo tree search in ms pac-man. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, pages 265–272. IEEE, 2012.

[28] Joelle Pineau, Geoffrey Gordon, and Sebastian Thrun. Anytime point-based approximations for large pomdps. *Journal of Artificial Intelligence Research*, 27:335–380, 2006.

[29] Spyridon Samothrakis, David Robles, and Simon Lucas. Fast approximate max-n monte carlo tree search for ms pac-man. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(2):142–154, 2011.

[30] Frederik Christiaan Schadd. *Monte-Carlo search techniques in the modern board game Thurn and Taxis*. PhD thesis, Maastricht University, 2009.

[31] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[32] David Silver and Joel Veness. Monte-carlo planning in large pomdps. In *Advances in neural information processing systems*, pages 2164–2172, 2010.

[33] Weijia Wang and Michele Sebag. Hypervolume indicator and dominance reward based multi-objective monte-carlo tree search. *Machine learning*, 92(2-3):403–429, 2013.

[34] Ben G Weber, Peter Mawhorter, Michael Mateas, and Arnav Jhala. Reactive planning idioms for multi-scale game ai. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, pages 115–122. IEEE, 2010.