A General Game Playing agent using Monte-Carlo Tree Search with split moves

(Agent General Game Playing używający
Monte-Carlo Tree Search z podzielonymi ruchami)

Wojciech Pawlik

Praca licencjacka

Promotor: dr Marek Szykuła

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

18 lutego 2021

Abstract

In many games, moves consist of several decisions made by the player. These decisions can be sometimes viewed as separate moves, which is a common practice in multi-action games due to efficiency reasons. So far, the application of splitting moves has been mostly limited to such straightforward cases.

In the thesis, we generalize the concept of splitting and offer the possibility of arbitrary splits applied in any game. We design an algorithm that is an adaptation of Monte-Carlo Tree Search to effectively work with split moves, including enhancements such as action-based heuristics. We implement a General Game Playing agent within the Regular Boardgames system, where moves can be automatically split with different granularity based on any abstract game description. Our generic and efficient implementation allows conducting a pioneering study of agents playing on different game tree structures, aiming to answer what is the practical impact of the split design and how to get maximal benefits from it.

W grach często pojedyncze ruchy składają się z ciągu decyzji podejmowanych przez graczy. Poszczególne decyzje mogą czasem być traktowane jako oddzielne ruchy, co jest popularną praktyką w grach typu multi-action w celu uzyskania lepszej wydajności. Do tej pory podziały ruchów były wykorzystywane tylko w tego typu oczywistych sytuacjach.

W tej pracy uogólniamy ideę podziału ruchów i proponujemy możliwość zastosowania dowolnej strategii podziału ruchów w każdej grze. Przystosowujemy algorytm Monte-Carlo Tree Search, wraz z heurystykami, do efektywnego działania z częściowymi ruchami. Implementujemy agenta General Game Playing wykorzystującego system Regular Boardgames, który pozwala na automatyczne generowanie częściowych ruchów z podziałami różnej gęstości na podstawie opisu reguł gry. Nasza uogólniona implementacja jako pierwsza pozwala na wszechstronne i bezpośrednie porównanie ze sobą agentów bazujących na różnych strukturach drzewa tej samej gry. Badając to, próbujemy odpowiedzieć na pytania jaki jest wpływ użycia podzielonych ruchów na wyniki agentów oraz jak maksymalnie wykorzystać potencjał tej metody.

Acknowledgments

Great thanks to my advisor Marek Szykuła, who introduced me to General Game Playing and encouraged me to start a research in this area, for being a great help at each step of this research process. I thank Jakub Kowalski for preparing the two figures included in this thesis, conducing experiments on a grid, and preparing statistical tools. Also, I thank Jakub Sutowicz for the initial implementation of the RBG agent that became a foundation for implementing the extended version from this thesis.

Contents

Chapter 1

Introduction

1.1 Monte-Carlo Tree Search and General Game Playing

Benefits of simulation-based, knowledge-free, open-loop algorithms such as Monte-Carlo Tree Search (MCTS) [18] and Rolling Horizon Evolutionary Algorithm [25] are especially suited to work within environments with many unknowns. In particular, they are widely used in General Game Playing (GGP) [12], a domain focusing on developing agents that can successfully play any game given its formalized rules, which was established to promote work in generalized, practically applicable algorithms [7, 13]. Initially based entirely on Stanford's Game Description Language (GDL) [21], GGP expands over time as new game description formalisms are being developed, e.g., Toss [16], GVG-AI [24], Regular Boardgames [20], and Ludii [26].

Recent advances in search and learning support the trend of generalization, focusing on methods being as widely applicable as possible. Deep Q-networks were applied to learn how to play classic Atari games and achieved above human-level performance on most of the 49 games from the test set [22]. More recently, AlphaZero, shown how to utilize a single technique to play Go, Chess, and Shogi on a level above all other compared AI agents [32]. This work also shows advances of MCTS on a field so far reserved for the min-max family of algorithms [3].

In the trend of developing enhancements for MCTS [4, 1], we tackle the problem of influencing the quality of the search by altering the structural design of the game tree itself. In many games, a player's turn consists of a sequence of choices that can be examined separately. A straightforward representation is to encode these choices as distinct moves, obtaining a split game tree, instead of using a single move in the orthodox design. The potential applications go beyond games, as the method can be used for any kind of problem that is solvable via a simulation-based approach and its representation allows splitting of moves.

1.2 Related work

The idea of splitting moves is well known, but apparently, it was not given proper consideration in the literature, being either used naturally in trivial cases or restrained to follow the human-authored heuristic, both cases in rather limited aspects given how general applications of split technique can be.

For the particularly complex environments, split is regarded as natural and mandatory. This technique is widely used for Arimaa, Hearthstone, and other multi-action games [9, 15, 30]. Here, the reduced branching factor is considered to be the main effect, as otherwise, programs could not play such games at a proper level. The case of Amazons is the only one that we have found where agents playing respectively on split and non-split representations were compared against each other [17]. For multi-action games such as real-time strategies, where the ordering of actions is unrestricted, Combinatorial Multi-armed Bandits algorithms are often employed [23]. Using actions separately as moves in the MCTS tree was also considered under the name of *hierarchical expansion* [30]. So far, splitting was applied only for such multi-action games, where it is possible and natural to divide a turn into separate moves and process them normally, that is, without the need of modifying search algorithms.

A practical application of splitting is found in GGP, where many games are manually (re)encoded in their split variants just to improve efficiency. For example, some split versions of games like Amazons, Arimaa, variants of Draughts, or Pentago exist in GDL. However, it is generally unknown how using such versions affect the agents' playing strength, especially apart from efficiency. Additionally, splitting via artificial turns causes some repercussions, in e.g., calculating game statistics, handling the turn timer (an agent gets the same time for each move, so he gets more total time as he performs more moves that logically form the same turn).

A related topic is *move groups* [31, 6, 36], where during MCTS expansion children of every tree node are partitioned into a constant number of classes guided by a heuristic. The basic idea of move groups is to divide nodes of the MCTS tree into two levels, creating intermediate nodes that group children belonging to the same class. There is no reported application of move groups beyond Go, Settlers of Catan [29], and artificial single-player game trees for maximizing UCT payoff. From the perspective of splitting, (nested) move groups are a side effect, and not every partition can be obtained by splitting.

As splitting just alters the game tree, it is compatible with any other tree search algorithm that uses this game tree as the underlying structure. In particular, it affects action-based heuristics such as MAST and RAVE [10], which operate on moves. But unlike usual techniques, in most cases the split design improves efficiency, thus it is beneficial if it just leaves the behavior of algorithms unchanged.

To summarize, splitting was so far applied only for multi-action games, where

split moves can act as regular moves and are defined manually by a human decision. In the literature, a direct comparison of both approaches was reported only for one game, and fundamental questions remained open, such as: is it generally better to use split moves, to what extent it affects the playing strength, and how to effectively split single-action games. Answers to these questions could change the default way of processing many games if we find out that splitting (or non-splitting) improves the results. A proper study of these questions requires developing methods dealing with split moves and a comprehensive framework for performing experiments with many variants.

1.3 Contribution

We focus on the general technique of altering the game tree by introducing split and its possible effects, rather than its application to a particular game combined with expert knowledge. This thesis is devoted to the development and implementation of a general MCTS agent supporting split moves. The agent is a prototype primarily designed to conduct pioneering research on the topic.

We propose the *semisplit* algorithm, which is a general adaptation of MCTS that works with arbitrarily split moves. In particular, this allows splitting moves of games that are not considered multi-action, where moves cannot be split in a straightforward way. We also propose several ways of adapting action-based heuristics – common MCTS general enhancements – to work with game trees with split moves. This includes state-of-the-art improvements of these methods. Due to that, we are able to test the effect of splitting also beyond the pure MCTS algorithm. We propose many available variants of the algorithm, such as mixing the semisplit and the orthodox designs for different phases of MCTS.

The algorithm is implemented within a generic agent for the Regular Boardgames (RBG) GGP system [20] – a universal GGP formalism for the class of finite deterministic games with perfect information. In RBG, a few *split strategies* of different granularity are provided and tested, which split moves in an automatic way basing on the given general game description. The agent uses the RBG compiler, which for any given game, generates a reasoner computing moves and/or semimoves.

A challenging task was to keep the high and comparable efficiency of the agent in all variants. To utilize the full potential of fast RBG reasoning, the implementation of each part of the agent must meet its efficiency level, as otherwise, it would become a performance bottleneck. We use, for instance, original data structures (such as the hash map for contextual action-based heuristics) and non-standard implementations (e.g., just-in-time and conditional compilation, custom allocating methods, dedicated hash maps). Combined with the efficiency of the RBG reasoning algorithms [19], the agent is arguably the fastest existing GGP agent, which allows more extensive experiments. The utilization of specific optimization possibilities ad-

mitted by particular variants of the algorithm allows more fair comparisons, making differences in the efficiency more visible in the results.

The experiments are conducted on a set of classic board games, comparing agents using both orthodox and split designs. We test a number of variants of the algorithm, applying split moves selectively to different phases of MCTS, and including action-based heuristics to observe the behavior also for non-basic MCTS. From the results, we identify the most beneficial variants and conclude that splitting moves can greatly improve the player's strength.

This work is a part of a collective research with Jakub Kowalski, Maksymilian Mika, Jakub Sutowicz, Marek Szykuła, and Mark Winands. The contribution of the author is focused on the development and implementation of the agent, which is a major ingredient of that study.

The thesis is structured as follows: Chapter 2 describes the abstract concept of semisplit and defines the algorithm and its variants. Chapter 3 is devoted to the implementation of the agent within the Regular Boardgames framework. Chapter 4 contains the guide how to use the agent and set its parameters. Chapter 5 shows the experimental results. We conclude in Chapter 6 with main corollaries and future research directions.

Chapter 2

Semisplit algorithm

We describe the abstract concept of playing with split moves and the general semisplit algorithm. Here we consider it independently on a particular implementation, where games and moves get a concrete representation. We start with theoretical definitions and then describe the algorithm with its variants.

2.1 Semisplit game trees

2.1.1 Abstract game

We adapt a standard definition of an abstract turn-based game [28] to our goals. A *finite deterministic turn-based game with perfect information* (later called simply *game*) $\mathcal{G}$ is a tuple $(\mathit{players}_{\mathcal{G}}, \mathcal{T}_{\mathcal{G}}, \mathit{control}_{\mathcal{G}}, \mathit{out}_{\mathcal{G}})$, where: $\mathit{players}_{\mathcal{G}}$ is a finite non-empty set of *players*; $\mathcal{T}_{\mathcal{G}} = (V, E)$ is a finite directed tree called the *game tree*, where V is the set of nodes called *game states* and E is the set of edges called *moves*, $V_{\mathrm{n}} \subset V$ is the set of inner nodes called *non-terminal* states, and $V_{\mathrm{t}} \subseteq V$ is the set of leaves called *terminal* states; $\mathit{control}_{\mathcal{G}} \colon V_{\mathrm{n}} \to \mathit{players}_{\mathcal{G}}$ is a function indicating the *current player* at the given state; $\mathit{out}_{\mathcal{G}} \colon V_{\mathrm{t}} \times \mathit{players}_{\mathcal{G}} \to \mathbb{R}$ is a function indicating the final *score* of each player. For a non-terminal state $s \in V_{\mathrm{n}}$, the set of *legal moves* is the set of outgoing edges $\{(s,t) \in E \mid t \in V\}$. During a play, the current player $\mathit{control}_{\mathcal{G}}(s)$ chooses one of its legal moves. The game tree which is rooted at an *initial state* s_0. A play starts from s_0 and ends at a terminal state, which finally must happen because these are precisely the leaves of the game tree. We assume that all states are reachable in the game tree from s_0.

Two games are *isomorphic* if there exist bijections between the sets of players and between the game trees that preserve at each state the current player (if the state is non-terminal) or the score (if the state is terminal). For a state $s \in V$, the *subgame* $\mathcal{G}_s$ is the game with the tree obtained from $\mathcal{T}_{\mathcal{G}}$ by rooting at s and removing all the unreachable states from s.

2.1.2 Split-equivalence

For a given game G, we can roll up all the states that have assigned the same current player and form a connected component. Then, instead of a sequence of moves, the current player performs just one move. We define the *rolled-up game* of G, where each maximal connected component $S \subseteq V_n$ of non-terminal states with the same current player is replaced with one new state v_S, the incoming edge to the (unique) root of S now goes to v_S, and all the outgoing edges from S now begin at v_S. Two games are *split-equivalent* if their rolled-up games are isomorphic. Obviously, split-equivalent games are strategically the same, except for technical differences of the execution (e.g., turn time limit, counting turns).

2.1.3 Abstract semisplit game

Going deeper into a particular representation of a move, it usually can be partitioned into a sequence of smaller pieces, which we call *semimoves*; e.g., they can be atomic actions, artificial groups, or, in the extreme case, even single bits of a move representation. Computing semimoves can be, but not always is, computationally easier than full moves and sometimes may reveal structural information desirable in a knowledge-based analysis.

Often a natural and most effective splitting does not lead to a split-equivalent variant of the game, because not every available sequence of easily computed semimoves can be completed up to a legal move. This especially concerns single-action games, but also splits inferred automatically, where without prior knowledge it is difficult to determine if we obtain a proper split-equivalent game.

Example 1. *In Chess, a typical move consists of picking up a piece and choosing its destination square. It would be much more effective first to select a piece from the list of pieces and then a square from the list of available destinations computed just for this piece than to select a move from the list of all legal ones, which is usually much longer. However, we may not be able to make a legal move with the selected piece, e.g., because the king will be left under check (see Fig. 2.1).*

A remedy could be checking if each semimove is a prefix of at least one legal move. However, in many cases, this can be as costly as computing all legal moves, losing performance benefits or even decreasing efficiency. Instead, we can work on semisplit games directly.

In contrast with split-equivalent games, we require a different game definition, including additional information about intermediate states. We extend the definition of a game to a *semisplit game* as follows. Let V be now the disjoint union of V_n, V_t, and the *intermediate states* V_i. The set of non-terminal states V_n is a subset of inner vertices of the game tree, terminal states V_t is a subset of leaves, and V_i may contain states of both types. The states in V_n and V_t are called *nodal*. A semisplit game

must satisfy that the initial state s_0 is nodal, and for every non-terminal $s \in V_n$, the subgame $\mathcal{G}_s$ contains at least one terminal state. The latter condition ensures that from each nodal state, a terminal state is reachable. Yet, for an intermediate state, there may be no nodal state in its subgame; then this state is called *dead*. An edge is called a *semimove*. A *submove* is a path where nodal states can occur only at the beginning or at the end, and in the middle, there are only intermediate states. A semimove is a submove of length 1. Then, a *move* is submove between two nodal states.

The *rolled-up* game of a semiplit game is obtained by removing all dead states and replacing each maximal connected component rooted at a non-terminal state and containing only semistates below with one non-terminal state. A semisplit game $\mathcal{G}'$ is *equivalent* to a game $\mathcal{G}$ if the rolled up game of $\mathcal{G}'$ is isomorphic to $\mathcal{G}$. Then, splitting a game means deriving an equivalent semisplit game.

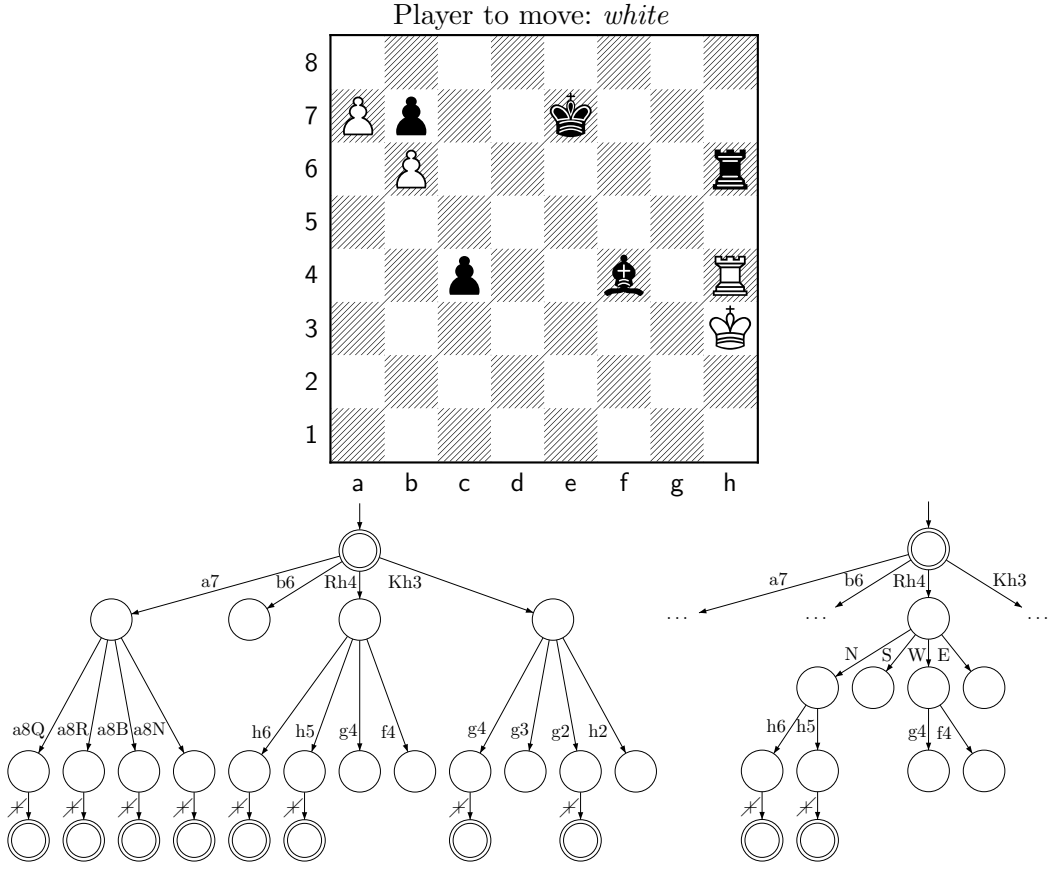

Figure 2.1: A Chess position (top) and the fragments of two semisplit game trees of smaller (left) and larger (right) granularity corresponding to the current game state. There are 8 legal moves in total denoted in the long algebraic notation ($\{$a7-a8Q, ..., Kh3-g2$\}$), which form 8 edges in the standard game tree. Nodal states are marked with a double circle; $\not$ indicates passed non-check king test.

An example of semisplit variants of Chess is shown in Fig. 2.1. There are fragments of the game tree with intermediate states between nodal ones.

2.2 The algorithm

2.2.1 Basic semisplit MCTS

Algorithm 1 Basic semisplit random simulation.

Input: s – game state

```
1: function SemisplitSimulation(s)
2:     while not s.IsTerminal() do
3:         m ← SemisplitRandomMove(s)
4:         if m = None then return None                ▷ s is dead
5:         s ← s.Apply(m)
6:     return s.Scores()
7: function SemisplitRandomMove(s)
8:     for all a ∈ Shuffle(s.GetAllSemimoves()) do
9:         s' ← s.apply(a)
10:        if s'.IsNodal() then return a
11:        m ← SemisplitRandomMove(s')
12:        if m ≠ None then return Concatenate(a, m)   ▷ Found legal move
13:    return None                                     ▷ No legal move
```

In the following description, we use a standard terminology [2] and focus on the differences with the orthodox MCTS. The simulation phase is shown in Alg. 1. Drawing a move at random is realized through backtracking (SemisplitRandomMove). Given a game state, we choose and apply semimoves in the same way as moves in orthodox MCTS, but we keep the list of legal semimoves computed at each level. When it happens that the current intermediate state does not have a legal semimove (is dead), we backtrack and try another semimove. Thus, we always find a move if it exists, and every legal move has a positive chance to be chosen, although the probability distribution may be not uniform. A single simulation (SemisplitSimulation) just uses the modified random move selection. Note that it can fail (**return None**), which happens if called for a dead state.

The semisplit MCTS is shown in Alg. 2. As orthodox MCTS, it uses the UCT policy in the selection phase [18]. But, in contrast, the semisplit MCTS uses both nodal and intermediate states as tree nodes. However, dead states are never added to the MCTS tree, which guarantees that we can reach a terminal state starting from any node in the MCTS tree. The expansion begins with the selection of an untried semimove. If the next state turns out to be dead, the semimove is removed from the list in the node, and the search goes back to the MCTS tree, so other untried semimoves are chosen; it can happen that all untried semimoves lead to dead states and thus the node becomes fully expanded, so the search continues according to the UCT policy.

The algorithm has two variants, *raw* and *nodal*, which differ in the expansion phase. The *raw* variant adds just one tree node as usual, either intermediate or nodal. However, this can leave semisplit MCTS behind orthodox MCTS, as a single node

Algorithm 2 Basic semisplit MCTS.

```
 1: function MCTSIteration( )
 2:     v ← TreeRoot()
 3:     while not v.State().IsTerminal() do
 4:         if v.FullyExpanded() then
 5:             v ← v.UCT()
 6:         else
 7:             a_1 ← RandomElement(v.UntriedSemimoves())
 8:             (v', scores) ← Expand(v, a_1)
 9:             if scores = None then continue
10:             Backpropagation(v', scores)
11:             return
12:     Backpropagation(v, v.State().Scores())

Input: v – leaf node in MCTS tree
Input: a_1 – selected untried semimove
13: function Expand(v, a_1)
14:     s ← v.State().Apply(a_1)
15:     scores ← SemisplitSimulation(s)
16:     if scores = None then
17:         v.RemoveSemimove(a_1)
18:         return None
19:     c ← CreateNode(s)
20:     v.AddChild(c, a_1)
21:     return (c, scores)

Input: v – last node of iteration in MCTS tree
Input: scores – final scores for each player of iteration
22: function Backpropagation(v, scores)
23:     while v ≠ TreeRoot() do
24:         v.scoreSum ← v.scoreSum + scores[v.Player]
25:         v.iterations ← v.iterations + 1
26:         v ← v.Parent()
27:     v.iterations ← v.iterations + 1
```

in the latter corresponds to a path between nodal states in the former. The *nodal* variant compensates this risk by adding that whole path during a single expansion. Usually, the nodal variant can increase the quality of the search when these paths are long and, in particular, cannot be exploited due to slower expansion. But, the raw variant is slightly faster. Alg. 2 shows the raw variant, whereas Alg. 3 replaces Expand with ExpandNodal for the nodal variant (called in Alg. 2 line 9).

There are two main reasons behind the efficiency benefits of semisplit. First, the number of semimoves is usually smaller than the number of moves, thus we reduce the branching factor. This particularly improves efficiency if the semimoves are cheaper to compute. Additionally, we have fewer choices to consider, which reduces the cost if we look at their statistics (e.g., in UCT or in MAST). Second, we do not always check if a move is legal. We maintain lists of move candidates that are lazily checked when selected so we can also benefit from having indecisive states in the semisplit game tree. On the other hand, traversing many intermediate states and frequent

Algorithm 3 The nodal expansion variant.

Input: v – leaf node in MCTS tree
Input: a_1 – selected untried semimove
function ExpandNodal(v, a_1)
  $s \leftarrow v$.State()
  $m \leftarrow$ SemisplitRandomMove(s.Apply(a_1))
  if $m =$ **None then**
    v.RemoveSemimove(a_1)
    return None
  $m' \leftarrow$ Concatenate(a_1, m)
  for all $a \in m'$ **do** ▷ For all semimoves a in m' in the order
    $s \leftarrow s$.Apply(a)
    $c \leftarrow$ CreateNode(s)
    v.AddNode(c, a)
    $v \leftarrow c$
  scores $\leftarrow$ SemisplitSimulation(v.State())
  return (v, *scores*)

backtracking can cause overhead compared to the orthodox computation.

2.2.2 Final selection

There are several policies to select the final best move to play. A common policy is to choose one that with the best average score [37]. In the case of ties, we use the largest number of iterations as the second criterion.

When it comes to selecting the final move to play, semisplit MCTS greedily chooses the best move semimoves till the first nodal state. If it goes outside the MCTS tree constructed so far, the raw variant chooses uniformly at random. This is very rare in practice and usually happens only when the branching factor is large enough compared to the number of iterations. In the nodal variant, this never happens, as leaves in the MCTS tree can be only nodal states.

We also propose another strategy called *final roll-up selection*. The idea behind it is that, in principle, we are mainly interested in performing the best full move rather than the best semimoves at each consecutive level, hence we should look at the score of the last semimove. However, there is a risk that there are semimoves with a high but unreliable average score, due to only a small number of iterations. Therefore, naive strategies that look just at the average score of the last semimove give overall poor results. On the other hand, determining the sufficient number of iterations is difficult, as the total number of iterations strongly depends on the game (actually, the current subgame) and the computation budget. Therefore, we propose to estimate this number based on the choice of the default greedy strategy. First, the greedy strategy finds a move, and we take the number of iterations of its last semimove. Then, this is multiplied by the *final roll-up factor* (FRF). In this way, if $FRF \geq 1.0$, final roll-up selection always chooses a move that is, in principle, not

worse than the greedy choice – it considers only moves with a not smaller number of iterations and the greedy choice is included. By decreasing *FRF*, we can increase the chance of selecting a move with a better average score at the cost of reliability in terms of the number of iterations. Finally, if the average score is equal to that of the greedy move, the latter is preferred. Similarly, if the greedy selection in the raw variant completes the move outside the MCTS tree, the final roll-up is not used.

2.2.3 Combined variants

Possible variants of semisplit MCTS involve combining both designs and using them selectively in different phases. First, there are two variants: the orthodox design in the MCTS tree phases (selection and expansion) together with the semisplit design in the simulation phase, and the opposite setting. Semisplit MCTS can work also on submoves that are dynamically obtained from concatenating semimoves.

Another proposed variant, modifying the MCTS tree phases, is the *roll-up*. It uses the semisplit design with the modification as follows. Whenever a non-root node v in the MCTS tree is fully expanded, i.e., all its children $v_1, \ldots, v_k$ were tried, then the algorithm removes node v and connects its parent v' with nodes $v_1, \ldots, v_k$. Then, nodes $v_1, \ldots, v_k$ become new children of node v', and the edges to them are submoves obtained from concatenating the submove from v' to v and the semimove from v to v_i. In this way, the semisplit design is mainly limited to the expansion phase, as the algorithm switches to the orthodox design in the parts of the MCTS tree that become fully expanded.

We can slow down the rolling-up process by tuning the *minimal simulation factor* (*MSF*) parameter. Creating new nodes from the given node v and its children $v_1, \ldots, v_k$, as described above, is performed when v is fully expanded and condition $v.iterations \geq k \cdot MSF$ is satisfied. Note that for $MSF = 1$, the full expansion of a node is enough to roll-up, because then $v.iterations \geq k$ is always fulfilled. The roll-up algorithm is shown in Alg. 4, where RollUp is executed during backpropagation for the MCTS nodes involved in the iteration (see Alg. 6).

Algorithm 4 The roll-up procedure.

Input: v – node to roll-up

```
 1: function RollUp(v)
 2:     if v.FullyExpanded() then
 3:         childrenCount ← v.Children().Length()
 4:         if v.iterationCount ≥ childrenCount · MSF then
 5:             v′ ← v.Parent()
 6:             m ← v′.Submove(v)                    ▷ Get the submove from v′ to v
 7:             v′.RemoveChild(v)
 8:             for all (c, a) ∈ v.Children() do
 9:                 m′ ← Concatenate(m, a)
10:                 v′.AddChild(c, m′)
11:                 v′.AMAF[m′] ← c.AMAF[a]          ▷ If RAVE is used
```

2.2.4 Action-based heuristics

MAST and RAVE

Commonly used general knowledge-free enhancements of MCTS are online-learning heuristics, which estimate the values of moves by gathering statistics. Two general methods are Move-Average Sampling Technique (MAST) and Rapid Action Value Estimation (RAVE) [8, 11, 37]. For both techniques, there are many policies proposed that differ in detail.

MAST globally stores for every move the average result of all iterations (samples) involving this move. Those values are updated after each iteration, and the score of each move is updated as many times as the move was applied in the iteration. The statistics are stored for each player separately. They are later used in the simulation and (optionally) expansion phases in place of the random selection. For each move from the set of all legal moves at some state, MAST defines the probability that this move will be selected. This probability distribution is defined depending on a particular policy, e.g. Gibb's distribution [8], Roulette wheel [27] or ϵ-greedy [35]. According to the previous research, ϵ-greedy is universally considered to be the best policy [27, 35]. It is also common to assume that if the move was not tried any time before, then its score is the maximum available reward, so it will be preferred to play as soon as possible.

An improvement for MAST is the *decaying* technique [34]. It comes from the insight that certain moves that are good in globally in the game may not be so strong in latter phases. In other words, that the global average may be not equally adequate for all subgames of the game. Decaying means that the weights of all the collected samples are multiplied by the *decay factor*, hence new samples gathered will have more impact on the average score. A few methods of decaying are known and proposed in [34]: *move decay* takes place after the effective move is applied in the game, *batch decay* applies decaying after a fixed number of simulation, and *simulation decay* applies decaying after each played move (of either the agent or the opponent). The last option was considered to be possibly the best.

Monte Carlo Tree Search needs many iterations to gather enough samples to differentiate the most promising moves. At the beginning, many choices are made randomly. RAVE tries to shorten this phase by storing additional statistics in nodes. Similarly to MAST, it stores the average reward for moves, but locally in MCTS nodes. These statistics are called *All Moves As First* (*AMAF*). For each node, each child with move m stores the average score (for the player of the node) from all iterations that passed through the node and that include m played at the node or at any place below. This is in contrast with the regular statistic of m in UCT, which is gathered only from iterations that play m at the node. In this way, AMAF collects much more samples, which are less reliable, however.

The AMAF statistics are used to bias the choice in the selection phase by modifying the UCB formula. The priority for each node is calculated as the weighted mean of the regular average score and the average stored for all simulations. There are many different formulas to calculate such weights [4, 8, 11, 33]. The general rule is that, for fewer visited nodes, the statistics stored due to using RAVE are more relevant due to a larger number of samples, but then they lose significance as they do not distinguish between the situations where moves are played.

Algorithm 5 MAST working on semimoves.

Input: *semimoves* – list of semimoves to choose from
Input: *player* – current player
1: **function** ChooseSemimoveWithMAST(*semimoves*, *player*)
2: **if** RandomRealNumber$(0, 1) \geq \epsilon$ **then**
3: $bestSemimoves = \arg\max_{m \in semimoves}(MASTstatistics[player].\text{Score}(m))$
4: **return** RandomElement(*bestSemimoves*)
5: **else**
6: **return** RandomElement(*semimoves*)

Input: s – game state
7: **function** SemisplitMASTMove(s)
8: $semimoves \leftarrow s.\text{GetAllSemimoves}()$
9: **while not** *semimoves*.Empty() **do**
10: $a \leftarrow$ ChooseSemimoveWithMAST($semimoves, s.\text{Player}()$)
11: $s' \leftarrow s.\text{Apply}(a)$
12: **if** s.IsNodal() **then return** a
13: $m \leftarrow$ SemisplitMASTMove(s')
14: **if** $m \neq$ **None then return** Concatenate(a, m)
15: **return None**

Both MAST and RAVE can be adapted to semisplit game trees and work in the pure semisplit design in a straightforward way, in the same manner as in the orthodox design. Alg. 5 shows the modified move selection in the simulation phase, where a move is selected with the ϵ-greedy policy. We use SemisplitMASTMove in place of SemimoveRandomMove that is called in Alg. 1, line 3. If statistics are used to select move in expansion phase, then line 7 of Alg. 2 is also replaced with ChooseSemimoveWithMAST(v.UntriedSemimoves(), v.Player()). RAVE just modifies the UCT selection (Alg. 2, line 5) and backpropagation (Alg. 6), and includes transferring AMAF values in the case of the roll-up variant (Alg. 4, line 11).

Split variants

In the combined variants, necessarily, both MAST and RAVE require some adjustments, which rely on dividing (sub)moves into semimoves or fusing semimoves into moves. Moreover, dividing moves into semimoves also give alternative strategies in the orthodox design.

The basic modification of MAST adapted to semisplit design is called MAST-split. It simply stores separate statistics for each semimove instead of for each move

Algorithm 6 Variants of backpropagation, depending on whether MAST, RAVE, or roll-up is used, respectively.

Input: v – last node of iteration in MCTS tree
Input: *scores* – final scores for each player of iteration
Input: *appliedSubmoves* – list of submoves applied in simulation phase

```
1: function BACKPROPAGATION(v, scores, appliedSubmoves)
2:     scores ← state.SCORES()
3:     for all (m, player) ∈ appliedSubmoves do                    ▷ with MAST
4:         MASTstatistics[player].UPDATE(m, scores)                ▷ with MAST
5:     while v ≠ TREEROOT() do
6:         v.scoreSum ← v.scoreSum + scores[v.Player]
7:         v.iterations ← v.iterations + 1
8:         v′ ← v.PARENT()
9:         m ← v′.SUBMOVE(v)                                       ▷ with MAST or RAVE
10:        MASTstatistics.UPDATE(m, scores, v′.PLAYER())           ▷ with MAST
11:        appliedSubmoves[v′.PLAYER()].APPEND(m)                  ▷ with RAVE
12:        for all (c, a) ∈ v.CHILDREN() do                        ▷ with RAVE
13:            if a ∈ appliedSubmoves[v′.PLAYER()] then            ▷ with RAVE
14:                v′.AMAF[a].UPDATE(score[v′.PLAYER()])           ▷ with RAVE
15:        ROLLUP(v)                                               ▷ with roll-up
16:        v ← v′
17:    TREEROOT().iterations ← TREEROOT().iterations + 1
```

or submove that can appear during the computation. Then, while evaluating the score of a submove, we need to combine somehow the result from the scores of the included semimoves. Here we propose the arithmetic mean of the scores of the semimoves; however, if a semimove has not been tried, then the maximum reward is returned as the final score of the whole submove. MAST-split can be applied to any variant of MCTS.

RAVE-split works similarly, i.e., it splits every submove into single semimoves. Statistics for a semimove in a tree node are updated if this semimove was used explicitly at the node or later in the iteration, either directly or possibly as a part of a submove. RAVE-split is easily enabled only in combinations using the semisplit design in the selection and expansion phases, i.e., when the domain of semimoves in these phases is the same as that of RAVE-split. It does not seem to be effective to if the domain of submoves in the MCTS tree phases is different than the set of single semimoves, because we would have to store the statistics separately for semimoves at each node and evaluate scores of submoves – the latter involves additional search for the statistics of particular semimoves.

Depending on the implementation, split variants can be much faster than regular heuristics due to much a smaller domain of semimoves (i.e., we can use faster data structures for storing statistics). However, obtained samples are less reliable, as semimoves carry less information than full moves.

Join variants

In opposition to the variants which rely on splitting moves into semimoves, there is also the option to combine semimoves into full moves. The main motivation for them is enabling RAVE to work in the variants with the orthodox tree design.

MAST-join and RAVE-join store statistics only for whole moves. After each iteration, all paths between nodal states are treated as moves, no matter which design is used. Of course, they are available only the variants where we do not need to evaluate semimoves separately.

Context variants

To partially overcome the weakness of less reliable samples, we introduce *context* variants. They lead to storing values of samples closer to those used in the pure orthodox design.

The main idea of MAST-context is entwined with *N-gram-Average Sampling Technique* (*NST*) [35]. It gathers statistics for fixed-size sequences of moves called N-grams. MAST-context maintains statistics for submoves of different lengths. When the iteration is over, for each move applied in the semisplit game, the statistics are updated not only for that move but also for each of its prefix. The *context* of a state is the submove from the last preceding nodal state to this state. Thus, a context is a submove that is a prefix of some move. Hence, in MAST-context, while selecting the best submove in the simulation phase, we use the statistics of the prefix of a move obtained by concatenating the current context with the semimove.

In RAVE-context, nodes store an AMAF statistics for each child just as in the regular RAVE. The difference is in updating AMAF, which updates the statistic of a semimove only if the iteration contains the same semimove played in the same context.

The context variants are available for every variant of MCTS because they store statistics for all submoves that may be needed. However, they are useless in certain variants where we do not need the statistics for semimoves. In particular, RAVE-context is not useful in the case of the orthodox design in the MCTS phases, as only the statistics for full moves are required. In this case, it is better to use RAVE-join directly.

A particular advantage of RAVE-context is that it works well in the combination with the roll-up variant. Each node stores AMAF for its context, independently of whether and what nodes are on the path to the node from its nodal predecessor. Therefore, if we remove an intermediate node and keep the AMAF value of its child, it will be the same as if the tree structure did not contain the intermediate node from the beginning.

Mix variants

The fact that, for the same MCTS variants, we have more than one choice for variants of action-based heuristics, lead to the possibility of using more of them simultaneously. The variants of the heuristics differ in the speed of gathering samples but also in their quality, e.g., MAST-split gathers samples faster than MAST-context, but they are more relevant in the latter. This leads to proposing mix variants, where we combine the context variant with the split variant.

MAST-mix combines MAST-split with MAST-context and RAVE-mix combines RAVE-split with RAVE-context. All nodes and semimoves are being updated according to both split and context strategies. For both heuristics, we have an additional parameter called the *mix-threshold*. During the evaluation, when the number (weight) of samples in the context heuristic is smaller than the mix-threshold, the score is evaluated according to the split variant. Otherwise, the context statistics are used. This also resembles NST, where statistics of n-grams are used only if the number of samples reaches a certain threshold (which is proposed to be 7 for good results) [35].

Available combinations

Table 2.1: Available combinations of the variants of MCTS and action-based heuristics.

MCTS variant		Standard		Split		Join		Context	
Tree	Sim.	MAST	RAVE	MAST	RAVE	MAST	RAVE	MAST	RAVE
orthodox	orthodox	✓^{J}	✓^{J}	✓	–	✓	✓	~~✓^{J}~~	~~✓^{J}~~
orthodox	semisplit	–	–	✓	–	–	✓	✓	~~✓^{J}~~
semisplit	orthodox	–	–	✓	✓	✓*	–	✓	✓
semisplit	semisplit	✓^{S}	✓^{S}	✓	✓	–	–	✓	✓
roll-up	orthodox	–	–	✓	–	✓*	–	✓	✓
roll-up	semisplit	–	–	✓	–	–	–	✓	✓

^{S} – equivalent to *split* variant
^{J} – equivalent to *join* variant
~~✓~~ – equivalent to another variant but less efficient than it, so useless
* – works without using MAST in the expansion phase

Table 2.1 summarizes possible variants of action-based heuristics depending on the MCTS semisplit combination. As a general rule, a variant is available if it gathers statistics required for the main MCTS phase where it is used (i.e., simulation phase for MAST and tree phase for RAVE). The standard (straightforward) variants are available only when the domain of moves is the same in both the MCTS tree phases and the simulation phase.

MAST-split can work in all cases, as it can divide a move or its factor into a

set of semimoves and supports evaluation of such a factor basing on the statistics of semimoves. RAVE-split works only when the MCTS tree uses semisplit, as in other cases it could not easily evaluate submoves that are not single semimoves (due to technical difficulties and only small benefits expected, we do not propose such a solution). MAST-join and RAVE-join can work whenever the simulation phase and the tree phase, respectively, use the orthodox design. MAST-join can work when the tree phase is not orthodox, but then it is not applied for the expansion phase. Finally, MAST-context and RAVE-context can work in all cases, as they gather all statistics that may be required. However, in some orthodox cases, they are fully equivalent to the corresponding join variants and thus are inefficient due to gathering some useless statistics (i.e., for semimoves). Of course, the mix variants are available precisely in the combinations where both the split and the context variants are available, and where the context variant is non-optimal, they can also use the join variant instead (i.e., in the case of orthodox MCTS).

Chapter 3

Implementation for Regular Boardgames

Deriving a semisplit game depends on the underlying implementation of a game reasoner, i.e., the algorithm computing legal moves or semimoves. In this chapter, we describe issues specific to implementing the semisplit algorithm for the Regular Boardgames GGP framework. We discuss the representation of a move in RBG, split strategies, used data structures, and the architecture of the agent.

3.1 Moves in Regular Boardgames

The agent uses the RBG compiler, which takes as the input a game description and produces a reasoner for it. Depending on options, the resulting reasoner provides generic functions for computing, for a given game state, the list of all legal moves and/or the list of all legal semimoves.

A move in RBG is a sequence of pairs of two integers, which are an action index and a board vertex. The second changes the current position at the board, whereas the first indicates what is changed in the game state (i.e., either the content of the current board vertex or the value of a variable). This provides a simple possibility of split: a single semimove can be just one such pair. It leads to a very effective representation of a semimove, as it is of a fixed length.

However, we may also need to test different ways of splitting. Thus, RBG supports *split points* that are put in the game description. Such split points act as additional actions, and the interface allows to effectively compute a submove ending with a split point or ending the move. Submoves computed by the reasoner can act directly as semimoves in the semisplit game tree, or they can be further split, e.g., in MAST-split.

A semisplit game is derived by modifying the reasoning algorithm [20, Theo-

rem 3], which stops at split points and/or at each action and reports a semimove. Computing shorter semimoves is generally more effective, although it is also faster to compute full moves directly instead of fusing all of them from semimoves as Alg. 1 does. The advantage of computing semimoves comes from not computing all full moves.

We note that split points are artificial actions, in the sense that they do not non-trivially modify the game state. Only the last split point in a submove has an effect, thus when we concatenate submoves, we remove all split points in the middle. Also, full moves never contain split points. This improves the efficiency but causes some minor issues when action-based heuristics are used in combined semisplit variants. For instance, in the orthodox MCTS tree design, we do not get split points, hence they are not updated by MAST-split from that part of the iteration, but still they are updated in the semisplit simulation phase.

3.2 Split strategies

Currently, RBG supports split strategies built from three components. They are algorithms taking as the input a pure definition of the game rules (without any heuristic information for good playing) thus can be considered knowledge-free. A semisplit game is derived by modifying the reasoning algorithm [20, Theorem 3] accordingly to compute semimoves instead of moves. The exact definition of split strategies is out of the scope of this thesis, thus here we describe only the intuitive meaning.

• *Mod*: This is a basic component of relatively low granularity. Every semimove corresponds to an action modifying either a single square on the board or a variable. Thus, each semimove becomes an elementary modification of the game state. A move in a chess-like game is split into two semimoves, one for selecting and grabbing a piece on the board, and one for dropping it on the destination square. For modifying more squares, more semimoves are introduced, accordingly (e.g., Amazons). Also, it separates the final decision, e.g., in Go the player first chooses whether to pass, and in Pentago it first puts a ball and then selects a board and a rotation. If a move consists of putting just one piece (e.g., Gomoku, Reversi), it is not split and a semisplit agent plays the same as the orthodox one (except minor efficiency differences). Using Mod allows assuming that semimoves are of a fixed size, which improves efficiency. An example of the resulting semisplit game tree (up to minor differences) is shown in Fig. 2.1(middle).

• *Plus*: This component introduces a semimove for every decision in the rules except iterative ones. It commonly involves choosing the direction of the movement (Amazons, Chess, English Draughts) and move type (capture or two movements in Breakthru). It does not split iterative decisions such as stopping on a square while moving in a direction (e.g., rook upwards move). An example of the resulting

semisplit game tree (up to minor differences) is shown in Fig. 2.1(right).

• *Shift*: This component also splits square selection when it consists of more than one decision. This commonly separates the selection of a column and a row on the board (all games) and also divides movements of some pieces (e.g., knight – long and short hops).

From the components, we can build split strategies such as *Mod*, *ModShift* or *ModPlusShift*. The exact details of a particular split strategy depend, of course, on the particular game description. The strategies can also introduce indecisive semimoves, which do not alter the behavior of MCTS but affect efficiency, either positively or negatively.

3.3 Data structures for action-based heuristics

MAST stores global statistics for each move. Updating and, in particular, reading those statistics should be as fast as possible to avoid the situation in which the time overhead from using MAST causes losings greater than benefits.

Probably the best general choice is a dictionary based on a hashtable. We use a custom implementation of a hashtable with closed hashing, taking into account its global character (adjusting the initial size and growth) and the domain of the integers in actions.

However, in particular in the case of MAST-split, when we know that the domain of all possible semimoves is small, it can be replaced with a raw array.

MAST-context could be implemented with the same hashtable as is used for full moves. However, it can be optimized, taking advantage of our specific application. We use the fact that whenever a submove is stored in the hashtable, also all its prefixes must have been already stored there since they have been applied earlier. Furthermore, we have queried these prefixes right before processing the submove. Thus, we instead of the full submove, we store pairs of the context and the semimove. And the context is represented by the bucket id in the hashtable, which uniquely represents a stored submove and is returned directly from the preceding query. In this way, elements in the hashtable have a fixed length, and their hashing and comparisons (in the case of conflicts) are cheaper.

RAVE uses similar structures, but it is enough to use a hash set instead of a hash map, as we only need to query if a given move occurs at the bottom of the iteration. In the case of RAVE-split, we use bit sets to encode this, taking advantage of a relatively small domain of semimoves.

3.4 Architecture of the agent

The main part of the agent is written in C++17. Additionally, there is a Python script that maintains the preparation and compilation of the main code.

The agent is run with a given configuration file that specifies the variant and all the parameters of the algorithm. The procedure of preparing the agent for playing a match is divided into several steps.

1. **Connecting to game manager**: At the beginning, the agent's running script (in Python) connects with the game manager and parses the chosen configuration file.
2. **Receiving game rules**: When a sufficient number of agents are connected, the game manager assigns roles to them, sends the time limit for preparation, and sends the game rules.
3. **Applying a split strategy**: Based on the configuration file, the agent decides how the reasoner should be generated, i.e., which split strategy and what compiler flags should be used. Split strategies are applied by the agent by calling an external utility to insert split points, before compiling the given game description. It modifies the original game rules sent by the game manager and creates a description of an equivalent semisplit game.
4. **Compiling the game**: The reasoner is generated by the RBG compiler on the basis of the (modified) description of the game rules.
5. **Customizing the agent code**: A C++ header file is generated based on the values of the parameters from the configuration file. This file consists of some `#define` directives, `constexpr` variables, and type definitions. These values are used for the conditional compilation of the main code. One of the advantages of just-in-time compilation is that many conditional jumps can be avoided when values of some parameters are known at compilation time. Moreover, some data structures (e.g., an array for storing statistics in MAST-split) need to know their initial size at compilation time. These values are known right after the reasoner is generated.
6. **Compiling the agent**: On the basis of configuration file, a proper `make` target and `gcc` flags are deduced and relevant command is executed. Each legal variant of MCTS combined with any suitable set of action-based heuristics has its own make target. For these targets, different subsets of files with source code are assigned to determine which tree design, data structures, simulation policy, etc. should be used.
7. **Running and connecting the main agent**: The script runs the compiled agent, which immediately connects to the script via a local connection. Then, the *ready* status is sent to the game manager.

8. **Playing the game**: After receiving this status from each agent, the game manager starts a match by sending information with the time limit for the first turn of the first player. From this point, the agent's script acts as a link between the manager and the script, transferring data between them.

Chapter 4

User guide

Besides the source code of the agent, there are other components that are needed to perform a match between agents. The agent must connect to the game manager and use the RBG compiler to generate reasoners. The experimental environment requires RBG modules `rbgGames`, `rbg2cpp`, and `rbggamemanager`. The complete setup for experiments, including the source code of the agent itself, is available at `https://github.com/WoojtekP/rbgPlayer-experiments` and can be easily downloaded using the following commands:

```
$ git clone --recursive https://github.com/WoojtekP/rbgPlayer-experiments.git
$ cd rbgPlayer-experiments
$ git checkout BSthesis
```

Then, by running script `prepare.sh`, all components should be ready for experiments.

4.1 Configuration file

The agent is configured through files in JSON format. A configuration file can contain the following parameters:

- `general` : *object* – section to describe parameters common for all agents.
  - `buffer_time` : *integer* – time in milliseconds reserved by the agent for communication with game manager and for choosing move to send.
  - `simulate_during_opp_turn` : *bool* – set to true, if iterations of MCTS should be performed also during opponents' turns.
  - `reasoning_overhead` : *integer* – multiplier for reasoning overhead. Values greater than 1 are used to handicap an agent or for experimental purposes. The parameter multiplies the cost of computing legal (semi)moves and game states.

- `algorithm` : *object* – section to describe the design used in particular phases of MCTS.
  - `name` : *string* – only MCTS is available.
  - `tree_strategy` : *string* – the design for the selection and expansion phases.
  - `simulation_strategy` : *string* – the design for the simulation phase.
  - `split_strategy`^{S} : *string* – the name of the applied split strategy.
  - `parameters` : *object* – general parameters used to tune MCTS:
    - `exploration_constant` : *float* – the exploration constant of UCT.
    - `max_semidepth`^{S} : *integer* – the maximal possible search depth for nodal state.
    - `is_nodal`^{S} : *bool* – set to `true` for the *nodal* variant; `false` denotes the *raw* variant.
    - `min_simulations_factor`^{R} : *float* – the *MSF* parameter used in the *rollup* variant.
- `heuristics` : *object []* – an array of objects which describes details of heuristics.
  - `name` : *string* – the name of the heuristic variant.
  - `parameters` : *object* – parameters used to tune the details.
    - `epsilon`^{M} : *float* – the value of ε used in the ϵ-greedy policy.
    - `decay_factor`^{M} : *float* – the value of the *decay factor* used in the move decay strategy.
    - `tree_only`^{M} : *bool* – set to `true`, if statistics should be updated only for (sub)moves applied in the selection phase (*tree-only MAST*).
    - `equivalence_parameter`^{R} : *integer* – tunes the balance of the average score and AMAF score used in the modified UCB formula.
    - `ref`^{G} : *integer* – the minimal number of simulations which is needed to consider AMAF scores for a given (sub)move. If non-zero, this is used in experimental TGRAVE heuristic, which is not used and not described in this thesis.

^{S} – enabled for *split* or *rollup* desing in any phase
^{R} – enabled for *rollup* design
^{M} – enabled for all MAST variants
^{R} – enabled for all RAVE variants
^{G} – enabled for TGRAVE

4.2 Software requirements

Before compiling source code, one has to install:

- GNU make
- gcc (recommended version $\geq$ 7.5.0)
- boost (recommended version $\geq$ 1.67.0)
- python3 with scipy (recommended version $\geq$ 3.5.2)

4.3 Run experiments

To play out a tournament between two players, use script `run_matches.sh` as follow:

```
$ ./run_matches.sh game player1 player2 boundtype bound plays
```

where:

- *game* – name of the game
- *player1, player2* – names of configuration files placed in `rbgPlayer/agents` directory, without `.json` suffix
- *boundtype* – one of:
  - `t` – time limit given in milliseconds
  - `m` – simulations limit
  - `s` – states limit
  - `fm0, fs0, fsOMR` - this type of bound uses suitable number of simulations or states for specific games to run matches in which orthodox mcts player (pure or with mast and rave) has approximately *bound* milliseconds per turn
- *bound* – the value of limit of chosen type
- *plays* – number of pairs of plays (one with swapped agents) per thread

Example:

```
$ ./run_matches.sh breakthrough mcts_orthodox_orthodox_mast_rave \
  mctsM_semisplitNodal_semisplit_mastsplit_rave fsOMR 500 10
```

Logs are collected in `logs` directory. A script `logreader.py` can be used to print logs in a readable format as follow:

```
$ python logreader.py logs
```

Another way to run matches among players is to run the game manager and agents as separate processes. It gives an opportunity for tracking moves chosen by agents on the fly, inspecting the number of simulations performed by agents in each turn, checking statistics stored for moves, and easier debugging. From `rbggamemanager` directory:

```
$ ./build/start_server game port [flags]
```

where:

- *game* – a path to file with description of the game rules in Regular Boardgames
- *port* – port at which game manager connects with agents
- *flags* – optional flags with their values:
  - `deadline` – time limit in milliseconds to make a move in each turn.
  - `log_results` – *stdout* or path to file in which scores should be stored.
  - `log_move` – *stdout* or path to file in which performed moves should be stored.
  - `limit` – the number of matches to play. Note that a single run of the game manager does not alternate players.

Example:

```
$ ./build/start_server ../rbgGames/games/amazons.rbg 7784 --deadline 500 \
  --limit 10 --log_moves moves.txt --log_results stdout
```

Then, a sufficient number of agent processes must be run. From `rbgPlayer` directory:

```
$ python play.py server-address server-port config-file [flags]
```

where:

- *server-address* - IP address of the game manager.
- *server-port* - the port of the game manager.
- *flags* – optional flags with their values:

- `simulations-limit` – the limit of simulations for each turn.
- `states-limit` – the limit of states for each turn.
- `release` – run with maximal efficiency, but without debug traps.
- `states` – print statistics for legal moves after each turn.
- `debug` – run agent's process with `valgrind` tool.

Using a simulations or states limit causes ignoring the time limit from the game manager. Flags `release`, `states` and `debug` do not take any numeric values. Example:

```
$ python play.py localhost 7784 agents/mcts_orthodox_semisplit.json \
  --simulations-limit 1000 --release
```

Chapter 5

Experiments

We show experimental results from comparing agents. Here, by an *agent*, we mean a particular configuration of our generic agent.

5.1 Technical setup

5.1.1 Parameters

All parameters of the tested agents were turned according to the recommendations in the literature. Both orthodox and semisplit agents were using exactly the same parameters set. This means that they may be not necessarily tuned for semisplit agents, as they are originally recommended for orthodox agents in the literature.

The exploration constant in the UCT formula [33, (1)] was set to $C = 0.4$ [7]. We have also tried the second popular value $C = 0.7$, but it turned out to give worse results than $C = 0.4$ in our setting and game set.

MAST uses the ε-greedy policy with $\varepsilon = 0.4$, i.e., with probability 0.6 it chooses a (sub)move with the best average score. The decaying parameter was set to 0.2 [34].

In the case of MAST-mix, the mix-threshold was set to 7, as this value was proposed for NST [35].

RAVE had its equivalence parameter set to 250 [33]. According to recommendations, whenever RAVE was used (in all agent types), the exploration constant was also set to $C = 0.2$.

The roll-up MSF parameter was set to 1.0, and the final roll-up selection was not used, unless otherwise indicated. Finally, an agent does not compute (sleeps) during the opponent's turn.

5.1.2 The setting

There are 240 plays in each test, where 120 plays are played with swapped agents. The final win ratio is computed by assigning 1 point for a win, 0.5 for a draw, and 0 for a loss. Win ratios are given in percents.

The confidence intervals are given for 95% confidence based using a standard method [14]. We say that an agent has *better performance in a game* if the average result is above 50% and 50% is outside the confidence interval. Symmetrically, it has *worse performance* if the average result is below 50% and it is outside the confidence interval. If 50% lies within the confidence interval, an agent has *similar performance*. Games from the test set where an algorithm has better, similar, and worse performance are counted and given as an additional statistic besides the total average win ratio. These are indicated in the form *better*:*similar*:*worse*.

In the timed experiments (*timed setting*), the time limit is set to 0.5s per turn, unless otherwise stated. The buffer time is not included, i.e., this is the time limit for pure computation; the real limit sent by the manager is $0.6s$. In view of computational power, this roughly corresponds to the setting with 10s limit based on GDL reasoning [19], which is often used [33].

There are also experiments with a fixed budget limit (*fixed-states setting*). We limit the number of states that an agent can compute in its own turn. Only nodal states are counted. The limits for each game are computed by benchmarking the corresponding orthodox agent, measuring the number of states computed in the first turn over 10s. In contrast with limiting the number of simulations, this better corresponds to the timed setting, because of two reasons. First, the average length of simulations may vary depending on whether the semisplit or orthodox design is used. Second, the average length of simulations decreases as a game play advances, since terminal states are closer.

5.1.3 Environment

The hardware used for the timed experiments was a grid belonging to the Institute of Computer Science, University of Wrocław. 4 computer were used, each with Intel(R) Core(TM) i7-4930K CPU @ 3.40GHz (6 cores) and between 16 and 32 GBi. The system was `Ubuntu 16.04.5 LTS (GNU/Linux 4.13.0-45-generic x86_64)`. The relevant software was `gcc 10.1.0`, `boost 1.67`, and `Python 3.5.2`.

The tests of agents were performed with 6 plays in parallel (one process computes 20 plays, giving 120 in total).

Because the experiments with fixed states limit do not depend on hardware efficiency (i.e., are deterministic up to random seed), they were performed on other computers.

5.1.4 Agents

Semisplit agents were tested against the orthodox ones: both pure MCTS agent (denoted by $\mathbb{O}$) and the agent using MAST and RAVE ($\mathbb{O}^{\text{tree:RAVE}}_{\text{sim:MAST}}$) were used as baselines.

Our semisplit agents (i.e., using some split strategy; non-orthodox) are denoted by $\mathbb{S}$ with indices describing the variant: "S" means the semisplit design and "O" means the orthodox design, which can be independently used in the MCTS tree or in simulations; "R" is the roll-up variant. There is also an indication if it is the raw or nodal variant and if MAST and/or RAVE is used.

5.1.5 Game test set

Our test set consists of 15 games: Amazons, Breakthrough, Breakthru, Chess, Chess-no check (i.e., Chess with king capture), English Draughts, Fox And Hounds, Go, Gomoku, Hex, Knightthrough, Pentago, Reversi, Skirmish, and The Mill Game. They are implemented in RBG and their codes can be found in `rbgGames` submodule under the corresponding names.

We note that Gomoku, Hex, and Reversi are not split under Mod strategy nor ModPlus strategy. Nevertheless, they are included in all tests for consistency and comparability of the results. Also, while both orthodox and semisplit do the same computation in these cases, they do not achieve the same performance, in particular, due to a different (sub)move representation.

5.2 Results

In this section, we show the experimental results. Because of a vast number of combinations to test, we focus on a selected subset of configurations.

5.2.1 Efficiency of agents

Table 5.1 demonstrates the efficiency differences between basic orthodox and semisplit agents, for a selection of games and Mod strategy. The results come from the 10s first-turn benchmark, i.e., measuring the number of states per second and the maximum used memory during the first turn with 10s time limit. We can observe significant speed-up in all cases. Also, the used memory stays relatively low, and it is comparable in both agent types, often being lower in the semisplit agent.

Table 5.1: A comparison of the efficiency (in time and memory) of basic orthodox and semisplit agents.

Game	$\mathbb{O}$		$\mathbb{S}^{\text{tree:S-nodal}}_{\text{sim:S}}$	
	States/sec.	Memory (MBi)	Speed-up	Memory (MBi)
Amazons *Mod*	158,760	258	1,417 %	265
Breakthrough *Mod*	1,885,488	79	222 %	74
Breakthru *Mod*	6,392	6	26,786 %	20
Chess *Mod*	211,225	6	580 %	13
Chess-no check *Mod*	562,180	21	405 %	49
English Draughts *Mod*	2,886,068	150	134 %	83
Knightthrough *Mod*	1,500,851	261	295 %	164
The Mill Game *Mod*	1,630,673	132	252 %	150

5.2.2 Basic agents

Table 5.2 shows the average results of selected variants, together with information about the number of games of better and worse performance. The results from the timed setting are associated with those from the fixed setting. The latter shows the results of semisplit agents apart from efficiency benefits.

We observe a large advantage of semisplit agents in most variants. Major benefits come from the speed, as in the fixed setting the results are noticeably worse, though still slightly above 50% in most cases. The Mod strategy is the best, and ModShift is also good. Larger-granularity split strategies with the Plus component give worse results, but this is due to performance overhead. In particular, altering the game tree (in a blind way) by itself is not harmful on average, which is good information supporting the semisplit algorithm, which usually improves the speed.

The best pure MCTS agents are $\mathbb{S}^{\text{tree:S-raw}}_{\text{sim:S}}$ and $\mathbb{S}^{\text{tree:S-nodal}}_{\text{sim:S}}$ with the Mod strategy, whose results are very similar. In general, we consider the nodal variant slightly better, taking into account the results for other split strategies, and because if the raw variant has a small advantage, it is only due to its little better efficiency. Another good agent is $\mathbb{S}^{\text{tree:O}}_{\text{sim:S}}$, which takes the advantage just from the improved speed of the simulation phase. For most of games it has a similar performance to the orthodox agent, thus it can be considered as a safe choice.

Table 5.3 shows the results of selected variants of semisplit algorithms on a selected subset of games. The simulation (flat MC) speed-up is also given. We can observe how strongly the results vary depending on the game. For some games, the semisplit design is inherently better; for others, it is worse than the orthodox design. Surprising results were obtained for Chess and Chess-no check – while strategically

Table 5.2: The average win ratios of semisplit variants without heuristics over the whole test set together with the numbers of games of better:similar:worse performance.

Agent		Timed setting	Fixed setting
		vs. $\mathbb{O}$	
$\mathbb{S}^{\text{tree:S-raw}}_{\text{sim:S}}$	*Mod*	**72.17** (11:2:2)	56.82 (6:5:4)
$\mathbb{S}^{\text{tree:S-nodal}}_{\text{sim:S}}$	*Mod*	71.39 (10:3:2)	**57.39 (7:3:5)**
$\mathbb{S}^{\text{tree:O}}_{\text{sim:S}}$	*Mod*	67.66 (12:2:1)	50.97 (4:9:2)
$\mathbb{S}^{\text{tree:S-nodal}}_{\text{sim:O}}$	*Mod*	58.21 (6:7:2)	56.61 (6:6:3)
$\mathbb{S}^{\text{tree:R-nodal}}_{\text{sim:S}}$	*Mod*	67.01 (10:3:2)	51.75 (4:7:4)
$\mathbb{S}^{\text{tree:R-nodal}}_{\text{sim:O}}$	*Mod*	51.70 (2:10:3)	52.09 (2:12:1)
$\mathbb{S}^{\text{tree:S-raw}}_{\text{sim:S}}$	*ModPlus*	62.89 (9:2:4)	57.42 (7:4:4)
$\mathbb{S}^{\text{tree:S-nodal}}_{\text{sim:S}}$	*ModPlus*	**65.73** (9:3:3)	**58.80** (7:4:4)
$\mathbb{S}^{\text{tree:O}}_{\text{sim:S}}$	*ModPlus*	60.05 (9:2:4)	51.58 (4:9:2)
$\mathbb{S}^{\text{tree:S-nodal}}_{\text{sim:O}}$	*ModPlus*	57.37 (6:7:2)	57.62 (6:7:2)
$\mathbb{S}^{\text{tree:R-nodal}}_{\text{sim:S}}$	*ModPlus*	57.99 (8:3:4)	53.15 (4:9:2)
$\mathbb{S}^{\text{tree:R-nodal}}_{\text{sim:O}}$	*ModPlus*	51.19 (1:13:1)	51.63 (2:12:1)
$\mathbb{S}^{\text{tree:S-raw}}_{\text{sim:S}}$	*ModShift*	66.23 (10:1:4)	54.64 (7:3:5)
$\mathbb{S}^{\text{tree:S-nodal}}_{\text{sim:S}}$	*ModShift*	65.56 (10:1:4)	54.39 (7:3:5)
$\mathbb{S}^{\text{tree:O}}_{\text{sim:S}}$	*ModShift*	**69.04** (11:3:1)	49.54 (4:6:5)
$\mathbb{S}^{\text{tree:S-nodal}}_{\text{sim:O}}$	*ModShift*	53.63 (8:3:4)	**56.22** (8:4:3)
$\mathbb{S}^{\text{tree:R-nodal}}_{\text{sim:S}}$	*ModShift*	65.45 (10:2:3)	48.49 (4:6:5)
$\mathbb{S}^{\text{tree:R-nodal}}_{\text{sim:O}}$	*ModShift*	47.25 (2:8:5)	50.03 (3:7:5)
$\mathbb{S}^{\text{tree:S-raw}}_{\text{sim:S}}$	*ModPlusShift*	58.42 (9:0:6)	51.75 (8:2:5)
$\mathbb{S}^{\text{tree:S-nodal}}_{\text{sim:S}}$	*ModPlusShift*	59.29 (9:1:5)	54.11 (8:2:5)
$\mathbb{S}^{\text{tree:O}}_{\text{sim:S}}$	*ModPlusShift*	**59.97** (9:1:5)	50.40 (4:9:2)
$\mathbb{S}^{\text{tree:S-nodal}}_{\text{sim:O}}$	*ModPlusShift*	53.65 (6:5:4)	**54.38** (8:3:4)
$\mathbb{S}^{\text{tree:R-nodal}}_{\text{sim:S}}$	*ModPlusShift*	57.15 (9:1:5)	48.98 (7:3:5)
$\mathbb{S}^{\text{tree:R-nodal}}_{\text{sim:O}}$	*ModPlusShift*	48.39 (2:10:3)	48.92 (2:10:3)

almost identical (the only real difference is the lack of stalemate), the former is friendly for semisplit and the latter is hard.

An important variant is $\mathbb{S}^{\text{tree:R-nodal}}_{\text{sim:O}}$; it restricts the semisplit design to the expansion phase, yielding a similar effect to *unprunning* methods [5], designed to deal with a large branching factor. Indeed, $\mathbb{S}^{\text{tree:R-nodal}}_{\text{sim:O}}$ is strong in Breakthru (which has a very large branching factor), yet similar in almost all other games. This concludes

Table 5.3: Comparison of orthodox and semisplit agents for a selection of games under the timed setting with 0.5s per move. 95% confidence intervals are shown. The cases counted as of better performance are marked bold.

Game	$\mathbb{S}^{\text{tree:S-nodal}}_{\text{sim:S}}$	$\mathbb{S}^{\text{tree:O}}_{\text{sim:S}}$	$\mathbb{S}^{\text{tree:S-nodal}}_{\text{sim:O}}$	$\mathbb{S}^{\text{tree:R-nodal}}_{\text{sim:O}}$
	vs. $\mathbb{O}$			
Amazons *Mod*	**98.3**±1.6	**98.3**±1.6	**78.8**±5.2	55.0±6.3
Breakthrough *Mod*	55.4±6.3	**62.5**±6.2	50.8±6.4	50.0±6.4
Breakthru *Mod*	**100**	**68.8**±5.9	**97.1**±2.1	**97.1**±2.1
Chess *Mod*	**96.0**±2.3	**95.4**±2.5	**58.5**±5.9	40.6±5.9
Chess *ModShift*	**96.9**±2.1	**98.3**±1.5	**62.5**±5.8	44.4±6.0
Chess *ModPlusShift*	**70.8**±5.6	**65.6**±5.6	**72.5**±5.3	45.6±6.1
Chess-no check *Mod*	53.5±5.9	**63.1**±4.9	32.7±5.6	20.2±4.7
English Draughts *Mod*	45.4±3.9	54.8±3.6	43.1±3.6	52.9±4.0
Gomoku *ModShift*	**97.9**±1.8	**92.9**±3.3	**67.9**±5.9	48.8±6.4
Knightthrough *Mod*	**91.2**±3.6	**80.0**±5.1	**72.5**±5.7	51.7±6.4
The Mill Game *Mod*	39.4±5.6	**44.0**±5.4	51.2±5.4	54.0±5.4

that the semisplit's effect on expansion alone is minor, yet it could be an alternative and a quite safe method to overcome the large branching factor difficulty.

Fig. 5.1 shows how the win ratio changes for different time limits and in relation to the fixed-states setting. Here, a general observation is that the results are kept consistent for different time limits, thus the semisplit algorithm should generalize well in this perspective.

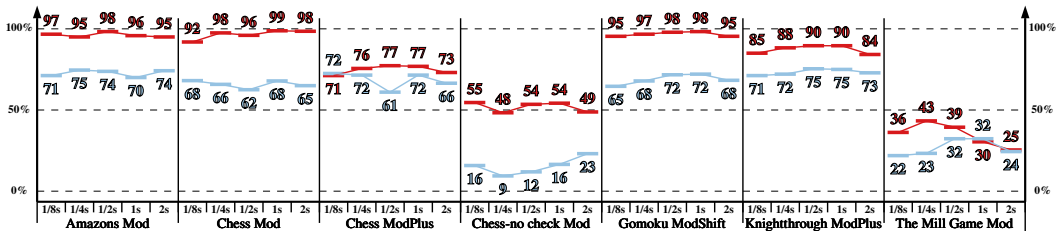

Figure 5.1: The results of $\mathbb{S}^{\text{tree:S-nodal}}_{\text{sim:S}}$ versus $\mathbb{O}$ for different time limits (red scores) and for equivalent states budgets (blue scores).

5.2.3 Agents with heuristics

Concerning the MCTS with action-based heuristics, we focus on the *Mod* strategy, as it appears generally the best. As there are many available combinations, we also

test only a selected subset of them. We test agents obtained by adding heuristics to $\mathbb{S}^{\text{tree:S-nodal}}_{\text{sim:S}}$ and $\mathbb{S}^{\text{tree:O}}_{\text{sim:S}}$, which we consider to be most promising, based on previous tests.

Table 5.4 shows the average results and Table 5.3 shows particular results for a selected subset of games and two best agents. The average results are divided into two groups: split/join variants and context variants. It seems that the best variants are $\mathbb{S}^{\text{tree:S-nodal}}_{\text{sim:S,MAST}}$ and $\mathbb{S}^{\text{tree:O,RAVE-join}}_{\text{sim:S,MAST-split}}$. Surprisingly, the former does not use RAVE. Apparently, RAVE does not combine so well with semimoves used in the MCTS tree (cf. $\mathbb{S}^{\text{tree:S-nodal,RAVE}}_{\text{sim:S,MAST}}$). However, orthodox-tree and semisplit-simulation agents show their additional advantage here, since RAVE works in them as in the orthodox agent. They use RAVE on full moves, taking at the same time efficiency benefits from semisplit simulations. Looking at particular games, $\mathbb{S}^{\text{tree:O,RAVE-join}}_{\text{sim:S,MAST-split}}$ has worse performance only in The Mill Game (which is an especially hard case for all semisplit variants). Additionally, $\mathbb{S}^{\text{tree:S-nodal}}_{\text{sim:S,MAST-mix7}}$ gives the best results in the fixed setting, but its efficiency overhead does not let to outperform $\mathbb{S}^{\text{tree:S-nodal}}_{\text{sim:S,MAST}}$.

Table 5.4: The average win ratios of selected semisplit variants with heuristics over the whole test set together with the numbers of games of better:similar:worse performance.

Agent		Timed setting	Fixed setting
		vs. $\mathbb{O}^{\text{tree:RAVE}}_{\text{sim:MAST}}$	
$\mathbb{S}^{\text{tree:S-nodal,RAVE}}_{\text{sim:S,MAST}}$	*Mod*	52.73 (7:3:5)	43.97 (5:4:6)
$\mathbb{S}^{\text{tree:O,RAVE-join}}_{\text{sim:S,MAST-split}}$	*Mod*	63.41 (11:3:1)	51.21 (4:8:3)
$\mathbb{S}^{\text{tree:S-nodal}}_{\text{sim:S,MAST}}$	*Mod*	**67.52** (10:3:2)	**54.11** (9:2:4)
$\mathbb{S}^{\text{tree:O}}_{\text{sim:S,MAST-split}}$	*Mod*	62.96 (9:1:5)	45.42 (3:5:7)
$\mathbb{S}^{\text{tree:S-nodal}}_{\text{sim:S}}$	*Mod*	63.54 (10:1:4)	40.63 (5:0:10)
$\mathbb{S}^{\text{tree:O}}_{\text{sim:S}}$	*Mod*	58.77 (9:1:5)	31.01 (2:0:13)
$\mathbb{S}^{\text{tree:S-nodal,RAVE-context}}_{\text{sim:S,MAST-context}}$	*Mod*	41.95 (6:3:6)	37.00 (2:6:7)
$\mathbb{S}^{\text{tree:S-nodal}}_{\text{sim:S,MAST-context}}$	*Mod*	61.60 (10:1:4)	51.37 (7:3:5)
$\mathbb{S}^{\text{tree:S-nodal}}_{\text{sim:S,MAST-mix7}}$	*Mod*	**64.13** (11:2:2)	**55.21** (4:8:3)
$\mathbb{S}^{\text{tree:O,RAVE-join}}_{\text{sim:S,MAST-mix7}}$	*Mod*	58.15 (7:7:1)	52.19 (4:8:3)
$\mathbb{S}^{\text{tree:S-nodal,RAVE-context}}_{\text{sim:S}}$	*Mod*	43.38 (6:1:8)	29.69 (2:2:11)

5.2.4 Additional features

Finally, we present preliminary results concerning two specific variations. Here, Gomoku, Hex and Reversi were removed from the test set, as in these cases the agents in the fixed setting work, in principle, identically; thus we have 12 games.

Table 5.5: Comparison of orthodox and selected semisplit agents with action-based heuristics for a selection of games under the timed setting with 0.5s per move. 95% confidence intervals are shown. The cases counted as of better performance are marked bold.

Game	$\mathbb{S}^{\text{tree:O,RAVE-join}}_{\text{sim:S,MAST-split}}$	$\mathbb{S}^{\text{tree:S-nodal}}_{\text{sim:S,MAST}}$
	vs. $\mathbb{O}^{\text{tree:RAVE}}_{\text{sim:MAST}}$	
Amazons *Mod*	**76.7**±5.4	**89.2**±4.0
Breakthrough *Mod*	51.7±6.4	**71.7**±5.7
Breakthru *Mod*	**100.0**	**100.0**
Chess *Mod*	**98.1**±1.7	**95.4**±2.5
Chess *ModShift*	**96.9**±2.2	**97.3**±1.9
Chess *ModPlusShift*	24.6±5.1	19.8±4.8
Chess-no check *Mod*	54.8±5.6	50.2±6.0
English Draughts *Mod*	51.0±4.2	**60.4**±4.3
Gomoku *ModShift*	**65.4**±6.1	43.8±6.3
Knightthrough *Mod*	**64.2**±6.1	**85.4**±4.5
The Mill Game *Mod*	28.3±4.3	18.8±4.1

Table 5.6 shows the results of the orthodox agent with MAST-mix (which mixes MAST-split samples with MAST-join). It seems that splitting can give noticeable benefits even in the orthodox design. The results are best for $\mathbb{S}^{\text{tree:O}}_{\text{sim:O,MAST-mix7}}$, giving an overall improvement of 5%. Importantly, it is not due to using MAST-split itself, but only due to mixing split samples with full move samples.

Table 5.6: The average win ratios of agents with the orthodox design combined with MAST-split and MAST-mix over the set of 12 games together with the numbers of games of better:similar:worse performance.

Agent		Fixed setting
		vs. $\mathbb{O}_{\text{sim:MAST}}$
$\mathbb{S}^{\text{tree:O}}_{\text{sim:O,MAST-split}}$	*Mod*	49.30 (2:8:5)
$\mathbb{S}^{\text{tree:O}}_{\text{sim:O,MAST-mix3}}$	*Mod*	53.97 (3:11:1)
$\mathbb{S}^{\text{tree:O}}_{\text{sim:O,MAST-mix7}}$	*Mod*	**55.22** (2:13:0)
$\mathbb{S}^{\text{tree:O}}_{\text{sim:O,MAST-mix15}}$	*Mod*	54.04 (3:12:0)
$\mathbb{S}^{\text{tree:O}}_{\text{sim:O,MAST-mix30}}$	*Mod*	52.59 (3:11:1)

Table 5.7 shows the results of $\mathbb{S}_{\text{sim:S}}^{\text{tree:S-nodal}}$ with different values of FRF parameter. The differences in the results are very little and more tests are required to find out if the final roll-up selection can slightly improve the results.

Table 5.7: The average win ratios of $\mathbb{S}_{\text{sim:S}}^{\text{tree:S-nodal}}$ agents with the final roll-up selection over the set of 12 games together with the numbers of games of better:similar:worse performance.

Agent		Fixed setting
		vs. $\mathbb{O}$
$\mathbb{S}_{\text{sim:S}}^{\text{tree:S-nodal}}$ (baseline)	*Mod*	59.42 (7:0:5)
$\mathbb{S}_{\text{sim:S}}^{\text{tree:S-nodal,FinalRollUp0.5}}$	*Mod*	58.87 (6:1:5)
$\mathbb{S}_{\text{sim:S}}^{\text{tree:S-nodal,FinalRollUp0.7}}$	*Mod*	58.98 (6:2:4)
$\mathbb{S}_{\text{sim:S}}^{\text{tree:S-nodal,FinalRollUp0.8}}$	*Mod*	57.85 (6:3:3)
$\mathbb{S}_{\text{sim:S}}^{\text{tree:S-nodal,FinalRollUp0.9}}$	*Mod*	59.16 (6:2:4)
$\mathbb{S}_{\text{sim:S}}^{\text{tree:S-nodal,FinalRollUp1.0}}$	*Mod*	59.00 (6:3:3)
$\mathbb{S}_{\text{sim:S}}^{\text{tree:S-nodal,FinalRollUp1.1}}$	*Mod*	**59.81** (6:3:3)

Chapter 6

Conclusions

We have introduced a family of Monte-Carlo Tree Search variants that work on semimoves – arbitrarily split game moves. The algorithm has been implemented in a General Game Playing agent for the Regular Boardgames GGP system. The agent supports many variations that allow comparing different approaches of using split moves.

From the current experiments, the impact of using semisplit has been revealed to be generally beneficial. First, it greatly improves search efficiency (a few times more on average, up to even 31 times more states/sec.). Moreover, for many games, the playing strength of a split-based MCTS agent is improved over its orthodox counterpart, even when both algorithms are capped to the same performance. Even using general and blind split strategies, we were able to obtain win rates higher than 75% on about half of the test set (e.g., with $\mathbb{S}^{\text{tree:S-nodal}}_{\text{sim:S}}$ and $\mathbb{S}^{\text{tree:S-nodal}}_{\text{sim:S,MAST}}$). Of course, we have tested only a selected subset of variations of the semisplit algorithm from a vast number of combinations. Thus, the results give just an initial insight. The algorithm needs more investigation, which, however, requires more long-lasting experiments.

This work opens a new way of improving game-playing algorithms, thus can be seen as pioneering. It is widely applicable and there are many directions for future research, e.g.:

- Action-based heuristics can be better adapted to semisplit in place of their straightforward application. There are also other MCTS enhancements to be combined with. Moreover, the used parameters were the same for both agent types and were tuned rather for orthodox agents. This indicates that after a tuning, semisplit should achieve even better results.
- There should be developed heuristic methods for choosing the most suitable semisplit variant for a given game.
- Semisplit can be also combined with prior knowledge [10], which could com-

pensate potential weakness but keep the efficiency. In particular, combining with self-learning approaches is an interesting opportunity, but challenging due to the altered domains of moves.

Bibliography

[1] Hendrik Baier and Mark H. M. Winands. MCTS-minimax hybrids with state evaluations. *JAIR*, 62:193–231, 2018.

[2] C. B. Browne, E Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE TCIAIG*, 4(1):1–43, 2012.

[3] M. Campbell, A. J. Hoane, and F. Hsu. Deep Blue. *Artificial intelligence*, 134(1):57–83, 2002.

[4] Tristan Cazenave. Generalized rapid action value estimation. In *IJCAI*, pages 754–760, 2015.

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

[6] Benjamin E Childs, James H Brodeur, and Levente Kocsis. Transpositions and move groups in Monte Carlo tree search. In *IEEE Symposium On Computational Intelligence and Games*, pages 389–395, 2008.

[7] H. Finnsson and Y. Björnsson. Learning Simulation Control in General Game Playing Agents. In *AAAI*, pages 954–959, 2010.

[8] Hilmar Finnsson and Yngvi Björnsson. Simulation-based approach to general game playing. In *AAAI*, volume 8, pages 259–264, 2008.

[9] David Fotland. Building a World-Champion Arimaa Program. In *Computers and Games*, volume 3846 of *LNCS*, pages 175–186. 2006.

[10] Sylvain Gelly and David Silver. Combining online and offline knowledge in UCT. In *Proceedings of the 24th International Conference on Machine Learning*, pages 273–280, 2007.

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

[12] M. Genesereth, N. Love, and B. Pell. General Game Playing: Overview of the AAAI Competition. *AI Magazine*, 26:62–72, 2005.

[13] Adrian Goldwaser and Michael Thielscher. Deep Reinforcement Learning for General Game Playing. In *AAAI*, 2020.

[14] G Mc C Haworth. Self-play: statistical significance. *ICGA journal*, 26(2):115–118, 2003.

[15] Niels Justesen, Tobias Mahlmann, Sebastian Risi, and Julian Togelius. Playing Multiaction Adversarial Games: Online Evolutionary Planning Versus Tree Search. *IEEE Transactions on Games*, 10(3):281–291, 2017.

[16] Ł. Kaiser and Ł. Stafiniak. First-Order Logic with Counting for General Game Playing. In *AAAI*, pages 791–796, 2011.

[17] J. Kloetzer, H. Iida, and B. Bouzy. The Monte-Carlo approach in Amazons. In *Proceedings of the Computer Games Workshop*, pages 185–192, 2007.

[18] L. Kocsis and C. Szepesvári. Bandit Based Monte-carlo Planning. In *European Conference on Machine Learning*, pages 282–293, 2006.

[19] J. Kowalski, R. Miernik, M. Mika, W. Pawlik, J. Sutowicz, M. Szykuła, and A. Tkaczyk. Efficient Reasoning in Regular Boardgames. In *IEEE Conference on Games*, pages 455–462, 2020.

[20] J. Kowalski, M. Mika, J. Sutowicz, and M. Szykuła. Regular Boardgames. In *AAAI*, pages 1699–1706, 2019. `https://arxiv.org/abs/1706.02462`.

[21] N. Love, T. Hinrichs, D. Haley, E. Schkufza, and M. Genesereth. General Game Playing: Game Description Language Specification. Technical report, Stanford Logic Group, 2006.

[22] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[23] Santiago Ontañón. Combinatorial multi-armed bandits for real-time strategy games. *J. Artif. Intell. Res.*, 58:665–702, 2017.

[24] D. Perez, S. Samothrakis, J. Togelius, T. Schaul, and S. M. Lucas. General Video Game AI: Competition, Challenges and Opportunities. In *AAAI*, pages 4335–4337, 2016.

[25] Diego Perez, Spyridon Samothrakis, Simon Lucas, and Philipp Rohlfshagen. Rolling horizon evolution versus tree search for navigation in single-player real-time games. In *GECCO*, pages 351–358, 2013.

[26] E. Piette, D. JNJ Soemers, M. Stephenson, C. F. Sironi, M. H. M. Winands, and C. Browne. Ludii-the ludemic general game system. In *European Conference on Artificial Intelligence*, 2020.

[27] Edward Powley, Daniel Whitehouse, and Peter Cowling. Bandits all the way down: Ucb1 as a simulation policy in monte carlo tree search. pages 1–8, 08 2013. doi:10.1109/CIG.2013.6633613.

[28] E. Rasmusen. *Games and Information: An Introduction to Game Theory*. Blackwell, 4th ed., 2007.

[29] Gijs-Jan Roelofs. Monte carlo tree search in a modern board game framework. 2012. Research paper available at umimaas.nl.

[30] Gijs-Jan Roelofs. Pitfalls and solutions when using monte carlo tree search for strategy and tactical games. *Game AI Pro 3: Collected Wisdom of Game AI Professionals*, pages 343—-354, 2017.

[31] Jahn-Takeshi Saito, Mark H. M. Winands, Jos W.H.M. Uiterwijk, and H. Jaap van den Herik. Grouping nodes for Monte-Carlo tree search. In *Computer Games Workshop*, pages 276–283, 2007.

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

[33] Chiara F. Sironi and Mark H. M. Winands. Comparison of rapid action value estimation variants for general game playing. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8, 2016.

[34] Mandy J. W. Tak, Mark H. M. Winands, and Yngvi Björnsson. Decaying simulation strategies. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(4):395–406, 2014.

[35] Mandy J.W. Tak, Mark H. M. Winands, and Yngvi Björnsson. N-grams and the last-good-reply policy applied in general game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2):73–83, 2012.

[36] Gabriel Van Eyck and Martin Müller. Revisiting move groups in monte-carlo tree search. In *Advances in Computer Games*, pages 13–23, 2011.

[37] Mark H. M. Winands. Monte-Carlo Tree Search in Board Games. In *Handbook of Digital Games and Entertainment Technologies*, pages 47–76, 2017.