THE UNIVERSITY of York

Using heuristics for Monte Carlo Tree Search

Submitted in part fulfillment for the degree of
MSc in Software Engineering
Department of Computer Science
University of York

Author:
Karol GÓRNICKI

Supervisor:
Dr. Daniel KUDENKO

September 12, 2012

Number of words: 20,547*, as counted by Texmaker.*
This includes the body of the report but excludes TOC, LOF, LOT, LOA and Bibliography.

Abstract

Monte Carlo Tree Search has been shown to be highly successful on very challenging games, such as, Go. The method is based on randomly sampling sequences of moves to the end of the game and assigning a score to the next move based on the outcomes of these samples. Experiments with Go have shown that biasing the sampling with heuristic expert knowledge did not help but rather hurt performance.

In this paper, we exercise the application of heuristic evaluation to Monte Carlo Tree Search algorithm for the ancient game PahTum. At first, we construct brand new heuristics and implement AI agent solely relying on it without performing any roll-outs. In the match of 100 games it outperformed UCT algorithm proposed by Kocsis and Szepesvári by winning 99% of the games. Then we demonstrate various techniques of how aforementioned heuristic evaluation function can be incorporated to the Monte Carlo Tree Search algorithm in order to improve the quality of play. The most successful implementation in the match of 100 games won over heuristic-based agent 87% of the games and 100% against UCT with 5 times smaller number of roll-outs in the match of 400 games.

Statement of Ethics

This project does not feature human interaction, private data collection and manipulation and involves no actions that can cause direct or indirect harm.

Consequently there are no ethical implications in regard to this project.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Background Information

It all started back in 1944 when John von Neumann and Oskar Morgenstern published *Theory of Games and Economic Behavior* [27] to lay the foundation of new intersection between mathematics and economy. They largely presented a novel approach which allows for mathematical interpretation of common occurrences in environments where goals of interested sides are very often in conflict. Shortly after, in 1950, John F. Nash published his proof of equilibrium points in n-person non-cooperative games [20] for which he was recognized in 1994 with a Nobel Prize. It opened a floodgate to a series of refinements and the game theory has been of great interest to scientists ever since.

Since the inception of computers scientists have aimed to engage them in various decision making processes. At first they served as simple calculation devices, however, together with recent increase of computational power they have started to be frequently used in more sophisticated analyses.

Economies are no more than multi-player games where individuals sometime play as a team for common purpose, some other time only for their own good. Occurrences of events during some of those games can be modeled as a tree in a sense of graph theory. Due to their simple implementation into computer architecture they quickly became one of the main focus of artificial intelligence research. Over the past few decades scientists came up with really good algorithm that addressed many concerns. However, some have remained an open problem, such as, the game of Go, where human players are by far better than any computer program.

In 2006 Rémi Coulom presented a new algorithm called Monte Carlo Tree

Search [12] (hereafter referred to as MCTS) which in the next few years has revolutionized the adversarial search domain. In this paper we investigate whether using heuristics improve the performance of MCTS, and if so, how it should be incorporated in order to maximize the efficiency of MCTS.

1.2 Motivation

MCTS algorithm proved to be profoundly successful in many games and planning tasks. By obtaining the highest ratings in Go competitions among other AIs has won worldwide recognition and originated prolific following of its variations. Plenty of which became very successful in various domains.

The interest of scientific community can be denoted by frequency of publishing, on average, one scientific paper regarding MCTS method per week recently [6].

In this project we intend to elaborate on existing algorithms, attain better understanding of them and why they succeed and fail in respect to the core notion of the algorithm as well as circumstances caused by the environments the AI acts in. At the end we present how encountered pitfalls can be addressed by employing novel approaches.

1.3 Report Structure

The remainder of this dissertation is organized as follows.

Chapter 2 – Literature Review
Elaborates on the domain of adversarial search problem and the game Pah-Tum with its properties.

Chapter 3 – Development Process
Discusses engineering approach adopted for the purpose of this project.

Chapter 4 – Requirements Analysis
Defines requirements of this project.

Chapter 5 – Design and Implementation
Documents the design of the test bed, heuristics, and all MCTS agents utilizing it. Various enhancements are examined to enlighten the reader about their background motivation.

Chapter 6 – Evaluation
Analyzes the results obtained in various tests, including their statistical con-

fidence discussion.

Chapter 7 – Conclusions and Future Work
Summarizes the project and spotlights further promising development paths of MCTS research.

1.4 Editorial Decisions

Male pronouns are used to refer to agents throughout the dissertation. The reverse female convention would have made the document less readable. The reader is urged not to read patriarchal intentions into this choice.

Chapter 2

Literature Review

In this chapter we introduce background information in adversarial search problems domain. The brief account of history is given of developed solutions with assessment of how well they address faced challenges. The emphasis is drawn to the most successful approaches. Some of their concepts can be adopted to the emerging Monte Carlo Tree Search (MCTS) algorithms.

After covering classic approaches we move on to the family of MCTS algorithms which are centerpiece of this thesis. Broad overview is provided in order to familiarize the reader with the concept that stands behind the theory and indicate where to seek the room for improvements in areas yet unexplored by the research community.

At the end we elaborate on games of our interest that served as a testing environment for purpose of this project.

2.1 Games

Adversarial search problems are broad domain, so let us start by dispelling all ambiguities. Due to the purpose of this project we only focus on one, specialized kind of games – two player, zero sum games with perfect information[1]. In accordance with Leyton-Brown and Shoham [18], these games in an extensive form can be interpreted as a tree in the sense of graph theory (commonly called game tree), in which each node represents the choice of one of the players, each edge represents a possible action and the leaves represent final outcomes over which each player has a utility function. Formally we can define them as follows [18].

[1]From this point whenever later we will refer to the *game* we implicitly mean this sort of game

Definition 1. ***(Perfect-information game)*** *A (finite) perfect-information game (in extensive form) is a tuple* $G = (N, A, H, Z, \chi, \rho, \sigma, u)$*, where:*

- N *is a set of* n *players;*
- A *is a (single) set of actions;*
- H *is a set of non-terminal choice nodes;*
- Z *is a set of terminal nodes, disjoint from* H*;*
- $\chi : H \mapsto 2^A$ *is the action function, which assigns to each choice node a set of possible actions;*
- $\rho : H \mapsto N$ *is the player function, which assigns to each non-terminal node a player* $i \in N$ *who chooses an action at that node;*
- $\sigma : X \times A \mapsto H \cup Z$ *is the successor function, which maps a choice node and an action to a new choice node or terminal node such that for all* $h_1, h_2 \in H$ *and* $a_1, a_2 \in A$*, if* $\rho(h_1, a_1) = \rho(h_2, a_2)$ *then* $h_1 = h_2$ *and* $a_1 = a_2$*;*
- $u = (u_1, ..., u_n)$*, where* $u_i : Z \mapsto \mathbb{R}$ *is a real-valued utility function for player* i *on the terminal nodes* Z*.*

Each player is equipped with a collection of strategies. By the term strategy we understand collection of single actions that can be taken for each node that belongs to him. Formal definition is as follows [18].

Definition 2. ***(Pure strategies)*** *Let* $G = (N, A, H, Z, \chi, \rho, \sigma, u)$ *be a perfect-information extensive-form game. Then the pure strategies of player* i *consists of the Cartesian product* $\prod_{h \in H, \rho(h)=i} \chi(h)$*.*

For each node the player is able to determine a non-empty set of candidate actions called best responses which ultimately lead to better utility evaluation than any others (utility-maximizing action). Adjusting players strategy becomes even easier in terms of perfect-information games where all possible maneuvers of the opponent are known from the very beginning. In order to provide a formal description [18] we use the notion of strategy without agent i's strategy:

$$s_{-i} = (s_1, ..., s_{i-1}, s_{i+1}, ..., s_n)$$

thus

$$s = (s_i, s_{-i}).$$

Definition 3. ***(Best response)***[2] *Player i's best response to the strategy profile* s_{-i} *is a mixed strategy* $s_i^* \in S_i$ *such that* $u_i(s_i^*, s_{-i}) \geq u_i(s_i, s_{-i})$ *for all strategies* $s_i \in S_i$*.*

[2]strategic profile is a choice of strategy for each agent

Fully expanded game tree enables to review the history of actions that led to the position of our interest since all nodes, with an exception from the root, have ancestors. Traversing the tree from the nodes that immediately precede leaves (terminal states) upward (in terms of hierarchical structure of the tree) to the root by considering "bottom-most" sub-trees enables us to select best responses for each node. Players make alternate plays, therefore at each time when we shift the investigated level we switch priorities to select best response in the interest of the agent currently making decision. At this point we assume that players will follow previously discovered best responses. By doing so we can find a sequence of best responses for each player which leads us to finding the Nash equilibrium. This reasoning proves a theorem coined by Zermelo[3], presented below [18]. Nash proved in 1950 that all n-person non-cooperative games have equilibrium [20].

Theorem 1. *Every finite perfect-information game in extensive form has a pure-strategy Nash equilibrium.*

In principle algorithm that follows presented strategy is *backward induction*, however in case of domain of players narrowed down to two is commonly regarded as *minimax algorithm*. In order to find out more about it see subsequent section.

2.2 Minimax Algorithm and Its Enhancements

Due to computational simplicity of aforementioned procedure, theoretically speaking, every two player zero sum game with perfect information is solvable and *minimax algorithm* addresses it. In a sense of Definition 3, the best response can be interpreted as a selection of branch which leads to the higher or smaller pay-off depending on the perspective of which player we look. Minimax value of the terminal state is its utility, MaxValue prefers to move to a state of maximum value whereas MinValue favors minimum value. Full algorithm is presented as follows [23].

Algorithm 2.1 Minimax algorithm

```
1: function MiniMax(node h) return a
2:     return arg max_{a∈χ(h)} MinValue(σ(h, a))
3: end function
```

[3] In 1913 Zermelo introduced the notion of a game tree and backward induction, for more information see [24]

Algorithm 2.1 Minimax algorithm (continued)

```
4: function MinValue(node h) return u(h)
5:     if h ∈ Z then
6:         return u(h)
7:     end if
8:     for all a ∈ χ(h) do
9:         v ← Min(v, MaxValue(σ(h, a)))
10:    end for
11:    return v
12: end function

13: function MaxValue(node h) return u(h)
14:    if h ∈ Z then
15:        return u(h)
16:    end if
17:    for all a ∈ χ(h) do
18:        v ← Max(v, MinValue(σ(h, a)))
19:    end for
20:    return v
21: end function
```

Presented algorithm is able to tackle every game, however number of nodes in the search space grows exponentially in the depth of the tree[4] which can quickly outnumber computational feasibility of contemporary machines. In order to address this issue few techniques were developed where combined altogether provide various results (in certain games they perform very well, however there are few where they fail against human players). The core notion of the algorithm stays unchanged when all those improvements aim to reduce the size of the search space, sometimes by compromising the precision by introducing some level of uncertainty. Let's elaborate them one by one [23]:

- **Alpha-beta pruning.** The notion of this strategy is to eliminate from the tree those nodes which ultimately lead to undesirable terminal states without even evaluating them to the full. In a nutshell, the algorithm avoids to investigate particular state any further when finds at least one subsequent move that is worse than previously checked candidate plays. It does not violate *best response* definition and provided outcome is exactly the same as the one given by *minimax algorithm*. It significantly reduce the size of the tree but the fact remains that the algorithm has to dig until reaches terminal state - which makes him still not good

[4]For chess, where average branching factor is 35 and game goes to 50 moves by each player search space reach the number of 10^{40} nodes. [23]

enough (the tree is still too big).

- **Transposition table.** In some games, e.g. chess, different sequences of moves lead to the exactly the same position. The target of this technique is to identify "sibling" nodes and copy the equity from other of the same kind which was previously evaluated. It can be achieved by implementing hash table for previously seen positions. Again, since the number of investigated unique position grows very quickly there are additional techniques for selecting which positions are worth keeping in the table, and which should be discarded, mostly based on heuristic evaluation of those positions. This significantly decreases the size of the game tree.

- **Heuristic evaluation and quiescence search.** Strategies mentioned above significantly decrease the size of the search space, however computer programs are expected to play in a timely fashion, i.e. up to 3 minutes per move. This is physically impossible to achieve with current size of the tree despite the fastest contemporary computers. The aim is to search faster hence further improvements have to be implemented. Over the years human players developed array of methods to evaluate the strength of positions and based on them make as accurate as possible guesses whether they favor one player, or lead to a draw. Due to oversize tree C. Shannon[5] coined the notion of cutting off the search in the early stage and applying heuristic evaluation function in order to estimate expected utility of the game for given non-terminal states. The drawback of this proposal is introduction of uncertainty to the process, because of the approximate nature of evaluation function. In the worst case scenario (when inferior heuristic in place) it may mislead the program toward lost positions. Following enumeration presents possible implementation of heuristic in alpha-beta algorithm which proved to be successful.

  - **Cutting off search.** Altered alpha-beta algorithm where utility function is replaced by a heuristic evaluation function ($u(h)$), and the terminal test (**if** $h \in Z$) by a cutoff test which decides when to apply evaluation function (allows to expand the tree until certain level is reached, then stops expansion and applies heuristic evaluation).
  - **Quiescence search.** Apart from inaccuracy of heuristic evaluation mentioned above, that method has another flaw that needs to be addressed – cutoff test. This method calls for extra attention because it is responsible for selecting which nodes are going to be expanded and to which apply evaluation method. Experiments have shown that positions which enable to make moves that in a

[5]For original paper from 1950 see [25].

short run can drastically change the value of estimated utility are not well suited for heuristic evaluations and are very likely misguide the algorithm. The remedy is to apply evaluation only to those states which are quiescent. Therefore non-quiescent positions can be expanded further until quiescent positions are reached. Heuristic knowledge can help to tune the cutoff test method in terms of which move of those that cause *drastic changes in values* we take into account and which we discard.

- **Beam search (also called forward pruning).** This method is inspired by human perception and deduction – who are able to quickly distinguish few candidate moves as reasonable and discard remaining as not promising. The same strategy can be applied in building game tree policy, by using heuristic knowledge. Building game tree policy is enhanced by heuristic evaluation function which designates n best moves for further exploration and discards all remaining marked with lowest utility. This approach is very risky, since there is no guarantee that the best moves are always within selected candidate plays.
- **Probabilistic cut.** It is enhanced alpha-beta algorithm with forward pruning where unpromising states are pruned based on statistical data gained from prior experiences (games from the past are used to evaluate potential utility, and when it is outside $[\alpha, \beta]$ range, then are pruned).
- **Table lookup.** Tables with expert knowledge improves speed of search in positions which frequently occur and human players have a good grasp of their theoretical strength and developed array of techniques. The best example is opening books used at the early stage of the game, where multitude of credible candidate moves constitutes an overkill for most of the programs. Lookup tables are also used to solve end game positions.
- **Metareasoning.** It is reasoning about reasoning, which candidate implementation is drawing a probability of success of certain position based on heuristic evaluation of them. Next, exploitation of the tree is conducted in accordance to probability values of investigated ply.

2.3 Heuristics

The broad term of heuristic evaluation functions is reserved for any sorts of methods that lead to approximate a pay-off of a given position or move by static evaluation of its features, not by inspecting states that are derived from it. Those functions can be categorized into 2 separated groups, as proposed

by [9]:

- **Action heuristics**: provides recommended actions from a given state.
- **Static heuristics**: provides an assessment of the state themselves.

Different heuristics serve for different purposes and thus the discrete value returned allows us for multiple interpretation. For instance, one evaluation strategy can aim to approximate the game-theoretical utility value of given state whereas the other one can assess the quality of certain position. However, the latter value has only meaning when compared with evaluation results for other states. Their aim to to approximate as accurate as it can be, although in a timely fashion.

Most of the invented heuristics makes use of a pattern which is also advocated in [9]. At first the game is abstracted to its core aspects, hereafter called features. Next each of them is calculated in certain manner, and then summed up altogether with different weights in order to yield a final value. In general case Clune's work distinguishes 5 typical features, which are: pay-off, relative mobility, proximity of termination, relative stability of pay-off and relative stability of mobility. Moreover, he proposes in [10] and [9] a framework that can automate the process of constructing a successful heuristics from the description of the game. Proposed framework is particularly designed to serve simplified games (however it conveys the notion of what it takes to construct a good heuristic on your own) and focuses on evaluating a state. Action evaluations are more domain-dependent, however the notion of breaking the position into core aspects is still present.

Heuristics are mostly used to parametrize the search process or evaluate non-terminal states as if they were terminal ones which aims to enhance the performance, mostly in a timely fashion by introducing some compromise in terms of precision (due to approximation nature of heuristic evaluation functions). Heuristics have been largely exercised and when properly utilized result in significant improvement of performance. The most successful story is Deep-Blue which managed to beat running world champion in chess in a six-game match in 1997 [19], however erroneous heuristic evaluation can actually hurt the performance (as it happened so many times with go playing programs).

2.4 Monte-Carlo Tree Search

Alternative approach of solving game problems that can be represented as trees of sequential decisions is family of algorithms known as Monte Carlo Tree Search. Their aim is to find an optimal decision in any given domain by random samples in it and persistently building a partial game tree guided by

the results of previous exploration of the tree. The tree is used to estimate the game-theoretic value of moves [6] and in a longer run should converges to results obtained by *Minimax* algorithm.

2.4.1 Algorithm

This algorithm employs iterative approach of building game tree until some computational budget is reached, e.g. number of iterations or time scheduler per move. Each iteration is broken into 4 steps:

1. **Selection.** The goal of this step is to select the most urgent expandable node (the one which has at least one action that was not tried before) in the currently developed tree. If the inspected node (the procedure is initialized at the root) is not fully expanded (untried $a \in \chi(root) \neq \emptyset$), it selects this node and proceeds to the next step. However, if inspected node is indeed fully expanded, it has to select one child-node based on implemented strategy (BestChild method), and repeat entire procedure starting from the newly selected node. At this point there is a room for debate, which factors should influence the selection process since it has to handle trade-off between exploration and exploitation. On one hand, it should be targeting at the most promising nodes, although on the other hand, due to probabilistic nature of this algorithm (evaluation is based on random sampling) it needs to take into consideration also less promising nodes, in case of unfortunate evaluation – especially when nodes are not highly explored. There are few techniques developed recently, e.g. UCT algorithm, which we describe in the next part of this chapter. Selected node is handed to the procedure responsible for the next step.

2. **Expansion.** This step results in creation of a new node derived from the one obtained in *Selection* step. From delivered node h_0 one untried action $a \in \chi(h_0)$ is selected at uniformly random and a new node is created based on it $h_1 = \sigma(h_0, a)$. Then the new node h_1 is appended to the tree as a child of h_0.

3. **Simulation.** From the moment new node is reached the game (in this iteration) progress randomly until it reaches its terminal state – at each turn the agents alternately select a random action from array of all candidate plays. There was a notion of using heuristic knowledge in order to favor more promising actions [8] and we will return to this idea in the next chapters.

4. **Back-propagation.** Intention of the closing stage is to update statistics of all visited nodes in this iteration (the sequence of nodes selected

in this iteration) on the result of proceeded simulation so as the next time the tree is searched future decisions are predicated upon the most recent statistics. This makes evaluation more accurate as number of simulations increases (similar to mathematical Monte Carlo method)[6].

Figure below sketches organizational schema of MCTS.

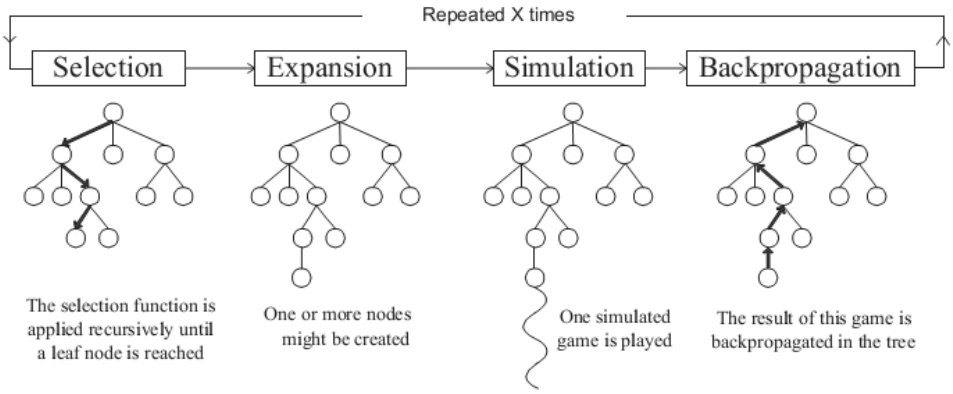

Figure 2.1: Outline of a Monte Carlo Tree Search.[8]

From the programming standpoint it is convenient to distinguish only 2 policies:

1. TreePolicy - joins *Selection* and *Simulation* steps.

2. DefaultPolicy - self-plays random game until terminal state is reached (*Simulation*)

All the mentioned steps are summarized in Algorithm 2.2, which constitutes a backbone for further analysis. In order to make is easier to comprehend we introduce the notion of a state, which is a formal representation of conducted moves upon to this point (the analogy from board games would be a board with settlement of all pieces remaining in the game). h_0 corresponds to the state s_0 and Δ is a reward for the terminal state reached by running DefaultPolicy.

[6]In terms of UCT algorithm (section 3.2) Koscic and Szepesvári has actually proved it [15].

Algorithm 2.2 General MCTS algorithm

```
1: function MctsSearch(state s_0) return action
2:     create root node h_0 with state s_0
3:     while within computational budget do
4:         h_new ← TreePolicy(h_0)
5:         Δ ← DefaultPolicy(state(h_new))
6:         BackUp(h_new, Δ)
7:     end while
8:     return aciton(BestChild(h_0))
9: end function
```

2.4.2 Upper Confidence Bounds for Tree (UCT)

Collection of Monte Carlo algorithms in which actions within TreePolicy are selected at random despite gathered statistics is called *flat Monte Carlo* and has demonstrated to perform rather poorly, like indicated by Browne [5]. Having said that the next target is to improve the selection by utilizing statistics attached to each node. Few primitive approaches were initially proposed, however non of them won appreciation for during tests. Based on Cheslot et al. work [7] basic criteria that need to be taken into account are derived:

1. *Max Child*: select root child-node with the highest reward so far.
2. *Robust Child*: select the most visited root child-node.
3. *Max Robust Child*: select root child-node with both, the highest reward and visit count [12].
4. *Secure child*: select the child which maximizes a lower confidence bound.

This short introduction leads us to well known *bandit problems* domain[7] where agent needs to make a choice between n candidate actions in order to maximize cumulative reward. Due to unknown distribution of reward the problem becomes *exploitation-exploration dilemma* – agent needs to make a trade-off between exploitation[8] of the action deemed to be optimal (based on currently collected statistics), and exploration of other actions that appear to be sub-optimal at this stage of the development of the tree. A popular measure of policy's success in addressing this dilemma is the regret, that is the loss due to the fact that the globally optimal policy is not followed all the times[2]. Lai and Robbins [16] as a first proved that the regret in context of this dilemma

[7] For comprehensive introduction see [1]

[8] By exploitation we mean conducting self-played random games that feed the statistics of all nodes selected in the iteration step.

has to grow at least logarithmically as the number of conducted plays increase. The regret after n plays is defined as

$$R_N = \mu^* n - \mu_j \sum_{j=1}^{K} \mathbb{E}[T_j(n)] \tag{2.1}$$

where μ^* is the best possible expected reward and $\mathbb{E}[T_j(n)]$ denotes the expected number of plays for arm j in the first n trials.

Koscic and Szepesvári [15] achieved remarkable results of their Monte Carlo Tree Search implementation in the game of Go due to improved TreePolicy. They advocate to use *Upper Confidence Bound for Tree* (UCT) algorithm which tackles mentioned earlier exploitation-exploration trade-off. Every time the choice between which candidate play select is to be made is inspected as an independent multi-armed bandit problem. A child-node is selected as

$$\arg \max_{h' \in \sigma(h,\chi(h))} \frac{Q(h')}{N(h')} + c\sqrt{\frac{2 \ln N(h)}{N(h')}} \tag{2.2}$$

$Q(h)$ is a quality of node h (e.g. number of won simulations when this node was selected), $N(h)$ is a number of total visit of node h, and c is a constant which aims to balance how big influence is given to second segment of the formula. As it can be easily observed, first part of the formula, $\frac{Q(h')}{N(h')}$ favor exploitation whereas the second one, $c\sqrt{\frac{2lnN(h)}{N(h')}}$ exploitation. Speculation with the value of c can be used to experiment whether deeper exploitation or broader exploration yields more accurate approximation and improves quality of play. If more than one child-node has the same maximal value, the tie is broken randomly.

In [15] authors have experimentally found that in its raw version algorithm yields the best results with constant $c = 1/\sqrt{2}$, however they deem to tune the value for each algorithm that differ from the Algorithm 2.3 [6]. In order to facilitate new selection approach a new method – BestChild – is incorporated to the TreePolicy, and in UctSearch with smoother $c = 0$ (at this point we focus on selecting the best child without making any compromise).

Algorithm 2.3 The UCT algorithm

```
1: function UctSearch(state s_0) return action
2:     create root node h_0 with state s_0
3:     while within computational budget do
4:         h_new ← TreePolicy(h_0)
5:         Δ ← DefaultPolicy(state(h_new))
6:         BackUp(h_new, Δ)
7:     end while
8:     return aciton(BestChild(h_0, 0))
9: end function
```

Algorithm 2.3 The UCT algorithm (continued)

```
10: function TreePolicy(node h) return node
11:     while h is non-terminal do
12:         if h not fully expanded then
13:             return Expand(h)
14:         else
15:             h ← BestChild(h)
16:         end if
17:     end while
18:     return h
19: end function

20: function Expand(node h) return node
21:     choose a ∈ untried χ(h)
22:     add a new child v' to v where h' = σ(h, a)
23:     return v'
24: end function
25: function BestChild(node c, double c) return node
26:     return arg max_{h'∈σ(h,χ(h))} Q(h')/N(h') + c√(2 ln N(h)/N(h'))
27: end function

28: function DefaultPolicy(state s) return double
29:     while s is non-terminal do
30:         choose a ∈ A(s) uniformly at random
31:         s ← f(s, a)
32:     end while
33:     return reward for state s
34: end function

35: function BackUp(h, Δ)
36:     while h is not null do
37:         N(h) ← N(h) + 1
38:         Q(h) ← Q(h) + Δ(h, p)
39:         h ← parent of h
40:     end while
41: end function
```

Koscic and Szepesvári [15] proved that the likelihood of selecting sub-optimal action at the root of the tree converges to zero at polynomial rate, which implies that given enough time UCT approximation converge to the minimax tree and is thus optimal.

Few features that distinguish MCTS among all previously used approaches need to be stressed.

1. **Aheuristic**: the only domain-knowledge feature is distinguishing whether particular state is terminal or not, and evaluating terminal state. This implies that MCTS can be used to any environment where decision chain can be modeled as a tree in a sense of graph theory.
2. **Timing**: the execution of the algorithm can be terminated at any time and it allows to return the result which is based on fair distribution of exploration of the tree in accordance to the implemented policy (having given time) where *minimax* approach fail due to exhaustive evaluation of each unique state at certain ply.
3. **Asymmetric tree**: since the development of the partial tree is directed toward the most promising nodes it ultimately leads to asymmetric tree. Figure 7.1 exemplifies this issue.

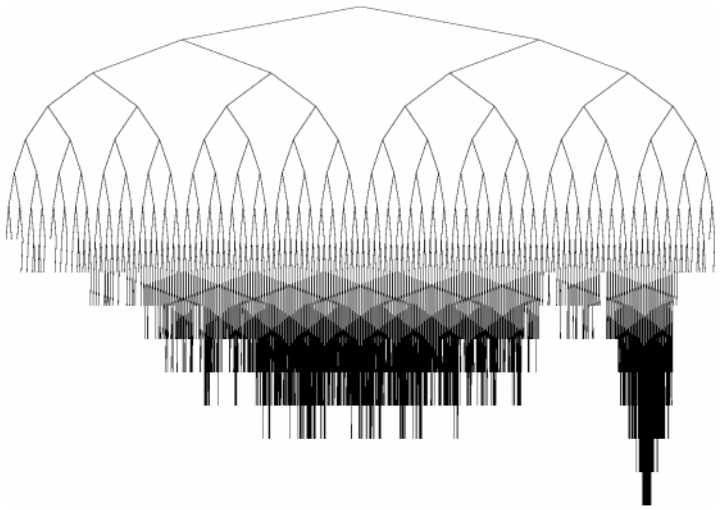

Figure 2.2: Asymmetric tree growth presented in [11] by using BAST variation of MCTS.

2.4.3 Enhancements

As we have seen in the previous section, classic Monte Carlo Tree Search approach is utterly free of any domain knowledge, therefore it is applicable to any suitable environment. However, this report narrows the scope of the game field to only those games that are equipped with reasonable heuristics and thus can be implemented to the Monte Carlo Tree Search so as to improve the quality of the performance. What heuristics do is assigning district

value for either action or state in order to denote its strength (or level of strength). Next this value can be used to adjust evaluation proposed either as a approximation of game-theoretic value or somewhere down the process of implemented procedures. The following subsections describe how utilization of heuristic evaluation contributed to MCTS.

Tree Policy Enhancements

Tree policy is central to MCTS because it directs which parts of the tree need to be explored the most. Heavy exploitation based on false promises may lead to unintentional underestimation of optimal plays.

1. **UCB1-Tuned**: however it is not driven by any heuristic it is vital to stress enhancement suggested by Auer et al. [2] where they suggest to replace the upper confidence bound

$$\sqrt{\frac{2\ln n}{n_j}} \tag{2.3}$$

with

$$\sqrt{\frac{\ln n}{n_j}\min\{\frac{1}{4},\, V_j(n_j)\}} \tag{2.4}$$

where

$$V_j(s) = \frac{1}{s}\sum_{r=1}^{s} X_{j,r}^2 - \bar{X}_{j,s}^2 + \sqrt{\frac{2\ln t}{s}} \tag{2.5}$$

It happened to improve the performance in several occasions, e.g. [14].

2. **EXP3** *Exploration-Exploitation with Exponential weights* : algorithm originally proposed by Auer et al. [3] improves the policy by computing the estimated gain for each arm based on probability distribution and updates the cumulative gain. Then new distribution of probability can be computed over the candidate plays for certain node.

3. **Transpositions**: the notion is exactly the same as for *minimax* algorithm as presented in Section 2.2, transition from tree into directed acyclic graph.

4. **Progressive bias**: enhancement proposed by Chaslot et al. [7] which employs heuristic evaluation of a state when a node has been visited only a few times and its statistics are not reliable. A new component is added to the formula (2.2),

$$f(n_i) = \frac{H_i}{n_i + 1} \tag{2.6}$$

where the node with index i has been visited n_i times. As the number of visits increases the function approaches zero. In the same paper the authors propose another solution, to apply heuristic evaluation only until fixed number of visits was reached[9].

5. **Opening books**: exactly the same approach as proposed in Section 2.2.

6. **Progressive unpruning/widening**: coined by Chaslot et al. [7] aims to reduce the size of the tree by investigating favored by heuristic evaluation actions at first and then gradually widening the scope of candidate plays. The advantage of this algorithm is that all moves will be considered (given enough time). This approach was heavily tested and was not always a success story[6]. This is the example of *soft pruning*.

7. **Pruning with domain knowledge**: aim to prune actions known to lead to a weaker positions based on their heuristic evaluation [6]. Employed in programs playing go showed slight improvement of the results.

Surprisingly, according to the [6] there was no attempt made to improve the expansion stage.

Simulation Enhancements

There is a lot of criticism regarding self-played simulation due to their random nature. In most cases they are unrealistic in comparison to the games played by rational players and their big sample can only reveal statistical tendency (which tend to be an approximation of game-theoretic value). There were few documented attempts to tackle this issue.

1. **MHSP** (*Mean-based Heuristic Search for Anytime Planning*): as proposed in [21] this algorithm differ from classic MCTS in two ways.

   - Random simulation is entirely replaced by heuristic evaluation.
   - Selection process uses only average reward and completely dismisses the notion of exploration.

2. **Using History Heuristics**: this approach based on the assumption that move which is good in one particular situation might be at least considered as a credible candidate play in different position. [6] indicates attempts made which improved the overall quality of play.

[9] This was mostly driven by timing factor – most of the evaluation function are highly time consuming.

3. **Evaluation Function**: Winands and Björnsson in [28] tested few policies and came up with conclusion that the most successful strategy is when at initial stage evaluation function is only used to avoid bad plays, but when the simulation progress further the agent should gradually more and more rely on heuristic evaluation and greedily selects best moves.

Other Enhancements

Another spot calling for attention is improving back-propagation procedure. They can reflect the notion that simulations conducted at the later stage of the game are more likely to resemble a game played by rational players and thus its outcome should be intensified in statistics.

Other enhancements may be focused on running simulation in parallel, however it is out of scope of this project.

2.5 The Game PahTum

PahTum is one of the oldest games in the world[10]. The board is 7×7 grid with odd number of randomly selected fields which are unavailable to play, so-called *black holes*. The game is played by placing white and black stones, one at a turn, alternately by two players. The stone can be placed only on an unoccupied field which is not the black hole. The game ends when the board is fully populated (or one of the players resigns or they agree on a draw). Traditionally white starts the game.

The goal of the game is to collect more points then the opponent by creating connected lines of stones for which player is awarded with certain amount of points. A connected line must be either vertical or horizontal and may contain only stones of the same colors, no opponent stones or black holes. A player is awarded by a certain amount of points for each connected line of his color – the score function is as follows,

$$\text{score}(1) = \text{score}(2) = 0$$

$$\text{score}(n) = n + 2 \cdot \text{score}(n-1);\ n \geqslant 3.$$

Player who gets more points wins the game, when both are equally awarded the game is a draw. Figures 2.3 and 2.4 exemplify starting position and middle game respectively. As we can see on Figure 2.4 black has created 1

[10] PAH-TUM gameboards have been discovered in Mesopotamia and Assyria and a board fashioned in ivory was discovered in the tomb of Reny-Seneb, XII dynasty, and dated at roughly 1800 BC [13].

vertical line of 3 stones for which he is awarded with 3 points. White has managed to form 2 lines, one horizontal of 3 stones (worth 3 points) and one vertical made up of 4 stones (10 points). Altogether white is leading 13 to 3.

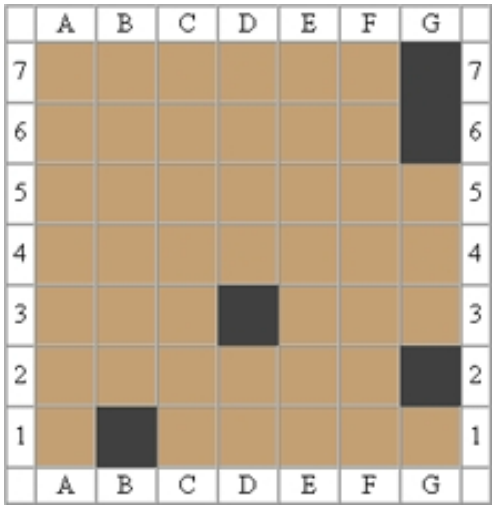

Figure 2.3: Empty board [22].

Figure 2.4: Sample game [22].

Despite the fact that both players will make the same number of moves the game is far from being balanced. Player who makes an initial move has the edge over his opponent, which is often called *tempo*. In the early stage it results in for each white's move black has to come up with a response (i.e. blocking move that limits the development of opponent's structure) or develops its own structure which yields the same or even more points as his opponent's. Otherwise he will lose the market and in fact the entire game. Although various strategies have been developed there is a consensus among players that one who makes initial move is sure to win the game with occasional exceptions for a draw. Those properties depend on the board, and more precisely distribution of *black holes* and their total number – some constellations can guarantee a win for white, some only a draw when perfect strategy is in place[11]. Boards with higher number of *black holes* generally give less flexibility in a sense, that fewer sequences of moves lead to exactly the same position. As a result, occupying spots of the highest strategic value becomes the most urgent task in the early stage of the game. Statistics gathered from the online game server, BrainKing (up to August 24, 2012), are presented in Table 2.1 and supports the claim of first player advantage. It needs to be said that those statistics are made up of games played by people of all sorts of skills.

[11]This is also a general consensus among top players, no study has proved this claim so far.

Statistics of won games	
white	30,648 (54.78 %)
black	17,191 (30.73 %)
draws	8,099 (14.47 %)

Table 2.1: Game statistics [22].

Chapter 3

Development Process

This chapter emphasizes engineering side of the project and elaborates development methodology in place. Meanwhile rationals are presented in order to justify the decisions made.

3.1 Agile Methodology

For purpose of this project, taking all the pros and cons into consideration, we selected agile methodology to facilitate the life-cycle development process. For obvious reasons employed framework was slightly customized so as to make the most of it by adopting it to unique features of the project.

3.1.1 Workload Organization

The entire workload was broken into series of sprints. Sprints were organized on a weekly fashion with two exceptions due to unrelated circumstances to the project. Each sprint aimed to release a testable piece of software[1] which could be analyzed during scrum session at the end of each sprint. The purposes of scrums were to:

1. measure to what extent the last release was successful,
2. evaluate the direction of particular approach[2] and whether it can be improved and how, or do we need to steer it in another direction,

[1] By testable piece of software we mean executables suitable for unit testing which can prove that investigated software does what was intended.

[2] By approach we mean a variation of algorithm.

3. report any problems or obstacles and to address them,
4. set up a milepost for the next sprint (based on all above points).

3.1.2 Rationales

Employed methodology is supposed to improve the quality of work by better time management and minimize the risk of failure. Following reasons support selection of agile methodology:

1. **Research-oriented project**: the project is about inventing different algorithms and the only way to measure how successful they are is by trial and error. We simply develop one algorithm and put it into practice. If it works then we seek for further improvements, if not (and there is no way to fix it) it is reported and discarded – simultaneously it aims to minimize time spent on not prospective approaches. It is expected that certain amount of research has to be conducted during each attempt. This approach fits very well the concept of sprints organized on a weekly basis where each ends with an analyzing session.
2. **Cyclical optimization of algorithms**: it is expected that first release of algorithm would not be the most efficient and would require some additional work. Very often candidate improvements are built on shoulders of past experiences and weekly meetings with experienced researcher are a big advantage.
3. **Credibility of tests**: in order to prove that one approach is more successful than others it needs to be set up in a context of different algorithms that tackle the same problem-space (also using the very similar concepts where various parameters have to be tuned experimentally). Often releases ensure creation of the variety of algorithms between which further comparisons and analyses can be conducted.
4. **Empower the developer**: often re-evaluations (scrum sessions) raise an opportunity to frequent proposing original solutions, sometimes based on intuition which can be quickly proved to be reasonable or not.

3.2 Critical Appraisal

Weekly fashion of sprints is absolutely reasonable for releasing a new piece of software. Reasons for that are as follows:

1. Each release is re-using once developed test environment.

2. Each release relies on the UCT algorithm which serve as a scaffolding for each new release.
3. The problems are very often well elaborated in the literature, so it is expected to find necessary information rather quickly and in condensed form of one or two articles or chapter in a book.
4. Confidence in programming in Java.

Chapter 4

Requirements Analysis

This section is dedicated to formalizing all requirements gathered alongside the development process. They were gradually increasing as the project advanced into more mature levels and new mileposts were set. Stakeholders, as a contributing factor, are analyzed as well.

The requirements are intended to guide the design and implementation of the software as well as build foundation for the tests.

4.1 Stakeholders

Stakeholders who are a party to this project are listed below. Due to their influence on the project requirements they can be categorized as follows.

- **Project supervisor**: a direct stakeholder who set up goals that software has to achieve. They are predicated on results obtained from previous sprint. Key requirements regarding the test environment are unchangeable. However, other might evolve (especially those tied with different variations of agents), mostly on a weekly fashion thanks to scrum sessions.

- **Developer**: a direct stakeholder who is responsible for creating a piece of software and plays an active role during scrum sessions when new goals for upcoming sprint are established.

- **Research community**: an indirect stakeholder who can freely re-use created software for any purpose under the terms of the GNU General Public License.

4.2 Requirements Specification

This subsection features requirements categorized as functional and non-functional. The functional requirements are those areas where the software needs to provide useful functionality. The non-functional requirements are the qualities that the software needs to have and constraints on the functional requirements that the system needs to adhere to.

In order to keep the list as short as possible we need to introduce a glossary:
Single game: one game proceeded from starting position (empty board) until terminal state is reached (fully populated board).
Match: sequence of single games where 2 agents switch sides after every finished single game. For testing purposes certain technique of switching boards was invented, largely elaborated in the Design section.
Experiment: collection of matches that run one after another until they all terminated (agents are assigned to the matches independently, so as the board collections that are used).

4.2.1 Functional Requirements

Requirement ID: FR001
Stakeholder: Project supervisor
Requirement: The environment will enable 2 AI agents to compete with each other in a match.
Rationale: Because of probabilistic nature of investigated algorithms their strengths have to be determined in a match constituted by multiple single games.
Fit criterion: The user is informed when the match started and right after the match terminated.

Requirement ID: FR002
Stakeholder: Project supervisor
Requirement: The user will be able to set the parameters of the match, i.e. collection of boards used in the match, number of single games played, name of the output file with statistical data.
Rationale: Boards used in the experiment can have different number of black holes and it is suspected that they need a slightly different strategy in play. Therefore testing the same agents on various fields may yield interesting outcomes.
Fit criterion: The user can select a file with boards to be used in the match, determine number of single games in the match, and type a name of the output file.

Requirement ID: FR003

Stakeholder: Developer
Requirement: The user will be able to design experiment compounded of few matches.
Rationale: Since one match can take up to few hours it is desirable to schedule few matches which altogether take about 24 hours, so the results can be collect in the next day.
Fit criterion: The user can add extra match to the experiment.

Requirement ID: FR004
Stakeholder: Developer
Requirement: After at match is over (as party of the experiment) the environment will provide a statistical results of it.
Rationale: Since we are not able to investigate every single game conducted we have to rely on statistical data about matches that were carried out.
Fit criterion: The user can browse the outcome file.

Requirement ID: FR005
Stakeholder: Developer
Requirement: The user can create a file with a given name that contains fixed number of boards of specified kind which can be reused in the experiments.
Rationale: It is necessary to conduct all tests on the same collection of boards, because randomization of black holes at each game can distort the results and ultimately undermine confidence in the results.
Fit criterion: The new *.sav file with specified boards is created.

Requirement ID: FR006
Stakeholder: Project supervisor
Requirement: The system will be able to evaluate each candidate move based on heuristic utility function and assign discrete value to it.
Rationale: This project investigates to what extent heuristics can be applied to the Monte Carlo Tree Search algorithms and whether they enhance or hurt the quality of performance.
Fit criterion: For given state the system can generate list of candidate moves where each is accompanied by the heuristic utility value.

Requirement ID: FR007
Stakeholder: Project supervisor
Requirement: The user can tune agents' parameters within the scope of one match.
Rationale: In order to conduct exhaustive tests it is crucial to check the behavior of agent under various conditions.
Fit criterion: The user can specify number of roll-outs per move and constant implemented in the BestChild method.

Requirement ID: FR008

Stakeholder: Project supervisor
Requirement: The system will be able to provide a list of all valid moves for any given position.
Rationale: Agents make analyses often based on complete information of all candidate moves.
Fit criterion: For given position the system generates list of all valid moves.

4.2.2 Non-functional requirements

Requirement ID: NFR001
Stakeholder: Project supervisor
Requirement: The system should quickly provide heuristic evaluation of every candidate play for given position.
Rationale: It is expected that agents will highly rely on heuristic evaluation, thus it will be frequently used. Having said that, it is not desired that those additional calculations significantly extend time needed to perform the same number of roll-out as it took place in agent which did not employ heuristic evaluation.
Fit criterion: Time of performing all roll-outs does not double when heuristic evaluation in place, in comparison to agent with similar architecture, only without heuristic.

Requirement ID: NFR002
Stakeholder: Project supervisor
Requirement: The evaluation of candidate moves is fairly accurate.
Rationale: Heuristic evaluation predicates further decisions in the process of building tree and approximating game-theoretic values. Therefore, the more accurate it is, the more promising our results should be.
Fit criterion: The agent employing only heuristic evaluation can easily outperform primitive AI or novice player.

Requirement ID: NFR003
Stakeholder: Developer
Requirement: The environment should provide live recording during the run of the experiment.
Rationale: The experiments may take up to few days (depends on the settings) and there is a possibility that some background process will cause shutting down of the system (e.i. system update), or power supply will fail and it would be a huge waste to lose all collected data in that way.
Fit criterion: After every single game is finished, the outcome of it is recorded in the dedicated file. Then when the unexpected event will occur we will only lose the outcome of a single game during which it happened.

Requirement ID: NFR004
Stakeholder: Project supervisor.

Requirement: All agents have to utilize the concept of Monte Carlo Tree Search in their decision process.
Rationale: This is the centerpiece of this project, to determine whether using heuristic evaluation enhance the performance of Monte Carlo Tree Search, or not.
Fit criterion: Every MCTS-alike agent during the decision process is building a partial game tree, employs some selection process and use at some point random self-played simulations to determine the utility value of a node.

Requirement ID: NFR005
Stakeholder: Project supervisor and developer.
Requirement: The implementation should adhere to Java coding convention and be easy to reuse for others.
Rationale: Those are basic principles and ethics of coding.
Fit criterion: Developed methods are limited to small number. Each of them performs one key task (i.e. getAllValidMoves), all are accompanied by the commentary.

Chapter 5

Design and Implementation

This chapter documents the design and implementation of the test bed and invented agents. All decisions made are discussed so as to make the reader aware of the concepts that inspired them and their justification. The material is organized in a conceptual manner, which means that similar approaches are grouped together in order to present them in a compact form and make them easy to follow. Mapping between implemented features and functional requirements is provided in order to demonstrate that requirements have been fully met. Satisfaction of non-functional requirements is presented in Section 6.8.

At the beginning we provides reasons for game selection. Then we elaborate on a test bed that facilitates this project alongside an agent that exercises UCT algorithm discussed in Chapter 2. Next we present the construction of heuristic and the first AI agent that solely relies on proposed heuristic evaluation without performing any roll-outs and easily outperforms UCT. Remaining sections are guided by analyses of the UCT algorithm and demonstrate how various enhancements utilizing proposed heuristic evaluation are incorporated into UCT algorithm which serves as a scaffolding for all further implementations.

5.1 Reasons for PahTum

For purpose of this project we decided to implement the ancient game of PahTum introduced in section 2.5, which was our free choice. Following reasons led us to this decision (in order to emphasize them we outline the comparison with Chess, however it could be done with any other game as well).

1. **Feasible heuristic evaluation**: Preliminary idea of this project was to investigate whether incorporating good heuristics[1] into MCTS would improve or hurt the quality of play. Therefore we were looking for a game that already had developed good heuristics or one where we could easily write heuristic on our own. Since the inception of this project we knew what factors influence the decision process while playing PahTum, how to evaluate them and combine together thanks of our experience in playing this game[2]. Games like Chess or Othello are already equipped with good heuristics whereas PahTum has not been studied before.

2. **State representation**: Since MCTS performs better when number of roll-outs increase[8] it was expected to evaluate multitude of states. In PahTum state is represented by the grid 7×7 where each field can be either a *black hole*, empty or occupied by the stone of one of two kinds. Once the stone is placed it remains there until the end of the game. Final score is calculated based on settlement of the stones where each of them features in exactly the same way. The simplicity of it ensures quick evaluation of the position and economizes the memory. On the other hand the process of evaluating a state in Chess is much more time consuming due to much higher complexity level (special moves like castling or en passant can be performed with regard to the history of previous moves).

3. **Simplicity of moves**: MCTS employs self-played random simulations which implies that before each turn the board has to provide a collection of all valid moves from which one can be chosen at random. PahTum suits it perfectly – each empty field is a candidate play. Therefore the task can be accomplished simply by checking the status of each of the 49 fields. This gives a huge computational edge in contrary to Chess, where not only pieces move differently but also their moves cannot put their own king in check.

4. **Terminal state test**: In PahTum the state is terminal when there is no empty field left. However in order to check whether the particular state is terminal or not we do not need to check all 49 fields. Instead, we can run a move count which would be incremented after each performed move. When the counter equalizes the $49 - number\ of\ black\ holes$ the terminal state is reached. It can significantly speed up evaluation in comparison to Chess where such technique cannot be implemented (the king has to be in check and has no way to escape to safety (checkmate), or not be in check and any piece cannot have a valid move to play (stalemate, also called pat)).

[1] By the term *good heuristic* we mean heuristic that proved to be successful in different algorithms, e.g. *alpha-beta*.

[2] In the past we won few international tournaments in on-line competition.

5.2 Implementation of the Test Bed

Before we get started there was one more decision to make – selecting the programming language. The above section advocates so strongly the need of quick evaluation so we decided to make the speed our main concern. Among candidate environments, which were Python and Java, at the end of the day we decided to go with Java. The main reason was expected run time. Python programs tend to be slower[26], despite using PyPy compiler[3]. In favor of Python the development generally takes less time than in Java.

5.2.1 Architecture Overview

The basic notion of the test bed is presented on Figure 5.1 which depicts 5 packages and associations between classes within them. Each developed agent was organized in a new package, however, they all interact with the application in the same manner. Thus only one is presented to avoid unnecessary complexity. The associations are intended to indicate the notion that governed their design and how they complement themselves.

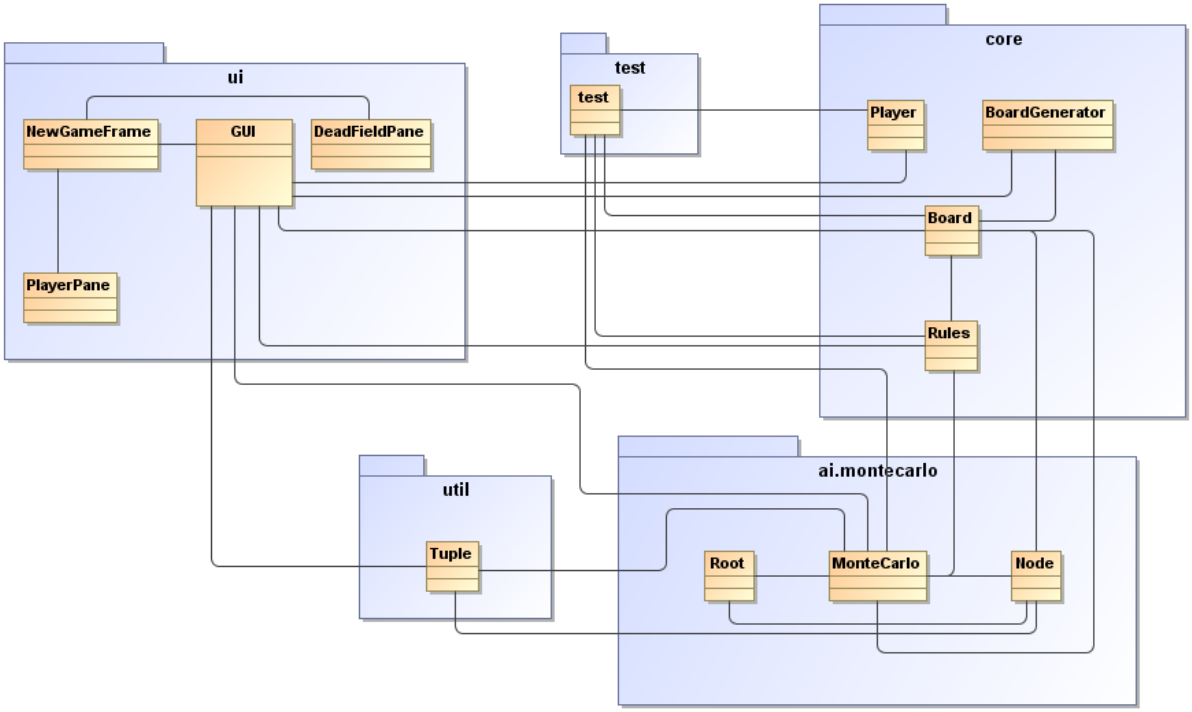

Figure 5.1: Class diagram of the test bed application.

[3]We have conducted implementation of chess in Python and share the lesson learned in Chapter 7.

Let us briefly explain the concept of each component.

- **Core package**: Provides means of conducting the game. Each class included facilitate different aspect:
  - **Board**: Represents the state of the game as a board with distribution of *black holes* and stones accordingly to previously performed moves. Additionally, provides cooperative classes with collections of valid moves and moves accompanied by their potential (calculated by heuristic evaluation function) (satisfies requirements FR006 and FR008)
  - **BoardGenerator**: Generates collection of boards of the same type, each with random distribution of *black holes* for testing purpose (in conjunction with Board class they satisfy requirement FR005).
  - **Player**: Organizes information about the agent regarding his type, color of stones that he plays and so forth.
  - **Rules**: Serves as a referee who calculates the final score.
- **Util package**: Contains only one class, **Tuple** and it aims to help other classes to accomplish their tasks by eliminating boilerplate code. It conveys exactly the same notion as mathematical ordered pair. The concept is taken from Python, with exactly the same outcome. For instance it facilitates the communication, when two coordinates are passed on from one object to another.
- **UI package**: Provides graphical representation of the game. It was mostly used to inspect whether agents were playing on a reasonable level at the initial state of development and in which direction parameters should be tuned. It is not used to any other purpose but we decided to keep it in case of potential further work on this game. **GUI** class represents the main frame with the board whereas other tree classes (**NewGameFrame**, **PlayerPane**, **DeadFiledPane**) facilitate the setting of the game and agents parameters. They also enable human players and AIs to compete at any configuration (satisfies requirement FR002).
- **Test**: Contains one class that facilitate the process of conducting experiments which is explained in detail in section 5.2.3 (satisfies requirements FR001, FR003, and FR004).
- **MonteCarlo package**: Provides implementation of MCTS algorithm by means of tree classes:
  - **Root**: Represents the root of the game tree by providing a handler to the first node.

- **Node**: Conveys the notion of a node in a game tree that represents choice of one of the players. In addition it collects needed statistics.
- **MonteCarlo**: Organizes entire process of creation the partial game tree adhering to the UCT algorithm and its various modifications (satisfies requirement FR007).

5.2.2 Monte Carlo Tree Search Scaffolding

The key features of the MCTS algorithm are *Selection* and *Simulation* policies which have a fixed place in the procedure. However, how they accomplish their tasks is the subject of this thesis. It is expected that some candidate solutions, although looking promising from the theoretical standpoint, will yield unsatisfying results in tests. Therefore the implementation phase should be conducted quickly (during one sprint) in order to reject unsuccessful approaches with the least effort made.

Those considerations led us to adopt the UCT algorithm presented in section 2.4.2 as a scaffolding for implementations of all further agents (except from one, solely heuristic-based). We decided that communication between objects, parameters and returning types must stay unchanged throughout the entire development process so as all new agents share similarities with the plain UCT algorithm in terms of the core structure.

When this approach is practiced any proposed enhancements take place only in the body of targeted methods. This enables to inspect how various enhancements combined together work when applied to different policies. Moreover, one of them can be changed without modifying the others in order to tune the values of constants used in various calculations so as to obtain the highest quality performance.

Despite the very neat structure of the UCT algorithm we applied few changes from the implementation point of view in order to make our rendition execute faster and consume less memory space.

Overwhelming part of the literature advocates to represent state as a node in the game tree structure. We regard this approach as a memory greedy policy and associate only a move to the choice-node object. Implication of this is slight modification of the process of creating a game tree. Before the algorithm starts a new roll-out it makes a deep-copy[4] of a board associated to the root node (initial state). Then when it traverses the game tree down to

[4]In the same sense as it takes place in Python, where original and cloned objects share no references. In Java implementation we obtained the same effect by capturing the object written to a stream in a byte array by making use of ObjectOutputStream and ByteArrayOutputStream classes to copy an object and ObjectInputStream and ByteArrayInputStream classes in order to restore it.

the leaf (during *Selection policy*) it performs all the moves associated to the visited nodes on this board (nodes also keep the information regarding the color of player who makes the move). From the moment it reaches a leaf node the *Simulation policy* takes place and proceeds a random self-played game on the same board until no empty field is left.

Making a copy of the board is unavoidable – the only difference in comparison to classic representation of the game tree (nodes hold explicitly state representations) is the moment when it would take place – after reaching the leaf node.

Another improvement that speeds up the execution is redefining the terminal test (check point which determines whether the state is terminal or not). We took the advantage of the game domain – every game finishes when the board is fully covered by stones and we know the number of the fields in the initial state. Therefore we run a move count which is incremented every time we visit a node and for each turn during the *Simulation policy*. This technique eliminates inspecting every board thoroughly after each move – which becomes essential time saver in case of simulations.

In Java when an object is passed as a parameter of a method it does not create a new object but instead holds a reference to an old one. This feature is very useful when the board object is shared between several different objects and no extra coding work needs to be done.

5.2.3 Experiment Organization

In order to compare two agents and assess their strengths of play we had to organize a duel between them. The game is biased and favors a player who makes the first move (hereafter called player #1, his opponent is called player #2). We decided to remove player advantage by organizing games in a set. Set requires 2 games be played with players switching sides in the second game where both games are conducted on the same board. We realize that one set is not sufficient enough to provide results that can empower us to formulate a credible conclusions. Discussion regarding number of games is delivered in subsequent section.

Another factor that can have an effect on the experiment outcome are boards, and being more specific random selection of *black holes*. We suspect, and the majority of top players share this opinion, that some boards secure a player #1's win whereas others only a draw (when both players implement a perfect strategy). In order to minimize this phenomenon we decided to organize our experiment in a way that each set is played on a different board, however all boards used in one experiment share the same number of black holes (they differ in their distribution, which is random).

In order to conduct a comprehensive analysis outcome of every game is recorded (who played at each side, who won or whether draw occurred) and at the end they are summarized. This allows not only to investigate whether the agents resemble the tendency observed between human players but enable to spot bizarre occurrences, i.e. certain agent always loses on one particular board. Our implementation provides a text file with all recordings and summaries.

5.3 Construction of Heuristic

During the game player's primary aim is to create stone structures for which he will be awarded with points. Additionally he can adopt a destructive strategy where he tries to block his opponent's structures from further development. The top players are able to mix those two strategies together. There is no theory behind and balancing them is up to the player, however, in some cases current score can slightly favor one approach or another. Because of random distribution of black holes and the development of the game which can go either way, creation of general rules of how to play is not feasible.

In this thesis we decided to create an action heuristic, although constructing a static heuristic is perfectly feasible. Few factors decided on our choice.

- **Threat notion**: In some positions there is only one good move and selecting a sub-optimal one can immediately squander the entire game. This is very similar to the notion of threat in chess. Research papers were constantly emphasizing this aspect as a weakness of MCTS algorithm and it was exemplified number of times in chess implementations. We thought that biased selection policy utilizing action heuristic can address this issue, for instance by incorporating a beam search notion. Similar effect is not easy to accomplish when static evaluation is in place.
- **Easy identification of weak moves**: It is very easy to identify weak moves in PahTum, especially in the early stage of the game where the number of candidate moves is the highest. In a sense it can work as an anti-heuristic and suggest which moves should not be considered as promising candidate plays. Because of high branching factor in the early stage of the game this technique can significantly deeper exploitation and in the same time does not miss any good move.
- **Lack of interest in action heuristics**: In most publications authors focus on static evaluation of states and action heuristics seems to be forgotten. It is not a mystery that the notion of static evaluation is easier to comprehend and visualize the concept of the game. Both

techniques share some similarities however the way how they can be applied differ which is another reason why we wanted to exercise them.

Our action heuristic evaluation aims to assess the potential of each field by assigning one discrete value to each of them. Their distribution across all empty fields (candidate plays) should reflect by how much one play is better than the other. The difference between two relatively close plays should be slight whereas between top play and ludicrous one should be significant. The next section explains how we accomplished this.

5.3.1 Algorithm

Our heuristic function evaluates each empty field independently by assessing their potential. To do so we decompose the game strategy to its core aspects. We distinguished 3 contributing factors[5] and present how to calculate their impact as follows.

1. **Open lines**: We inspect horizontal and vertical lines in all 4 directions starting from our field of interest and move toward the edge of the board, unless we encounter opponent's stone or *black hole* as shown on Figure 5.2. We increase the value of potential for all fields marked as *empty* or occupied by our stone as presented in Algorithm 5.1. This piece of code has to be executed 4 times, twice for vertical line, twice for horizontal (here in line 15 vertical_adjustment would be changed into horizontal_adjustment). Before each execution distance variable has to be re-set to 7 and case to True.
   This aspect favors fields which allow for more prosperous development of our own structure – especially fields located in the center of the board and those which link with our other formations without opponent's stones in between.

2. **Immediate pay-off**: During the inspection of lines in 4 directions we also check whether in the immediate vicinity there are any stones of our color so as we could extend our structure and therefore be awarded with more points. Then we calculate the pay-off as shown in Algorithm 5.2.

3. **Blocking opponent's structure**: This strategy works exactly the same as Immediate pay-off from the point above with one difference – we investigate the field from our opponent's perspective. If he would place the stone here what would be his pay-off. Then the length of possible structures are re-scaled. This enables us to add additional values to

[5]Their names are our own invention.

those points which are of our opponent's interest. This certainly favors those moves which can improve our own structures (look points above) and simultaneously mess up our opponent's plans.
In order to emphasize the importance of *Immediate pay-off* aspect the values that represent the reward of a structure that they create are shifted up by one stone, e.g. 3-stone row is awarded with 10 points instead of 3, 4-stone row is awarded with 25 instead of 10, and so forth.

Algorithm 5.1 Heuristic evaluation – inspection of one direction.

```
 1: potential = 0
 2: distance = 7
 3: vertical_adjustment = 1
 4: case = TRUE
 5: field = field.GETNEXT()
 6: while field.EQUALS(empty) or field.EQUALS(myColor) do
 7:     if field.EQUALS(empty) then
 8:         case = FALSE
 9:         potential = potential + distance
10:         distance = distance - 1
11:     else
12:         potential = potential + distance + 2
13:         distance = distance - 1
14:         if case is TRUE then
15:             vertical_adjustment = vertical_adjustment + 1
16:         end if
17:     end if
18:     field = field.GETNEXT()
19: end while
```

Next values of all 3 aspects are summed up as the value of potential of a field. The procedure repeats for each empty field and then further selection can be performed.

This heuristic evaluation intentionally omits total score of the game because we aim to evaluate one particular move and value of awarded points for structures not affected by our move could be misleading in terms of further selection.

Algorithm 5.2 Heuristic evaluation – pay-off adjustment (vertical line).

```
1:  switch vertical_adjustment do
2:      case 7
3:          potential = potential + 246
4:      case 6
5:          potential = potential + 119
6:      case 5
7:          potential = potential + 56
8:      case 4
9:          potential = potential + 25
10:     case 3
11:         potential = potential + 10
12:     case 2
13:         potential = potential + 3
14:     case 1
15:         potential = potential + 2
```

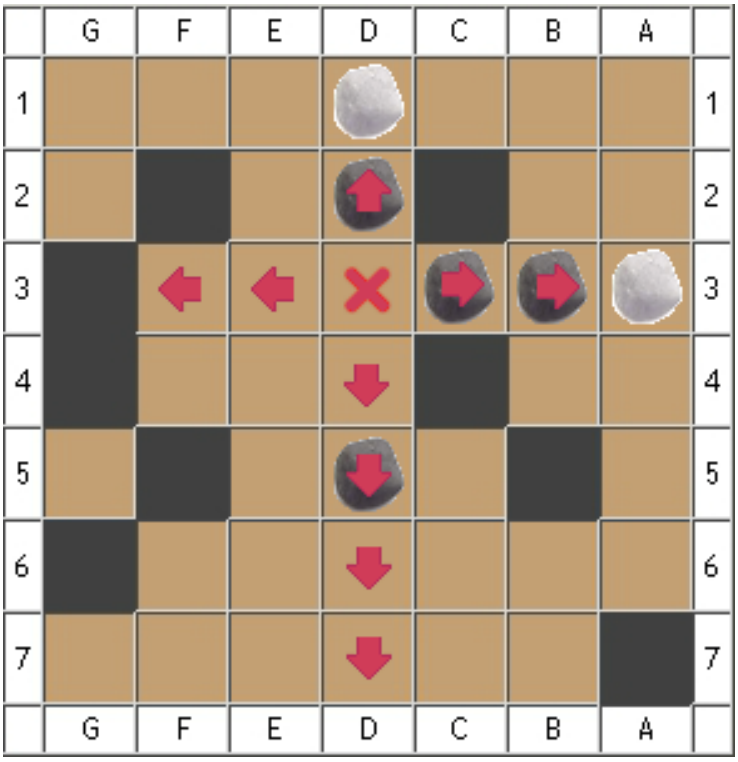

Figure 5.2: Heuristic evaluation – open lines (evaluation for black, D3). This is unrealistic situation and serves only as a demonstration.

5.4 Solely Heuristic-based Agent

The measure of quality of heuristics is accurate distribution of potential across the board so the best moves are favored. Further selection is predicated upon those results (often not only one best play is selected but several fields with the highest potential).

In order to assess the quality of heuristic we created an agent that is solely based on heuristic evaluation. It places the stone on the field with the highest potential according to the heuristic evaluation. When more than one field is awarded with the same value then it picks at uniformly random one from their collection.

This agent does not perform any roll-outs and manages to play a very credible game on all sorts of boards. The comprehensive results from various tests are largely presented in Chapter 6. Due to its relatively high quality of play the agent was used as a baseline for the further study.

The agent was named *Charles* and hereafter is referred so.

5.5 Heuristically Enhanced Selection Policy

We realize that in a long run MCTS can produce good approximation of game-theoretical values of choice-nodes, however, due to timely fashion under which algorithm has to operate this approximation is vulnerable to be inaccurate. Faulty results may be misleading and direct search explorations toward sub-optimal decisions.

In Chapter 6 we benchmark the agent with implemented UCT algorithm and the results of it are not impressive – in 100 game match it managed to win 1 game and lost all the others against *Charles*. We consider the *Selection policy* in its raw form as a weakness of this algorithm and subsequent subsections address our concerns (*Simulation policy* is also regarded as erroneous and we elaborate on it in the section 5.6).

Two factors lean toward critical appraisal of *Selection policy*.

1. In most positions there are only few reasonable candidate plays and all remaining can be regarded as unpromising (they often lose a *tempo* and have less impact on the strategic points and developed structure). Experienced human players can conduct such evaluation in a very short time without doing any calculations. Making a sub-optimal move in the early stage of the game can even jeopardize the entire game because of significant difference in points awarded for rows of various length. We

deem that in this game during the decision process of the next move substantial group of candidate moves can be discarded. Only reasonable plays should be included in the game tree for further exploitation and we strongly advocate forward pruning.

2. We consider the selection process of multi-armed bandit problem as another weakness in this algorithm. Our biggest concern touches the aspect of approximation, which is predicated on self-played random games that have very little in common with the games played by reasonable players. We realize that in a very long run[6] they might very well reflect the statistical tendency (and accurately approximate game-theoretical value) however since the number of roll-outs has to be limited due to timely fashion of the algorithm this approximation becomes vulnerable.

5.5.1 Beam Search

We invented two beam search strategies which narrow the scope of candidate plays based on heuristic evaluation. Our preliminary tests have shown that *Charles* is not playing perfectly (but relatively good), which implies that heuristic evaluation is not perfect. Nonetheless, we deeply hope that the best play is always among the top $5-10$ moves assessed with the highest potential by heuristic function.

1. All nodes representing choice of algorithm's opponent are initially equipped only with n candidate plays as untried move collection (those moves are explored when the node is expanded). Only the first n moves with the highest potential values as per heuristic evaluation are incorporated. Therefore the algorithm's choice is made between the top n moves according to the heuristic. Nodes representing choice of the algorithm are in their initial state fully equipped with all valid moves.

2. All nodes regardless which player's choice they represent are initially equipped only with n best moves adhere to heuristic evaluation as a collection of untried moves.

Those two enhancements go side by side with improving *Simulation policy* because certain moves will never be played however they can be used in raw version of the *Simulation policy* in order to approximate the game-theoretical value of a nodes, which is logically incoherent and erroneous.

Those improvements impose changes in Algorithm 2.3 in lines 2 and 22.

[6] We believe few millions of roll-outs would be sufficient enough to accurately approximate game-theoretical values in the early stage of this game.

5.5.2 The Gibbs Distribution

The performance of agents which implemented beam search technique presented in the section 5.5.1 was by far better than plain UCT. However it still could not cope with *Charles* which was winning about 90% of the games. In this enhancement we stick to the beam search idea since it significantly improves the quality of play but we focus on addressing concern regarding multi-armed bandit dilemma by using the Gibbs distribution inspired by statistical physics.

As we learn from Landau and Lifshitz [17] the objective of this technique is to find the probability w_i of a state of a whole system such that the body concerned is in some defined quantum state i (with energy E_i), i.e. microscopically defined state. After few transformation we have finally for w_i the expression

$$w_i = A \exp(-E_i/T) \tag{5.1}$$

where A is a normalization constant independent of E_i and T is the temperature of the system; the temperatures of the body and the medium are the same, since the system is assumed to be in equilibrium. This formula gives a statistical distribution of any macroscopic body which is comparatively small part of a large closed system. The other name frequently used in the literature is canonical distribution.

Given the condition $\sum_i w_i = 1$ and substituting $-E_i$ with α_i as a potential calculated by heuristic evaluation function we obtain a formula

$$p(\alpha_j) = \frac{\exp(\alpha_j/T)}{\sum_{j=1}^{n} \exp(\alpha_j/T)}. \tag{5.2}$$

which is a distribution of probability of selecting choice i predicated upon constant T. Higher values of T make the distribution more flat and approach to the state where all choices have the same probability. Lowest values of T increase the probability of choices of higher potential, and decrease those with lower potential.

We persistently create new nodes with only best n candidate play according to heuristic evaluation and in addition we use this technique every time the BestChild method is invoked in Algorithm 2.2 except from line 8 where we select *Max child* (the child with the highest number of wins). Preliminary tests indicated that selecting *Robust child* results in much worse play. Obviously root node does not have a potential since it has not got a parent node.

In order to facilitate this process and do not significantly increase computational time few changes are required in our implementation as follows.

- Each node contains information about the potential (based on heuristic evaluation) of the move that led to it. Therefore the potential value is calculated only once when BestChild method is invoked for this node's parent.

- Each node contains information regarding probability calculated from (5.2). Obviously the value is assigned when all sibling nodes are created, thus when BestChild is invoked for the first time for this node's parent. After that its calculations are stored with no need to redo them.

- In order to conduct the final selection child-nodes are organized in a generic list which enables to order them accordingly to calculated probability so as the additional accompanying values form boundaries of probability range to which random samples are assigned. Standard Java mechanism is used to provide a random number from $[0, 1]$ range.

In order to obtain the best quality of play the temperature T needs to be tuned. Different values may suit better different beam search techniques regarding the number of not pruned candidate plays. Each case have to be investigated individually.

5.6 Heuristically Enhanced Simulation Policy

We deem that unsatisfying results of UCT's performance are caused by inaccurate approximation of nodes' game-theoretical values which are misleading. Approximation mechanism relies on self-played random games which often resemble very little the games played by reasonable players. In order to improve this aspect we propose to use similar to the beam search technique, namely random moves are made from the collection of candidate moves provided by the heuristic evaluation function, say n best moves. Before each turn the evaluation function investigates the current status of the board and designates n best moves for color which is about to make the next move.

Another supporting idea is that when we apply a beam search strategy to agent A certain moves will never be done by the agent A in the first place and therefore there is no point to take them into account during the approximation process. Of course the degree of limiting the number of candidate moves has to be correlated with the beam search strategy.

During the simulation this technique can be applied to one color, or both. Even though beam search is applied to nodes of one color this enhancement applied for both sides can provide more accurate approximation.

5.7 Discarded Candidate Enhancements

Sections 5.5 and 5.6 emphasize enhancements implemented in our agents. However, during the development process we took into consideration all candidate techniques (motivated by enhancements created for *Minimax* algorithm presented in Section 2.2) and selected only those which we deemed to be promising and feasible. Following list provides discarded techniques with rationals of our decisions.

1. **Opening books**: Each new game starts on a board with odd number of *black holes* randomly distributed. In practice, chances of encountering exactly the same board twice are nearly zero. Therefore creating any sort of opening books becomes pointless and unfeasible due to the amount of possible setups. Unfortunately, even initial moves for one particular board do not have to be the best choice for the other one.

2. **Probabilistic cut**: As we mentioned previously, random distribution of *black holes* minimizes chances of encountering twice any initial position almost to zero. This fact makes almost every game unique and players face situations they have never seen before. We agree that some games feature similar sequence of moves in certain stage of the games. However from our experience it occurs very rarely. Implementing such pattern recognition would cause significant increase of computational time. Nonetheless, static pattern recognition in a sense of stone structures that are already placed on a board can be very useful for developing better action heuristics. We introduce this notion in more detail in Section 7.2.3.

3. **Quiescence search**: In this project we utilize action heuristics. In order to evaluate whether investigated state is quiescent or further explorations are required static heuristics would be better suited. Nonetheless, we consider merging this technique with MCTS method as very promising and return to this idea in Section 7.2.2.

4. **Transposition table**: It would almost certainly improve quality of the performance of MCTS algorithm since some nodes represent the same positions and make some parts of the game tree redundant. Although in Chess programs this technique was applied only to particular states selected by heuristic evaluation – applying it to all nodes in the game tree would be an overkill. We do not see how our action heuristic evaluation could be used for this purpose. Nonetheless, designing static evaluation which recognizes patterns in existing stone structures could be used for this purpose. We also mention this approach as a candidate research topic in Section 7.2.2.

5.8 Developed Monte Carlo Tree Search Agents

Here we document created agents which make use of aforementioned improvement technique. In order to assess the quality of certain technique we test them against each other on a different board. Our findings are presented in the next chapter.

In order to distinguish them easily we invented an intuitive naming convention. Agents are listed below.

1. **MCTS UCT**: Plain UCT as presented in Algorithm 2.2.
2. **MCTS+H(5)**: Improved on *Selection policy* by applying beam search to agent color's decision process during creating the game tree (the opponent's color was using all valid moves). BestChild as in pure UCT. The heuristic evaluation function provides 5 best moves. *Simulation policy* is also improved – every time during the simulation agent's color makes a move from collection of the best 5 provided by the heuristic evaluation function for each position.
3. **MCTS+H(7)**: The same notion as agent no. 2 is used. Beam search with is applied to agent color's decision process with the scope narrowed down to 7 candidate plays. BestChild as in pure UCT. During the simulation process agent's color makes a move from collection of the best 7 provided by the heuristic evaluation function for each position.
4. **MCTS+H(7)err**: The same implementation as agent MCTS+H(7) with 2 exceptions: (1) $c = 2/\sqrt{2}$ and (2) in Algorithm 2.3, line 8 BestChild method takes as second parameter c, not 0. We implemented it by mistake, but in tests it provided very intriguing outcomes and we decided to keep him.
5. **MCTS+H(10)**: The same notion as agent no. 2 is used. Beam search with is applied to agent color's decision process with the scope narrowed down to 10 candidate plays. BestChild as in pure UCT. During the simulation process agent's color makes a move from collection of the best 10 provided by the heuristic evaluation function for each position.
6. **MCTS+H(5+5)**: The same as MCTS+H(5) with only one difference. During *Simulation policy* both colors make their moves from the collection of best 5 provided by the heuristic evaluation function for each position.
7. **MCTS+Gibbs(5)(t=1.5)**: *Selection policy*: Beam search narrowed down to 5 best moves is applied. BestChild method utilize the notion of the Gibbs distribution. Temperature is set to 1.5.

Simulation policy: Both colors pick their moves at random from collections of best 5 moves provided by the heuristic evaluation function for each position.

8. **MCTS+Gibbs(5)(t=2.5)**: *Selection policy*: Beam search narrowed down to 5 best moves is applied. BestChild method utilize the notion of the Gibbs distribution. Temperature is set to 2.5.
   Simulation policy: Both colors pick their moves at random from collections of best 5 moves provided by the heuristic evaluation function for each position.

9. **MCTS+Gibbs(5)(t=4.0)**: *Selection policy*: Beam search narrowed down to 5 best moves is applied. BestChild method utilize the notion of the Gibbs distribution. Temperature is set to 4.0.
   Simulation policy: Both colors pick their moves at random from collections of best 5 moves provided by the heuristic evaluation function for each position.

10. **MCTS+Gibbs(5)(t=10.0)**: *Selection policy*: Beam search narrowed down to 5 best moves is applied. BestChild method utilize the notion of the Gibbs distribution. Temperature is set to 10.0.
    Simulation policy: Both colors pick their moves at random from collections of best 5 moves provided by the heuristic evaluation function for each position.

11. **MCTS+Gibbs(7)(t=1.5)**: *Selection policy*: Beam search narrowed down to 7 best moves is applied. BestChild method utilize the notion of the Gibbs distribution. Temperature is set to 1.5.
    Simulation policy: Both colors pick their moves at random from collections of best 7 moves provided by the heuristic evaluation function for each position.

12. **MCTS+Gibbs(7)(t=2.5)**: *Selection policy*: Beam search narrowed down to 7 best moves is applied. BestChild method utilize the notion of the Gibbs distribution. Temperature is set to 2.5.
    Simulation policy: Both colors pick their moves at random from collections of best 7 moves provided by the heuristic evaluation function for each position.

13. **MCTS+Gibbs(7)(t=4.0)**: *Selection policy*: Beam search narrowed down to 7 best moves is applied. BestChild method utilize the notion of the Gibbs distribution. Temperature is set to 4.0.
    Simulation policy: Both colors pick their moves at random from collections of best 7 moves provided by the heuristic evaluation function for each position.

14. **MCTS+Gibbs(7)(t=10.0)**: *Selection policy*: Beam search narrowed down to 7 best moves is applied. BestChild method utilize the notion

of the Gibbs distribution. Temperature is set to 10.0.
Simulation policy: Both colors pick their moves at random from collections of best 7 moves provided by the heuristic evaluation function for each position.

Chapter 6

Evaluation

This chapter documents conducted experiments and evaluates their outcomes. The primary intention for these tests was to provide empirical evidence of using heuristics for Monte Carlo Tree Search and assess which applications of them improve quality of the performance and which have a negative influence.

At first the reader is guided through outline of the experiments where we explain our purposes and how the outcomes were analyzed. Next we elaborate on the test cases featuring this project. Their designs were influenced by the preliminary tests of the agents as well as results of previous test cases. Closing subsection addresses non-functional requirements.

6.1 Evaluation Aims

The main question of this project is to assess usage of heuristics for Monte Carlo Tree Search algorithm. Since various enhancements go hand in hand (i.e. beam search techniques impose certain *Simulation policy* enhancements) we implemented few agents featuring different configurations of those. Different agents compete with each other in matches[1] made up of multiple sets. We conduct thousands of games and therefore we rely on general statistics of the matches, rather than on inspection of each game individually, move by move. In order to provide a comprehensive evaluation we decompose the analysis of the single match into six basic factors as follows.

1. **General strength of the agent**: We assess the strength of the agents based on their overall score in the matches in respect to statistical confidence discussion provided in section 6.2. We draw a comparison between

[1]The term *match* is interchangeable with *experiment*.

agents who performed the same number of roll-outs and used the same collection of boards in their matches.

2. **Utilizing the first player advantage**: It is expected that statistics of won games as player #1 and player #2 will resemble those obtained by the human players presented in table 2.1. Flat distribution can suggest that despite the fact that one player is significantly better than the other it can still play rather poorly and therefore further improvements are advised.

3. **Behavior on different boards**: During the tests we want to address the question whether the same algorithm performs equally well on different boards or not. As we pointed out in section 2.5, human players apply different strategies depending on the type of the board. All enhancement techniques make use of exactly the same evaluation function.

4. **Increase of roll-outs affects the performance**: According to Chaston et al. [8] increasing number of roll-outs should improve the quality of play, which is obvious implication of mathematical Monte Carlo method. Our investigation aims to prove, that it's also true when search strategies are biased by heuristic knowledge. In this analysis we take into account only matches where agents differ in number of roll-outs.

5. **Tune parameters**: *Selection policy* enhancements incorporate various calculations which aim to provide the most useful selection of nodes. Those calculations are predicated upon constant, e.i. temperature T in the Gibbs distribution. In order to maximize chances of winning they have to be tuned individually for each case. The purpose of this investigation is to determine the best values.

6. **Special circumstances**: Special considerations are drawn to cases which significantly vary from predicted results (in terms of proportion as well as quantity). For instance when one agent always wins exactly the same number of games on the same set of boards we will investigate whether exactly the same boards favor this player or is it just a coincidence.

Factors listed above cannot be prioritized in a sense of their importance to the evaluation as a whole. They inspect different aspects but when brought together they provide complete analysis of the agent. Then we formulate critical appraisal of different enhancement techniques based on agents' assessments of performance under various conditions.

6.2 Outline of the Experiments and Their Constraints

To recapitulate from section 5.2.3, the experiment is organized in the following manner:

- Agents play 50 sets between each other, hence 100 games. Set requires 2 games be played with players switching sides in the second game where both games are conducted on the same board. This is done in order to reduce player advantage since the game is biased and favors the player who makes the first move.
- Each set uses different boards, however all boards used in one experiment have the same number of black holes, randomly distributed. This approach minimizes the chances that board's setup would influence the outcome of the experiment.
- The same collections of boards are used in different experiments in order to make the results of the tests comparable between each other.

6.2.1 Statistical Confidence Discussion

How many games do we need to conduct in order to obtain expected results with expected probability? We base our theoretical discussion on Chebyshev inequality [4].

Theorem 2. *Let X be a random variable with mean EX and variance D^2X. Then for any $\varepsilon > 0$*

$$P(|X - EX| \geqslant \varepsilon) \leqslant \frac{D^2X}{\varepsilon^2}. \tag{6.1}$$

In order to keep the calculations simple we assume that in conducted experiment random variable adheres to Bernoulli distribution where $P(X = k) = \binom{n}{k}p^k q^{n-k}$, $q = 1-p$, $k = 0, 1, ..., n$, $EX = np$, and $D^2X = npq$. Let us examine Chebyshev inequality (6.1) in case when $X = \frac{k}{n}$, where $k = 0, 1, 2, ..., n$. Thus $EX = p$, $D^2X = \frac{pq}{n}$ and formula (6.1) becomes

$$P(|\frac{k}{n} - p| \geqslant \varepsilon) \leqslant \frac{pq}{\varepsilon^2 n}. \tag{6.2}$$

Due to obvious condition $P(Y < K) = 1 - P(Y \geqslant K)$ inequality (6.2) turns into

$$P(|\frac{k}{n} - p| < \varepsilon) \geqslant 1 - \frac{pq}{\varepsilon^2 n}. \tag{6.3}$$

At first let us assume that probability of win by *agent A* against *agent B* in a single game is $p = q = \frac{1}{2}$. Let k be a number of won games by *agent A* and $\varepsilon = 0.1$. Based on (6.3) we calculate how many games n need to be played so that the probability of $|\frac{k}{n} - \frac{1}{2}| < 0.1$ is greater or equal than 0.7. Since $P \geqslant 1 - \frac{\frac{1}{2} \cdot \frac{1}{2}}{(\frac{1}{10})^2 \cdot n}$ therefore $1 - \frac{25}{n} \geqslant 0.7$ and finally $n \geqslant 83.33...$

Other scenarios,

- When we expect that *agent A* will beat *agent B* 90% of the times when $\varepsilon = 0.1$ with probability greater or equal than 0.9 we need to conduct $n \geqslant 90$ games.

- When we expect that *agent A* will beat *agent B* 70% of the times when $\varepsilon = 0.1$ with probability greater or equal than 0.79 we need to conduct $n \geqslant 100$ games.

Taking into consideration above reasoning we decided to conduct all experiments with 100 games. Confidence of obtained results satisfies our expectations and empowers us to make further conclusions based on them.

Only twice we conducted the experiments made up of 400 games, where we expected that one agent will win 99% of the games with $\varepsilon = 0.05$ and probability of 99%.

6.2.2 Constraints

Due to timely fashion of this project and short deadline we were not able to conduct tests which would run for weeks. This factor influenced proposed probability limits, which obviously implicated number of games conducted during the experiments. Trade-off was made between factors influencing duration of the experiments – confidence of the results and number of agents' roll-outs.

After we finished 2 test cases we organized more exhaustive tests for selected agents in order to support our claims with even more credible results. The most exhaustive experiment took a little more than 15 hours. Altogether tests took 140 hours, excluding time needed for their preparations.

6.3 Preliminary tests

In few places we mention that preliminary tests guided us at first in terms of selecting some features or first tuning of the parameters. By this we mean us playing with the agent and subjectively assessing his quality of play based on very few games. For instance, the temperature T in the Gibbs distribution can be any number, therefore we had to decide whether it performs better with numbers from $[1, 10]$ range or $[40, 50]$. Then we conducted exhaustive tests in order to determine which exact value fit the algorithm really well.

What needs to be stressed, we were able to win with *Charles* most of the time. Agents employing the Gibbs distribution were much tougher opponents, however we still managed to beat them, though not as often as it took place with *Charles*.

6.4 Test Case 1 (Beam Search)

In the first test case we assess algorithms which employ beam search technique from 5.5.1, point 1, while playing pure UCT. Design of few experiments was influenced by previous results. Our objective was to omit ones with the most predictable outcomes.

6.4.1 Outline

In the test case we use two different sets of boards – each containing only boards with 3 or 11 *black holes*, all randomly distributed (hereafter referred to as *3-point board* and *11-point board*). Those types of boards require slightly different strategies in order to maximize player's chances of winning the game.

The full game tree for 3-point board contains $45! = 1.2 \cdot 10^{56}$ (for 11-point board it is $38! = 5.2 \cdot 10^{44}$). Hence we decided that $20,000$ and $50,000$ roll-outs performed by each agent before every move would be enough for creating significantly big game tree (especially when the process is biased by using heuristics). Also quick tests gave satisfying outcomes in terms of computational time needed.

Selection policy in all algorithms is featured by the BestChild method. Its calculations are predicated upon constant $c = 1/\sqrt{2}$ (as reported in [6] to be the best fit for Kocsis and Szepesvári program). We do not focus on tuning constant c in our implementations.

Tables 6.1 - 6.4 present the outcomes of tests conducted in different setups. Across all the tests we use shortened notations where A refers to agent, $=$

symbolize a draw, and *A1 W (P2)* needs to be read as agent #1 wins as player #2. In the section 6.4.2 we analyze them.

Agent 1	Agent 2	A1 W	=	A2 W	A1 W (P1)	A1 W (P2)	A2 W (P1)	A2 W (P2)
MCTS(UCT)	MCTS+H(5)	3	0	97	2	1	49	48
MCTS(UCT)	MCTS+H(5+5)	2	0	98	2	0	50	48
MCTS(UCT)	MCTS+H(7)	5	0	95	3	2	48	47
MCTS(UCT)	MCTS+H(10)	6	0	94	5	1	49	45
MCTS+H(5)	MCTS+H(7)	72	0	28	35	37	13	15
MCTS+H(7)	MCTS+H(10)	78	0	22	41	37	13	9
Charles	MCTS+H(5)	95	0	5	48	47	3	2
Charles	MCTS+H(5+5)	93	0	7	47	46	4	3
Charles	MCTS+H(7)	97	0	3	47	50	0	3
Charles	MCTS+H(10)	97	1	2	49	48	2	0

Table 6.1: Experiment #1 for 3-point boards and 20,000 roll-outs.

Agent 1	Agent 2	A1 W	=	A2 W	A1 W (P1)	A1 W (P2)	A2 W (P1)	A2 W (P2)
MCTS(UCT)	MCTS+H(5)	1	0	99	1	0	50	49
MCTS(UCT)	MCTS+H(5+5)	0	0	100	0	0	50	50
MCTS(UCT)	MCTS+H(7)	5	0	95	3	2	48	47
MCTS(UCT)	MCTS+H(10)	8	0	92	5	3	47	45
MCTS+H(5)	MCTS+H(7)	64	0	36	33	31	19	17
MCTS+H(7)	MCTS+H(10)	70	0	30	36	34	16	14
Charles	MCTS+H(5)	96	0	4	49	47	3	1
Charles	MCTS+H(5+5)	96	0	4	48	48	2	2
Charles	MCTS+H(7)	93	0	7	45	48	2	5
Charles	MCTS+H(10)	99	0	1	50	49	1	0

Table 6.2: Experiment #2 for 3-point boards and 50,000 roll-outs.

Test Case 1 (Beam Search)

Agent 1	Agent 2	A1 W	=	A2 W	A1 W (P1)	A1 W (P2)	A2 W (P1)	A2 W (P2)
MCTS(UCT)	MCTS+H(5)	0	0	100	0	0	50	50
MCTS(UCT)	MCTS+H(5+5)	0	0	100	0	0	50	50
MCTS(UCT)	MCTS+H(7)	6	0	94	4	2	48	46
MCTS(UCT)	MCTS+H(10)	6	2	92	5	1	48	44
MCTS+H(5)	MCTS+H(7)	61	0	39	39	22	28	11
MCTS+H(7)	MCTS+H(10)	78	0	22	43	35	15	7
Charles	MCTS+H(5)	95	0	5	49	46	4	1
Charles	MCTS+H(7)	97	0	3	50	47	3	0

Table 6.3: Experiment #3 for 11-point boards and 20,000 roll-outs.

Agent 1	Agent 2	A1 W	=	A2 W	A1 W (P1)	A1 W (P2)	A2 W (P1)	A2 W (P2)
MCTS(UCT)	MCTS+H(5)	2	0	98	2	10	50	48
MCTS(UCT)	MCTS+H(5+5)	2	0	98	2	0	50	48
MCTS(UCT)	MCTS+H(7)	0	0	100	0	0	50	50
MCTS(UCT)	MCTS+H(10)	10	1	89	9	1	49	40
MCTS+H(5)	MCTS+H(7)	63	0	37	34	29	21	16
MCTS+H(7)	MCTS+H(10)	69	0	31	39	30	20	11
Charles	MCTS+H(5)	95	0	5	47	48	2	3
Charles	MCTS+H(7)	95	1	4	47	48	1	3

Table 6.4: Experiment #4 for 11-point boards and 50,000 roll-outs.

6.4.2 Analysis

The first test case clearly indicates that agents utilizing Monte Carlo Tree Search algorithm enhanced by the beam search technique easily outperform the agent who makes use of pure UCT algorithm. In all their matches heuristically enhanced agents managed to win 90% or so of the games. However, interesting tendency can be noticed – the more we widen the beam scope (increase number of candidate moves) the quality of play slightly decreases. Matches between heuristically enhanced agents only amplify this phenomenon. It can be interpreted that wider score of possible choices increase chances to make a wrong decision. It can be caused either by approximation strategy related to inadequate amount of roll-outs or incorrect *Selection policy*.

Matches between agents heuristically enhanced yields one very important conclusion – the statistics of won games as a player #1 and #2 are much more polarized when agents compete on the 11-point boards rather than on a 3-point ones.

In respect to general outcomes of the experiment of aforementioned algorithms change of boards does not affect them. Similar observation can be drawn regard increasing number of roll-outs.

Agent powered by the heuristic evaluation only, *Charles*, outperformed all agents despite change of boards, or increase roll-outs of the opponent agents. Detailed statistics of won games by Monte Carlo agents are peculiar. They very often indicate that agents played more effectively as a player #2.

In comparison to results obtained from games among human players very surprising is marginal number of draws. We expected it to be less than shown in statistics in table 2.1 but not almost none. On agent's defense, they use our heuristics which primary objective is to maximize the immediate pay-off (such behavior is often called greedy policy).

6.5 Test Case 2 (The Gibbs Distribution)

The second test case aims to assess the strength of agents making use of the Gibbs distribution (see section 5.5.2). Also we direct our focus on tuning parameters in order to maximize winning chances.

6.5.1 Outline

As previously, here we also use two different set of boards, 3 and 11-point, for exactly the same reason. All participating agents, except from *Charles* were set at $20,000$ roll-outs. In additional tests (Section 6.6, Table 6.10) we prove that increase of roll-outs does not positively affect quality of the performance in terms of agents utilizing the Gibbs distribution enhancement. We need to keep in mind that aforementioned agents need more time to perform the same number of roll-outs than their counterparts enhanced with beam search technique introduced in section 5.5.1.

We reduced the number of tests by eliminating paring where one agent was very likely to win all or almost all games. After 2 experiments between agents utilizing the Gibbs distribution and ones enhanced by the beam search (Table 6.5) we decided to drop all remaining tests as they were pointless. To support this decision we can say, that beam search programs could not cope with *Charles*, who is easily outperformed by Gibbs implementations.

In recordings we use slightly modified notation. MCTS+Gibbs(5)(t=1.5) matches Gibbs(5/1.5) – number before forward slash refers to narrowed beam search policy whereas number after the slash to temperature T.

Test Case 2 (The Gibbs Distribution)

All conducted tests are presented in tables 6.5 - 6.9.

Agent 1	Agent 2	A1 W	=	A2 W	A1 W (P1)	A1 W (P2)	A2 W (P1)	A2 W (P2)
Gibbs(5/2.5)	Gibbs(7/2.5)	52	0	48	37	15	35	13
Gibbs(5/2.5)	MCTS+H(5)	97	0	3	47	50	0	3
Gibbs(5/2.5)	MCTS+H(7)	99	0	1	49	50	0	1

Table 6.5: Experiment #5 for 3-point boards and 20,000 roll-outs.

Agent 1	Agent 2	A1 W	=	A2 W	A1 W (P1)	A1 W (P2)	A2 W (P1)	A2 W (P2)
Gibbs(5/1.5)	Gibbs(5/2.5)	44	0	56	27	17	33	23
Gibbs(5/2.5)	Gibbs(5/4.0)	49	0	51	29	20	30	21
Gibbs(5/4.0)	Gibbs(5/10)	44	0	56	28	16	34	22
Gibbs(5/1.5)	Charles	87	0	13	45	42	8	5
Gibbs(5/2.5)	Charles	83	0	17	47	36	14	3
Gibbs(5/4.0)	Charles	85	0	15	46	39	11	4
Gibbs(5/10)	Charles	82	0	18	46	36	14	4

Table 6.6: Experiment #6 for 3-point boards and 20,000 roll-outs.

Agent 1	Agent 2	A1 W	=	A2 W	A1 W (P1)	A1 W (P2)	A2 W (P1)	A2 W (P2)
Gibbs(7/1.5)	Gibbs(7/2.5)	51	0	49	28	23	27	22
Gibbs(7/2.5)	Gibbs(7/4.0)	51	0	49	33	18	32	17
Gibbs(7/4.0)	Gibbs(7/10)	49	0	51	28	21	29	22
Gibbs(7/1.5)	Charles	86	1	13	45	41	8	5
Gibbs(7/2.5)	Charles	83	0	17	47	36	14	3
Gibbs(7/4.0)	Charles	85	0	15	47	38	12	3
Gibbs(7/10)	Charles	77	0	23	44	33	17	6

Table 6.7: Experiment #7 for 3-point boards and 20,000 roll-outs.

Agent 1	Agent 2	A1 W	=	A2 W	A1 W (P1)	A1 W (P2)	A2 W (P1)	A2 W (P2)
Gibbs(5/1.5)	Gibbs(5/2.5)	45	1	54	25	20	29	25
Gibbs(5/2.5)	Gibbs(5/4.0)	42	4	54	21	21	28	26
Gibbs(5/4.0)	Gibbs(5/10)	46	0	54	26	20	30	24
Gibbs(5/1.5)	Charles	81	4	15	41	40	9	6
Gibbs(5/2.5)	Charles	84	0	16	44	40	10	6
Gibbs(5/4.0)	Charles	80	0	20	41	39	11	9
Gibbs(5/10)	Charles	82	0	18	42	40	10	8

Table 6.8: Experiment #8 for 11-point boards and 20,000 roll-outs.

Agent 1	Agent 2	A1 W	=	A2 W	A1 W (P1)	A1 W (P2)	A2 W (P1)	A2 W (P2)
Gibbs(7/1.5)	Gibbs(7/2.5)	45	2	53	25	20	29	24
Gibbs(7/2.5)	Gibbs(7/4.0)	48	3	49	28	20	28	21
Gibbs(7/4.0)	Gibbs(7/10)	54	1	45	31	23	27	18
Gibbs(7/1.5)	Charles	83	0	17	45	38	12	5
Gibbs(7/2.5)	Charles	79	1	20	44	35	14	6
Gibbs(7/4.0)	Charles	77	3	20	41	36	13	7
Gibbs(7/10)	Charles	77	0	23	42	35	15	8

Table 6.9: Experiment #9 for 11-point boards and 20,000 roll-outs.

6.5.2 Analysis

Records collected in Table 6.5 clearly show a great advantage of agents utilizing the Gibbs distribution enhancement (hereafter referred to as G-agents) over agents improved only of pure beam search. Very strange thing occurred though – beam search agents managed to win only when playing as player #2. The actual scale of strength can be observed when G-agents play *Charles*, who managed to indisputably beat all previous implementations of MCTS.

G-agents tests indicate different behaviors of agents on different boards, as we expected in the Literature Review chapter. Experiments conducted on a 3-point boards shows that both beam restrictions – 5 and 7 works well. Their detailed statistics of won games are well balanced. Slightly better results are obtained when temperature T approaches 1.5 value.

However, experiments for 11-point boards shows slightly different tendencies. In general G-agents restricted to only 5 candidate plays during search process perform a little bit better than their counterparts adjusted at 7. But very peculiar occurrence can be observed in detailed statistics of won games. In

G-agents with 5 possible moves the statistics are flat and do not indicate the advantage of first player, but G-agents with 7 possible moves (which turned out to be slightly worse), indeed, very clearly reflect this regularity.

In general, results obtained on 3-point boards are narrowly better than for 11-point boards experiments. Therefore, the heuristics is better suited for boards featuring lowest number of *black holes*.

Another very interesting point is tuning temperature T. Although on a 3-point boards lowest temperatures were favored, experiments on 11-point boards are indicating another phenomenon. Different temperatures suit better differently adjusted beam techniques. For beam technique adjusted at 5 the best results yields $T = 2.5$ whereas for beam technique adjusted at 7 the best results are obtained with $T = 1.5$.

6.6 Test Case 3 (Additional Tests)

Although Test Cases 1 and 2 adhere chronological following tests collected in this section are from various stage of development. We present them because they either amplify previously obtained results by increasing confidence of the experiment, or are interesting to share due to their very surprising outcomes which exemplify strange behavior.

6.6.1 Outline

In Table 6.10 we present results of G-agents' performances depending on number of roll-outs. G-agents need approximately 4 times more time to conduct the same number of roll-outs. For instance:

- Match Charles vs MCTS+H(5) on a 3-point boards with 20,000 roll-outs took $4,262$ sec. (approx. 1h 11min).

- Match Charles vs MCTS+Gibbs(5)(t=2.5) on a 3-point boards with 20,000 roll-outs took $18,656$ sec. (approx. 5h 11min).

The results of this test influenced design decision of the Test Case 2, which only investigate agents set to 20,000 roll-outs.

Agent 1	Agent 2	A1 W	=	A2 W	A1 W (P1)	A1 W (P2)	A2 W (P1)	A2 W (P2)
Gibbs(5/2.5/50k)	Charles	85	1	14	47	38	11	3
Gibbs(5/2.5/100k)	Charles	85	1	14	47	38	11	3

Table 6.10: Experiment #10 for 3-point boards.

After completion of Test Case 2 we noticed significant advantage of G-agents over everyone else. Therefore we decided to design a test which aim to demonstrate the strength of our algorithms. First we confront our solely based on heuristic evaluation agent *Charles* with pure Monte Carlo Tree Search UCT implementation set at 100,000 roll outs. Then we investigate how well MCTS+Gibbs(5)(t=2.5) with roll-outs set at 20,000 can cope with aforementioned UCT with roll-outs set at 100,000. Both experiments are conducted on new collection of 200 boards, 3-point type each (in every set they play 2 games on the same board, thus 400 games).

Agent 1	Agent 2	A1 W	=	A2 W	A1 W (P1)	A1 W (P2)	A2 W (P1)	A2 W (P2)
Charles	UCT(100k)	399	0	1	199	200	0	1
Gibbs(5/2.5/20k)	UCT(100k)	400	0	0	200	200	0	0

Table 6.11: Experiment #11 on 3-point boards, 400 games each.

Table 6.12 features the performance results of erroneous implementation of MCTS+H(7)err agent, where $c = 2/\sqrt{2}$ and in Algorithm 2.3 with line 8 changed so BestChild method takes parameter c instead of 0 (which theoretically speaking is wrong, because in the last selection we ought not to focus exploration-exploitation dilemma). All tests are conducted on a 3-point boards.

Agent 1	Agent 2	A1 W	=	A2 W	A1 W (P1)	A1 W (P2)	A2 W (P1)	A2 W (P2)
H(7)err(50k)	Charles	51	0	49	28	23	27	22
Gibbs(5/2.5/50k)	H(7)err(50k)	87	0	13	48	39	11	2
H(7)err(100k)	Charles	3	0	97	2	1	49	48

Table 6.12: Experiment #12 for 3-point boards.

6.6.2 Analysis

Increasing number of roll-outs for G-agents turns out to be pointless. The statistics of their performances are exactly the same. We suspect that in

terms of this algorithm, augmenting the number of roll-outs can result in the same or even stronger approximation then it was previously. What is very comforting, they managed to win 94% of their games as a player #1. In order to improve the performance even more we need to either adjust the heuristics or simply write a better one. Adjusting heuristics can be done by weighting different aspects of the game while they are summed up during the evaluation process so as some would become more or less important than the others.

Results collected in Table 6.11 demonstrate strength of our algorithms. *Charles* outperforms the UCT implementation 99.75% times. What is interesting, in our test case he managed to lose only as player #1. MCTS+Gibbs(5)(t=2.5) gives no chance to UCT and wins all 400 games. UCT's performance is disappointing, however it could have done better with better tuned c. Table 6.10 exemplifies G-agents' performances when higher numbers of roll-outs are set.

Table 6.12 documents performance of erroneous agent (we made a small mistake during the development process, later it was fixed). It shows that the quality of performance decrease when number of roll-outs is elevated. This phenomenon has never occurred in any of our previous tests. This can be explained either by faulty approximation because of misdirection caused by c or including exploitation factor when final selection is made.

6.7 Summarized Evaluation

All 3 test cases yield very interesting results, most of which support our and referred authors claims. It turns out that employing good heuristic evaluation into Monte Carlo Tree Search algorithm significantly improves the quality of performance. The most important conclusions are:

1. **Improvement of the performance**: Our experiments show that no matter how tuned, both pure beam search and the Gibbs distribution, hand in hand with *Simulation policy* enhancements, significantly improve quality of the performance. The best tuning of parameters results that classic implementation of UCT algorithm is not able to win a single game, and in most of the cases its results approach the rounding error.

2. **The Gibbs distribution is better than pure beam search**: Probabilistic *Selection policy* which assesses probability of selecting candidate moves while building the game tree results in better approximation of nodes' game-theoretic values rather than BestChild method addressing multi-armed bandit by calculating upper confidence bounds.

3. **Augment of roll-outs does not improve the performance**: Our tests have not shown improvement of results were number of roll-outs

were elevated. In terms of G-agents we suspect that such phenomenon could be observed during transition from $1,000$ to $20,000$ roll-outs, however our tests make use of very large tree which already approximates game-theoretical values well.

4. **Type of board affects the performance**: Type of board influences the quality of play only in case of the best agents. It is similar to top human players who can take the advantage of special feature of the board. Proposed algorithms perform slightly better on boards with lesser number of *black holes*.

5. **Our heuristic proved to be good**: Collected results prove that created heuristics serves well for the purpose of Monte Carlo Tree Search Algorithm.

6.8 Satisfaction of Non-functional Requirements

Non-functional requirements are mostly relate to the qualities that software has, hence only when the tests are over we are able to address them.

- **NFR001 (The system should quickly provide heuristic evaluation of every candidate play for given position.)**: Match between MCTS(UCT) and MCTS+H(5) took $6,141$ sec. whereas match between MCTS+H(7) and MCTS+H(5) $8,611$ sec. Clearly duration does not double when agents are switched, therefore this requirement is satisfied.

- **NFR002 (The evaluation of candidate moves is fairly accurate.)**: *Charles* who solely relies on heuristic evaluation function can easily outperform agent utilizing pure UCT algorithm. Thus the requirement is satisfied.

- **NFR003 (The environment should provide live recording during the run of the experiment.)**: The test class provides a mechanism that secures recording after each game, however, overall statistics are recorded to the file right after the match is finished. Therefore, if exception raises half-way through the match, general statistics will be lost. Nonetheless, they can be easily recreated from collected data of completed games before the exception happened. Recording temporal statistics would make the file suffer from data overload. Hence we deem that the requirement is met.

- **NFR004 (All agents have to utilize the concept of Monte Carlo Tree Search in their decision process.)**: All agents with the excep-

tion of *Charles* are based on MCTS scaffolding which retains the core notion of MCTS method. Therefore the requirement is met.

- **NFR005 (The implementation should adhere to Java coding convention and be easy to re-use for others.)**: All methods are accompanied by the commentaries. Since development of an agent often goes hand in hand with comparing the code of other implementations to all agents we applied the rules of 80 character line limit in order to enable the developer easy comparing two source files on one screen. The requirements is met.

Chapter 7

Conclusions and Future Work

This chapter summarizes the project and spotlights future development of the method as well as the game.

7.1 Summary of the Project

In conclusion, we proved that using good heuristics for Monte Carlo Tree Search enhances quality of the performance. At first we created a new heuristics from scratch. Than we implemented an agent equipped only with this heuristic evaluation, *Charles*, who does not perform any roll-outs or simulations. He easily outperformed the UCT agent. In the next stage we used various enhancements to incorporate the same heuristic evaluation into UCT algorithm and they all managed to outperform *Charles* and, of course, pure UCT. What is even more satisfying, heuristically enhanced agents need to perform much less roll-outs rather than pure UCT in order to play on a very good level, unreachable for pure UCT in a timely fashion.

7.2 Future Work

The subject of Monte Carlo Tree Search is very broad and constant interest of scientific community in this domain can assure that new approaches and enhancements will shortly emerge. The next subsections aim to outline possible directions of further development.

7.2.1 Further Study on Using Heuristic for Monte Carlo Tree Search

This project features the ancient game, by most forgotten. However human players over the decades created number of good heuristics for much more popular games, such as, Chess or Othello. Addressing them by Monte Carlo Tree Search enhanced by heuristic evaluation methods could yield interesting analyses.

However, as good as it sounds, we envision few obstacles on this route. As we pointed out in Section 5.1, where we voiced our reasons for selecting PahTum as our testing game, running simulations for Chess would cause a lot of troubles. We implemented Chess in Python and it managed to perform on average 33 roll-outs in a minute. Obviously Python is not the best suited language for this purpose, but reaching the efficiency of PahTum does not look feasible. The problem with Chess are constant evaluation of terminal state and providing list of valid moves, which from obvious reason cannot put their own king in a check situation.

Intriguing approach would be utilizing 2 or more different heuristics where each is tailored for different stage of the game, i.e. one dedicated for opening moves, one for middle game, and one for endings. They could be applied to both, *Selection* and well as *Simulation policies*.

Another interesting research topic would be application of Monte Carlo Tree Search to games with imperfect-information, such as, Stratego. There is some potential since Monte Carlo Tree Search was successfully implemented with Backgammon and obtained very good results.

7.2.2 PahTum and Its Future

Our currently best implementations do not play perfectly – we managed to beat them once in a while as a player #2. However, introducing transposition table may very well improve the game, or at least decrease number of roll-outs needed for sufficient approximation of nodes' game-theoretical values in the game tree.

Using different heuristics, or as we mentioned in the preceding subsection few heuristics could improve the performance. This could also include static evaluation of state in order to introduce the notion of quiescence search and merge it with Monte Carlo Tree Search.

7.2.3 Improving Heuristic Evaluation

In order to provide more accurate evaluation of the move our heuristics could incorporate pattern recognition. Figure 7.1 provides an example of one. Playing on a $C3$ may look like an unreasonable play but if we think what we would do if it would be our turn to play right after we played $C3$ then we would realize that we have a 2 alternative moves ($D3$ and $C4$, indicated by yellow color). After either of them we create a fork formation 2×2 which secures further development no matter how opponent will try to block it and it keeps the *tempo*, which in this game is crucial.

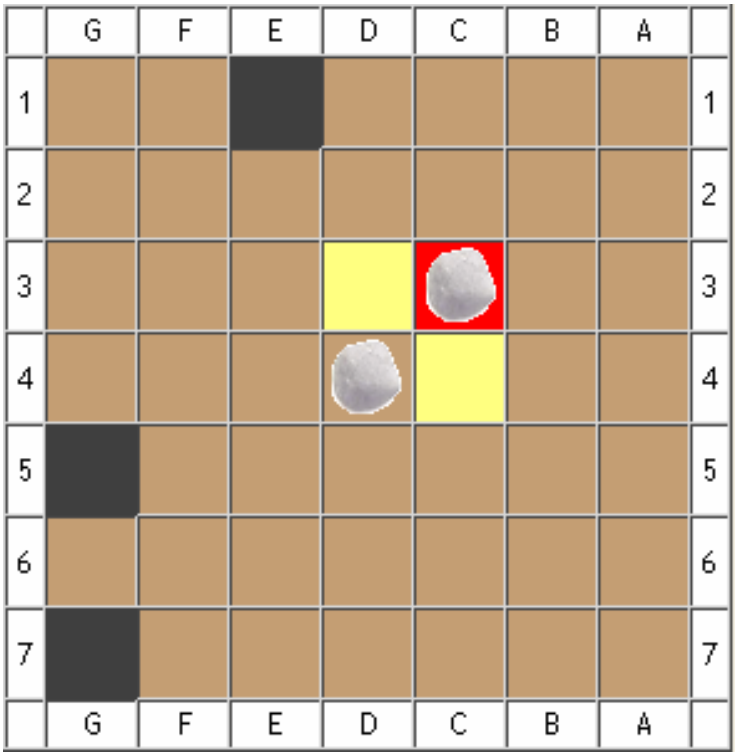

Figure 7.1: Example of pattern in PahTum.

7.2.4 Popularity of the Game

Future of PahTum is bleak. In general, people who know this game regard it as a simple one and therefore not worth spending time on. Although it is not as complex as Go, keeping playing this game may improve skills, such as, visual pattern recognition and sequential thinking of how your moves can affect decisions of your opponent and the other way around. People very often do not realize that most of the skills are easily transferable from one domain to another.

Creating an application which would enable to play against computer on different levels, from novice to expert, and then provide analyses of every move would be much of a help to the game. The idea is inspired by GNU Backgammon, an open source implementation of one of the best backgammon program. It significantly elevated the level of play of intermediate players and allow masters to conduct very deep analysis. This also helped players to save a lot of money which earlier they had to spend on literature in order to achieve intermediate or advanced level of play.

Bibliography

[1] Peter Auer. Using confidence bounds for exploitation-exploration tradeoffs. *Journal of Machine Learning Research*, 3:397–422, 2002.

[2] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, 47(2-3):235–256, 2002.

[3] Peter Auer, Nicolò Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. Gambling in a Rigged Casino: The Adversarial Multi-Arm Bandit Problem. In *FOCS*, pages 322–331. IEEE Computer Society, 1995.

[4] Alexandr A. Borovkov. *Probability Theory.* Gordon and Breach Science Publishers, 1998.

[5] Cameron Browne. On the Danger of Random Playouts. *Int. Comp. Games Assoc. J.*, 34(1):25, 2010.

[6] Cameron Browne, Edward J. 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 Trans. Comput. Intellig. and AI in Games*, 4(1):1–43, 2012.

[7] G. Chaslot, M. Winands, H. Herik, J. Uiterwijk, and B. Bouzy. Progressive strategies for Monte-Carlo tree search. *New Matchematics and Natural Computation*, 4(3):343–358, 2008.

[8] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-Carlo Tree Search: A New Framework for Game AI. In Christian Darken and Michael Mateas, editors, *AIIDE*. The AAAI Press, 2008.

[9] James E. Clune. *Heuristic Evaluation Functions for General Game Playing.* PhD thesis, University of California, Los Angeles, 2008.

[10] James E. Clune. Heuristic evaluation functions for general game playing. *KI*, 25(1):73–74, 2011.

[11] Pierre-Arnaud Coquelin and Rémi Munos. Bandit Algorithms for Tree Search. In Ronald Parr and Linda C. van der Gaag, editors, *UAI*, pages 67–74. AUAI Press, 2007.

[12] Rémi Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. Donkers, editors, *Computers and Games*, volume 4630 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 2006.

[13] Fred Hicinbothem. Pah-Tum -the long form. `http://dinsights.com/POTM/PAHTUM/details.php`, 2006.

[14] Philip Hingston and Martin Masek. Experiments with Monte Carlo Othello. In *IEEE Congress on Evolutionary Computation*, pages 4059–4064. IEEE, 2007.

[15] Levente Kocsis and Csaba Szepesvári. Bandit Based Monte-Carlo Planning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *ECML*, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293. Springer, 2006.

[16] T. L. Lai and H. Robbins. Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, 6:4–22, 1985.

[17] Lev Landau and Evgeny Lifshitz. *Statistical Physics* , volume 5 of *Course of Theoretical Physics*. Pergamon Press, 1969.

[18] Kevin Leyton-Brown and Yoav Shoham. *Essentials of Game Theory: A Concise Multidisciplinary Introduction*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2008.

[19] Debbie Miesel. IBM100 - Deep Blue. `http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/`, 2011.

[20] John F. Nash. Equilibrium points in n-person games. In *Proceedings of the National Academy of Sciences of the United States of America*, 1950.

[21] Damien Pellier, Bruno Bouzy, and Marc Métivier. An UCT Approach for Anytime Agent-Based Planning. In Yves Demazeau, Frank Dignum, Juan M. Corchado, and Javier Bajo, editors, *PAAMS*, volume 70 of *Advances in Soft Computing*, pages 211–220. Springer, 2010.

[22] Filip Rachunek. BrainKing - Game rules (PahTum). `http://brainking.com/en/GameRules?tp=72`, 2012.

[23] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education, 2010.

[24] Ulrich Schwalbe and Paul Walker. Zermelo and the early history of game theory. *Games and Economic Behavior*, 34(1):123–137, 2001.

[25] Claude E. Shannon. XXII. Programming a computer for playing chess. *Philosophical Magazine (Series 7)*, 41(314):256–275, 1950.

[26] Guido van Rossum. Comparing Python to Other Languages. `http://www.python.org/doc/essays/comparisons/`, 2012.

[27] John von Neumann and Oskar Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 1944.

[28] Mark H. M. Winands and Yngvi Björnsson. Evaluation Function Based Monte-Carlo LOA. In H. Jaap van den Herik and Pieter Spronck, editors, *ACG*, volume 6048 of *Lecture Notes in Computer Science*, pages 33–44. Springer, 2009.