DEGREE PROJECT, IN COMPUTER SCIENCE , SECOND LEVEL
STOCKHOLM, *SWEDEN 2015*

Attacking SameGame using Monte-Carlo Tree Search

USING RANDOMNESS AS GUIDANCE IN PUZZLES

SIMON KLEIN

KTH ROYAL INSTITUTE OF TECHNOLOGY

SCHOOL OF COMPUTER SCIENCE AND COMMUNICATION (CSC)

KTH Computer Science
and Communication

Attacking SameGame using Monte-Carlo Tree Search

Att angripa SameGame med Monte-Carlo Tree Search

Using randomness as guidance in puzzles
Använda slumpen som vägledning i pussel

SIMON KLEIN

Master's Thesis in Computer Science at CSC
Civilingenjör Datateknik internationell inriktning japanska
Masterprogram i Datalogi

Supervisor: Per Austrin
Examiner: Johan Håstad

Done at The University of Tokyo
Supervisor: Hiroshi Imai

sklein@kth.se

May 2015

Abstract

Since the birth of the computer, the creation of algorithms to beat humans in games has been a hot topic, algorithms that if successful often have vast implications. For some games and problems however, it has been notoriously difficult for computers to perform well. But in 2006 an algorithm known as Monte-Carlo Tree Search (MCTS) was invented that boosted the performance of computers in some of the difficult two-player games, e.g. computer Go. Around the same time researchers started to look into how MCTS could be applied to single-player domains, one of them was the puzzle of SameGame.

This thesis will continue the work on SameGame to further improve algorithms, ideas and knowledge on how to attack puzzles using MCTS. We mainly used a test set consisting of 250 randomly generated SameGame instances to evaluate the performance of various techniques. Doing this we managed to devise – among many things – a number of MCTS variations dynamically updating their explorative factor, leading not only to a clear improvement in performance but also mitigating the task of manually tuning parameters for the programmer. By combining ideas that empirically proved themselves successful in isolation, an algorithm matching (and sometimes outperforming) the performance of the previous NMCTS algorithm while using only a fraction of the resources was created, and on a different well known test set two competitive scores of 78936 and 80146 points were achieved.

The results suggest that there are still many ways in which MCTS for single-player domains may be improved, and many unexplored lines of thought remain. This work's contribution lies not in a single algorithmic idea tuned to perfection, but in the survey of several ideas and their combination, providing a solid foundation for future research to build upon.

Referat

Att angripa SameGame med Monte-Carlo Tree Search

Sedan datorns skapelse har algoritmer för att besegra människor i spel varit ett hett forskningsämne, algoritmer som när de är lyckade ofta får brett genomslag. För vissa spel har det dessvärre visat sig vara oerhört svårt för datorer att slå människor. Men år 2006 uppfanns en algoritm vid namn Monte-Carlo Tree Search (MCTS) som förbättrade datorernas spelförmåga i några av de svåra spelen, bl.a. Go. Omkring ungefär samma tidpunkt började forskare även undersöka hur MCTS kunde tillämpas på enspelar-problem, ett av dem var pusslet SameGame.

Detta arbete kommer att fortsätta forskningen på SameGame för att förbättra algoritmer, idéer och kunskap om hur pussel kan angripas med MCTS. Vi använde huvudsakligen en testuppsättning bestående av 250 slumpgenererade SameGame-instanser för att utvärdera hur olika metoder presterar. Genom detta tillvägagångssätt lyckades vi bl.a. skapa ett antal MCTS-varianter som dynamiskt uppdaterar sin explorativa faktor, vilket inte bara bidrar till förbättrad prestationsförmåga, utan också lättar programmerarens arbetsbörda att manuellt justera parametrar. Genom att kombinera idéer som empiriskt var för sig visade sig fungera bra, skapades en algoritm som matchar (och ibland överträffar) den tidigare NMCTS-algoritmen och som därtill bara använder en bråkdel lika stora resurser, och på en annan välanvänd testuppsättning lyckades två konkurrenskraftiga resultat om 78936 och 80146 poäng åstadkommas.

Resultaten tyder på att det fortfarande finns många sätt på vilka MCTS inriktat på enspelar-problem kan förbättras, och många tankegångar att utforska återstår. Detta arbetes bidrag till forskningsvärlden består inte av en enstaka finputsad algoritmisk idé dragen till sin spets, utan av undersökandet av många idéer och dess kombination, vilket bistår framtida forskning med en stabil grund att bygga vidare på.

Contents

Chapter 1

Introduction

Many problems appearing in everyday life and industry are way too complicated for a human being to solve analytically. Therefore the aid of a computer is indispensable. With the help of the computer's immense computational power, optimal solutions to a problem might be found. This can sometimes be done by trying all possible ways of solving the problem and choosing the best solution, a method commonly referred to as exhaustive search.

This approach, to try all solutions, is however even for a computer a far from feasible method for most problems. Therefore the study of algorithms is important. However, there exists a class of problems – so called NP-hard problems – for which it is believed that no efficient algorithms exist. Unfortunately these problems are not just a peculiar interest of mathematicians and scholars, but arise in many parts and aspects of society, such as planning, logistics and DNA-sequencing to name a few. Often the problems involve using given resources in the most economical way to achieve a certain goal, a task of greater importance than ever in light of the current environmental issues due to excessive and rash usage of natural resources.

Combinatorial search is a field within computer science and artificial intelligence that to a great extent deals with these NP-hard problems. The algorithms developed within this field are fundamentally the same as exhaustive search, but thanks to clever tricks and insights, optimal or close to optimal solutions may be found within reasonable time.

This work will address deterministic problems with perfect information and a single actor. That is, there is no randomness involved, no information is hidden from us and we are not competing with anyone but ourselves. For the sake of brevity we will refer to this class of problems as puzzles.

1.1 Puzzles

Puzzles and games are, despite being constructed mostly for the sole purpose of stimulating and teasing the minds of humans, a valuable domain for the development of algorithms. The well defined and clear environments of puzzles and games along

with their hardness, make it easy for new algorithms focused on relevant and central properties to emerge.

The ideas, concepts and algorithms that are developed in the field of games and puzzles often spread to neighboring fields. In industry a typical problem is often a hybrid of several fundamental tasks cluttered with special cases, making the corresponding solutions difficult to adapt to other domains. Puzzles on the other hand often address difficulties of a more general variety. For this reason, the use of a puzzle as a main focus and driving force in this work is justified, despite the lack of any immediate important application in practice.

1.2 Traditional search

1.2.1 Terminology

Before we start talking about the ways of search, we need to explicitly establish some common terminology on the notions to be used.

State A state is a specific position in the game reached by a sequence of moves from the initial position. The state does not only incorporate information about the current layout of a possible game board, but may also keep track of who's turn it is (two-player games), the number of doubles rolled in a row (the game of Monopoly), etc. Everything that affects the outcome from a specific point of the game is represented by the state.

Transition A transition is a possible move from a state. Often we also incorporate the change between the states that is brought about by the move in the transition, e.g. a change in the score.

Node From an algorithmic point of view we often store some of the states we have visited in some data structure, e.g. a tree. The information comprised by a state alone is not always sufficient, e.g. we might also want to keep track of how the state was reached. An entry including such auxiliary information along with a state is called a node.

Expand From a node x we typically create new related nodes, usually ones corresponding to states that can be reached from the corresponding state of x in a single move. When we create a new node y in this manner we say that node x expands node y, and that node y is expanded.

1.2.2 Approach

The common way to attack NP-hard puzzles is using some form of exhaustive search. Even though the search space is too large to search completely, many problems may be attacked successfully by methodically using strategies. The two most frequently used ones are as follows:

- Use a heuristic function to guide the search towards the goal faster.
- Prune fruitless states and moves, often accomplished using domain dependent knowledge.

Examples where these strategies can improve performance significantly over naïve search include the 15-puzzle and Sokoban [32, 33]. Alone they might not suffice to successfully attack a difficult search problem, but in most cases they constitute the major part of the approach.

1.3 Heuristic function

When a problem is NP-hard and has a too large search space, a heuristic function (often heuristic for short) is usually required in the context of search algorithms. The heuristic function's task is to evaluate the value of a certain state. In this way a search algorithm can use the heuristic to rank different choices and explore them in that order to find the goal state faster.

The heuristic is sometimes a vital part of an algorithm. Some algorithms require that the heuristic being used fulfills some criteria in order to function properly. Furthermore, the properties of the heuristic can also affect the guaranteed quality of a solution found by an algorithm.

In the following definitions we will consider the problem of maximization. Let S denote the set of all states and let g denote the goal state.

Admissible A heuristic is admissible if it does not underestimate the cost of reaching the goal from a state. Formally speaking, let $c(a, b)$ represent the true cost of going from state a to state b. In the context of maximization, a heuristic h is admissible iff $\forall s \in S : h(s) \geq c(s, g)$.

Consistent/monotone A heuristic is said to be consistent or monotone if it satisfies $\forall a, b \in S : h(a) \geq c(a, b) + h(b)$ and $h(g) \geq 0$. Intuitively this means that the heuristic estimates will improve as we move towards the goal state. A consistent heuristic is as a consequence also admissible.

Bounded relaxation Sometimes it may take too long time for a search algorithm to reach the goal state despite using a heuristic to guide it. And sometimes we are merely interested in finding a solution or a sufficiently good one quickly. In these cases the search may be sped up by sacrificing the admissibility criterion, which on the other hand often also incurs the loss of optimality. This is done by weighting the heuristic estimate by some value ε, i.e., we use $h_\epsilon(s) = \varepsilon \cdot h(s)$ to guide the search instead. The value of ε should satisfy $\varepsilon \leq 1$.

For minimization the opposite inequalities should hold in the way one would expect. We have also assumed that there is only a single goal state g, but should it be the case that there exists several goal states, we could connect them all to a single artificial goal state using transitions with zero cost.

1.4 A*

One of the most successful approaches to attacking NP-hard problems is the A* (pronounced A star) search algorithm. A* belongs to a family called best-first search algorithms, which iteratively expands nodes from the most promising node according to some specific rule. In A* this rule is – within the context of maximization – to choose the node x maximizing the function $f(x) = g(x) + h(x)$ where $g(x)$ is the reward of reaching node x and $h(x)$ is an admissible heuristic estimate of the maximum possible reward achievable from node x.

The expanded nodes are stored in a priority queue. If the heuristic is consistent as well we may keep track of the states we have visited and avoid visiting the same state twice. This gives the algorithm the following structure.

```
(1)   Q ← {start}
(2)   g[start] ← 0
(3)   while Q is not empty
(4)      current ← POLL(Q)
(5)      if current = goal
(6)         return TRACE(current)
(7)      foreach successor succ of current
(8)         if succ ∉ g or g[current] + d(current, succ) > g[succ]
(9)            g[succ] ← g[current] + d(current, succ)
(10)           if succ ∉ Q then ADD(Q, succ)
(11)                       else UPDATE(Q, succ)
(12)           parent[succ] ← current
```

Due to the fact that the heuristic is admissible we know that we have found the optimal solution once we try to expand nodes from the goal state. If a heuristic with a bounded relaxation of ε is being used, the solution found will at least be as good as ε times the optimal solution.

1.5 Beam search

One issue with A* is however, that it may require a significant amount of memory. An algorithm solving this issue but sacrificing completeness and optimality is beam search. Beam search works analogously to BFS with the exception that it limits the number of states kept at each level of the search to B, the so called beam width. The algorithm works then as follows, given the at most B states at level n, generate all successors for each state, if the average branching factor is b we will typically have bB candidates, then we assign each state a heuristic estimate and the B nodes with the best heuristic values will be kept and the rest discarded. Choosing the B best states can be done in $O(bB \log B)$. These B states gives the so called beam at level $n + 1$.

The memory consumption of beam search is typically $O(B)$ and the running time is $O(dbB \log B)$ where d is the depth of the solution.

1.6 Lack of heuristic

Both A* and beam search as well as many variants thereof rely on the existence of a good heuristic. A* needs one in order to achieve low running time, if the heuristic is too rough the algorithm will degenerate into a BFS. Beam search requires a sensible heuristic in order to not prune a state on the path to the optimal solution. However, there exist many problems where no good heuristic is known.

Go is a two-player game where it turns out to be difficult to estimate the value of an intermediate game position [35]. Due to this most computer programs were – from the time of the first Go program in 1968 up until recently – at an amateur level compared to humans, and Go was thusly seen as the next great challenge of AI after a computer had defeated the human champion in Chess in 1997 [36]. But in 2005-2006 a new algorithm showed up that gave rise to a drastic improvement of Computer Go programs over the following years [18]. The name of the algorithm was *Monte-Carlo Tree Search* (MCTS).

MCTS does not need any heuristic to estimate the value of a position. The estimation is based on random sampling of a position. This turned out to work remarkably well in Go and thus research on MCTS literally exploded. On average a new article in the field is estimated to be published every fourth day. So in spite of being a relatively new research field, a lot of research is already present and MCTS has been successfully applied in two-player games such as *Hex* and *Havannah*, real-time video games such as *Ms. Pac-Man* as well as non-deterministic games such as *poker* and *Magic: The Gathering*.

1.7 Research focus

However, most research on MCTS has been focused on Go and two-player games. In comparison single-player games and puzzles have not gained a lot of attention. Research has been done in the field, but there is certainly room for improvement and new insights. Therefore this thesis aims to improve the knowledge of MCTS in a single-player context, more specifically its usage addressing puzzles. The research question may then be formulated as follows:

> In what ways can we hope to improve algorithms and methodologies in fully observable deterministic contexts where it is difficult to estimate the value of an intermediate state?

This includes resolving unsolved issues left by previous work, critically review their strategies and improving upon them, as well as coming up with fairly new ideas and evaluating their applicability in the field.

To accomplish this we will have a look at a difficult game for search algorithms called SameGame to tackle in Chapter 2. In Chapter 3 we will get a deeper understanding of MCTS and the many variants relevant for a single player context. Using this knowledge we devise and evaluate new strategies and methods in their most basic form in Chapter 6, but before that we ourselves examine the complexity of SameGame in Chapter 4 and explain the method of our experiments in Chapter 5. From the experimental results and gained insights, we further enhance methods in Chapter 7 by combining the most fruitful ideas. Finally we conclude and give further directions of research in Chapter 8.

The ultimate goal of this work is to develop an algorithm, as well as several techniques, performing well in a domain where input is variable. The purpose of the latter criterion is to impose some kind of generality on the work and techniques developed, increasing their chance of usability in domains apart from the targeted one. A concrete goal of the work is to achieve a score of at least 80000 on the so called standard test set of the game SameGame while keeping the underlying algorithm as general as possible without resorting to test set specific parameter tuning and excessive use of computational resources.

Part I

Background

Chapter 2

SameGame

SameGame is a fully observable and deterministic single-player puzzle game devised by Kuniaki Moribe originally released under the name *Chain Shot!* in 1985. In 1992 Eiji Fukumoto ported the game to Unix with a slight variation in settings, this time under the name we know the game by today, SameGame. [31]

2.1 Rules

SameGame has a board consisting of an $m \times n$ grid initially filled with blocks of k different colors. The player is allowed to remove orthogonally connected groups of blocks of the same color, provided that the number of blocks in the group is at least 2. Points are awarded on a per move basis depending on the number of blocks removed. The game ends when the board is cleared or no more blocks can be removed, i.e. no group of blocks consists of more than a single block. The goal of the game is to maximize the score.

2.1.1 Mechanics

After a group of blocks is removed, some cells of the grid will be empty and blocks above these will fall down. If a column becomes empty, columns to the right will be shifted to the left. More formally the game mechanics taking place after each move can be described in two steps applied in order.

1. For each block B, move it down by y cells, where y is the number of empty cells below B in the same column.

2. For each column C, move it to the left by x columns, where x is the number of empty columns to the left of C.

In this way new groups of blocks of the same color might be created and old groups might split into parts. However, ignoring color all blocks will continue to remain a single connected component (see Figure 2.1).

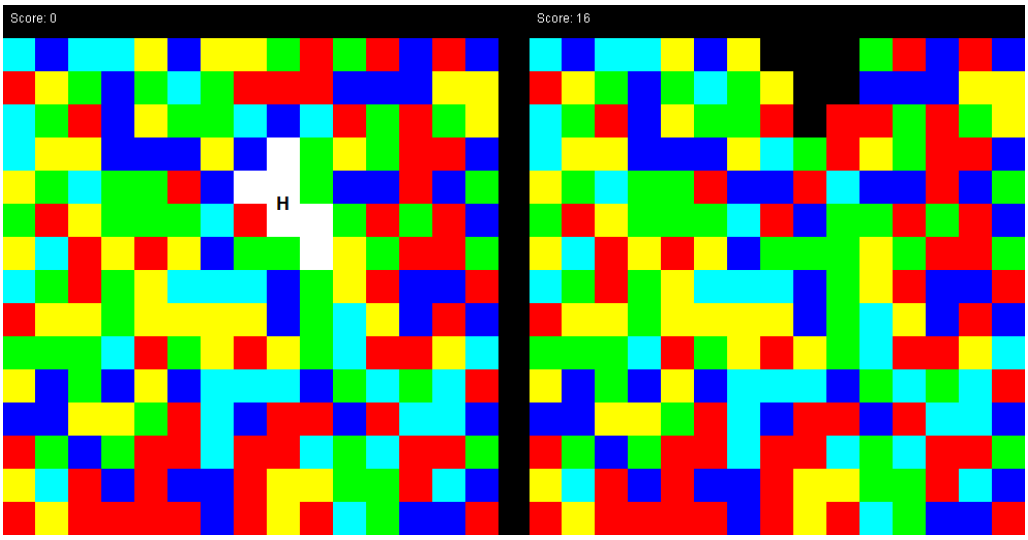

Figure 2.1. Illustration showing a SameGame board before removing the group of 6 blocks highlighted in white and marked with an H (to the left), and the SameGame board after applying the move (to the right).

2.1.2 Scoring formula

The points awarded for removing a group of n blocks is $(n-2)^2$, i.e. we benefit greatly from creating and removing large groups. If a different scoring formula is used, the game is usually named differently.

2.1.3 End game bonus

If the board is cleared of all blocks a bonus of 1000 points is awarded. This bonus usually constitutes somewhere around 20-50% of the final score for a typical instance using standard settings if solved by the best SameGame solvers/players.

2.1.4 Score deduction

If the board is not cleared, points are subtracted from the score based on the number of blocks remaining. The procedure used for deducting points is however not entirely uniformly defined across different sources. The reason being most likely the fact that most solvers manages to clear the board anyway.

Some sources simply deducts $(m-2)^2$ points from the score where m is the number of remaining blocks [14]. Other sources does the same but for each color, i.e., $\sum_{i=1}^{k}(R(i)-2)^2$ points are deducted where $R(i)$ represents the number of blocks remaining of color i [23]. The former procedure puts an overwhelmingly big focus on clearing the board, whereas the latter offers – in our opinion – a far more interesting tradeoff between clearing the board and optimizing the move sequence. For that reason we have opted for the latter procedure in this work.

2.1.5 Standard settings

The standard settings used for SameGame within research is a 15×15 board with 5 colors [23].

2.1.6 Naming

There exists a handful of games very similar to SameGame. Usually a game is named differently if the scoring formula is different. Some common names include *Bubble breaker*, *Clickomania!* and *Jawbreaker*. *Clickomania!* is for instance a variant where the sole objective is to clear the board or minimize the number of blocks remaining. And in *Bubble breaker* the scoring formula $n(n-1)$ is used.

2.2 Motivation

At first it might not be obvious why we should care about such an obscure game such as SameGame. But there exists as a matter of fact a handful of reasons why SameGame is interesting from both a theoretical as well as a practical point of view.

2.2.1 Beyond traditional search

The main reason for using SameGame as a means for evaluating search algorithms – Monte-Carlo based ones in particular – is that there seems to exist no good admissible heuristic, which is more or less a prerequisite for A*-based approaches. Furthermore, few moves/states can be pruned, and the game-tree and state-space complexity are both rather overwhelming (see Section 2.5) making any naive search approach infeasible. Also, the game mechanics makes it impossible to create sub games. Put simply, SameGame is a formidable challenge for AI and search algorithms.

2.2.2 Beyond traditional optimization

In SameGame moves are irreversible, since we can only remove blocks. It is also possible that the very first move is crucial in order to get even somewhere close to the global optimum. This makes it difficult for global optimization methods such as Simulated Annealing (which has been successfully applied to TSP [13]) that are based on iteratively moving to neighboring solution strings – a sequence of moves – in order to escape local maxima. The reason being that if a specific move is altered all subsequent moves must be altered as well, i.e. a suffix of the solution string can only be changed, degenerating the approach to a simple brute force search stuck in a local maximum.

2.2.3 Previous research

A fair share of previous research has been done using SameGame as a benchmark for evaluating search algorithms. This makes it easier to compare performance with existing approaches. The most commonly used benchmark[1], hereby referred to as the standard test set, consist of 20 instances of which the first 10 are random, the next 5 has an equal amount of each color and the last 5 has one dominating color (the number of blocks for each color is 40, 40, 40, 40 and 65 respectively).

The first approach to successfully attack SameGame was an undocumented one called Depth Budgeted Search (DBS) originating at the University of Alberta in 2007, scoring 72816 points on the standard test set. But it did not take long before Schadd et al. in 2008 beat the score using their so called Single-Player Monte-Carlo Tree Search (SP-MCTS), scoring 73998 points. In 2009 a score of 76764 was achieved by Monte-Carlo with Roulette Wheel Selection (MC-RWS) which in the same year was beaten by Nested Monte Carlo Search (NMCS) scoring 77934. Meanwhile an unknown competitor named Spurious AI of which little is known achieved a score of 84414 points. This record was then beaten in 2010 by Heuristically Guided Swarm Tree Search (HGSTS) with a score of 84718, which remains today the best out of all documented approaches. [26]

Name	Year	Score
DBS	2007	72816
SP-MCTS	2008	73998
MC-RWS	2009	76764
NMCS	2009	77934
Spurious AI	????	84414
HGSTS	2010	84718

A lot has happened since 2010 though. In 2012 SP-MCTS was improved achieving a score of 78012 [26]. Spurious AI has been able to improve its score to 87140. As of today (April 2015) the high score list is topped by a competitor under the name “tcooke” scoring 87842, though there is no paper describing the approach and resources used by this solver. [14]

Sadly there is no restriction on the resources used in order to find the solutions to the standard test set’s 20 levels and the used resources by existing approaches are rarely specified in detail. This makes it both difficult and unfair to use the standard test set to compare different existing approaches. However, Schadd et al. has devised a test set of 250 random levels [26, p. 12] which has been extensively used in their research together with a good documentation of computational resources used, making it an appealing alternative and complement for evaluation.

[1] http://www.js-games.de/eng/games/samegame

2.2.4 Interesting properties

Many problems can be modeled as high score games, so an algorithm yielding good solutions in SameGame very well could have more straightforward practical applications. Moreover SameGame exhibits some interesting properties.

Long term vs. short term The end game bonus adds long term thinking, which we must balance against the short term rewards of creating and removing large groups. If we only focus on making large groups we might lose the chance of the bonus, while on the other hand concentrating solely on the bonus will likely lead to low cumulative rewards.

Gambling We must be willing to split a large group in order to reconnect it and make it bigger later on. Sometimes a move required will split large groups built up, but opening up for new better possibilities. This must be foreseen by an algorithm, gambling the current guaranteed rewards to get future greater rewards.

Anti-greed We must also choose the right point in time when to delete a large group. If all effort is focused on making a specific group larger we may miss the opportunity to create other promising groups at the expense of only making futile moves keeping the specific large group intact. Although similar to *Gambling* in that the problematic actions undertaken are more or less the same, this point focuses on a different aspect of human psychology and the algorithmic remedies required are possibly different.

These are all properties showing up more or less frequently in real world situations. Situations in which most humans find difficulties making choices.

2.2.5 SameGame in comparison

Although SameGame is in itself an interesting optimization problem with a huge search space requiring long term thinking, it is not the only one of its kind. Thus it is imperative to compare it with other puzzles/problems and outline the properties making SameGame unique.

Sokoban

A popular and famous problem in the domain of planning is the puzzle of Sokoban, where the objective is to push a set of boxes to a set of goal squares. The game is known to be notoriously difficult due to the sheer size of the search space. Apart from completely different game mechanics, the main difference between SameGame and Sokoban is their objective. SameGame is a problem in the domain of maximization where the length of any solution is bounded by the size of the board. Sokoban on the other hand belongs to the domain of solving (although we can reformulate it into a problem of minimization) where solutions may be of arbitrary

length due to the possibility of undoing previous moves, something being impossible in SameGame. This is a major difference since it affects the set of applicable techniques to the two domains drastically. Moreover, Sokoban is not only NP-hard but also PSPACE-complete [33].

A difference of a minor characteristic is the usability of pruning. In Sokoban there exists a fair number of known strategies to discover fruitless states and pointless moves. SameGame however, has not even near as many and efficient pruning strategies, making it difficult to guide the search using domain specific knowledge.

A similarity of the two problems is however, that they have to deal with variable input, i.e. different instances/levels. A strategy successfully working on one instance might be severely degraded in performance on another. This forces techniques in both domains to a large degree be solely developed with the game mechanics in mind, rather than the structure of the instance adressed.

Morpion Solitaire

Another problem popular for the evaluation of algorithms in single-player domains is the puzzle of Morpion Solitaire where the player aims to draw as many lines as possible in accordance with the game mechanics. Just like SameGame the puzzle is in the maximization category and any solution is of finite length. For this reason, many algorithms performing well in SameGame also perform well in Morpion Solitaire and vice versa. There is however one major difference. The input of Morpion Solitaire is not variable, the game typically always use one specific configuration (or one of a few appropriate ones). The reason for this being that other configurations are simply not suitable, they either have an infinite achievable maximum score or a known very low upper bound on the maximum score [19]. This is in contrast to SameGame where it is not only possible to change the configuration of all the blocks and their colors, but also the size of the board and the number of the colors. For this reason we may argue that an algorithm addressing SameGame has to be more versatile than one addressing Morpion Solitaire.

2.3 Heuristic overestimate

Although there is no known good admissible heuristic for SameGame there are still room for devising heuristics that may be usable in a subset of the solving process.

Bonus available: If there exists a color of which there exists only one block, then we can be sure that the bonus of 1000 points cannot be achieved. Otherwise we assume that we can get the bonus.

Over-estimator: Assuming that all remaining blocks of the same color are connected and then applying the scoring formula to each one of them and summing the values gives an upper bound on the score together with the *bonus available* heuristic.

2.4 Pruning

Unlike many other puzzles very few states or moves can be pruned, which is one of the reasons why SameGame is a challenging task for artificial intelligence and search algorithms. Nevertheless there exists at least two strategies, one for pruning states and one for pruning moves, that can be utilized. Although of limited use in traditional search when used in isolation, it remains to be examined to what degree they improve performance in anytime-algorithms such as MCTS.

2.4.1 State pruning

Given the maximum score achievable reaching a certain state and an upper bound on the score remaining as prescribed by the *over-estimator* heuristic, if the sum of those two does not exceed the overall best solution so far, then we can prune that state. Obviously this will in most cases be a fruitless strategy, due to the coarse upper bound. However, the upper bound improves towards the end of the game when the blocks get fewer and if the search tree is deep this might have some positive impact.

2.4.2 Move pruning

In SameGame we can refrain from considering removing a group if the removal of that group does not result in any new groups and the group itself cannot be split into parts by a removal of another group, provided that there are other moves not having these two properties. The reason being that if the removal of a group does not yield any new moves and cannot be fragmented by another move, we can postpone the removal of the group hoping that it might become larger later on.

However, checking these two criteria would require a one-step look-ahead, which may be considered too computationally expensive with respect to the gains. But we can limit our pruning to a subset of these moves, namely groups being vertical segments without any block above and with at least one block below (see Figure 2.2). Obviously the removal of such a group cannot yield a new move and since it is a vertical segment it cannot be split into parts. Checking if a group is a vertical segment fulfilling said conditions can be done relatively fast on average. We will call this specific pruning technique *VS-pruning*.

2.5 Complexity

As previously mentioned, SameGame is an NP-complete problem, meaning that there (probably) exists no algorithm that can decide in polynomial time (with respect to the number of cells) given a target score and a game position whether a score larger or equal to the target score can be achieved from the position. This was essentially proven in two steps. First Biedl et al. [7] showed among other things that the similar game of Clickomania with 2 columns and 5 colors is NP-complete

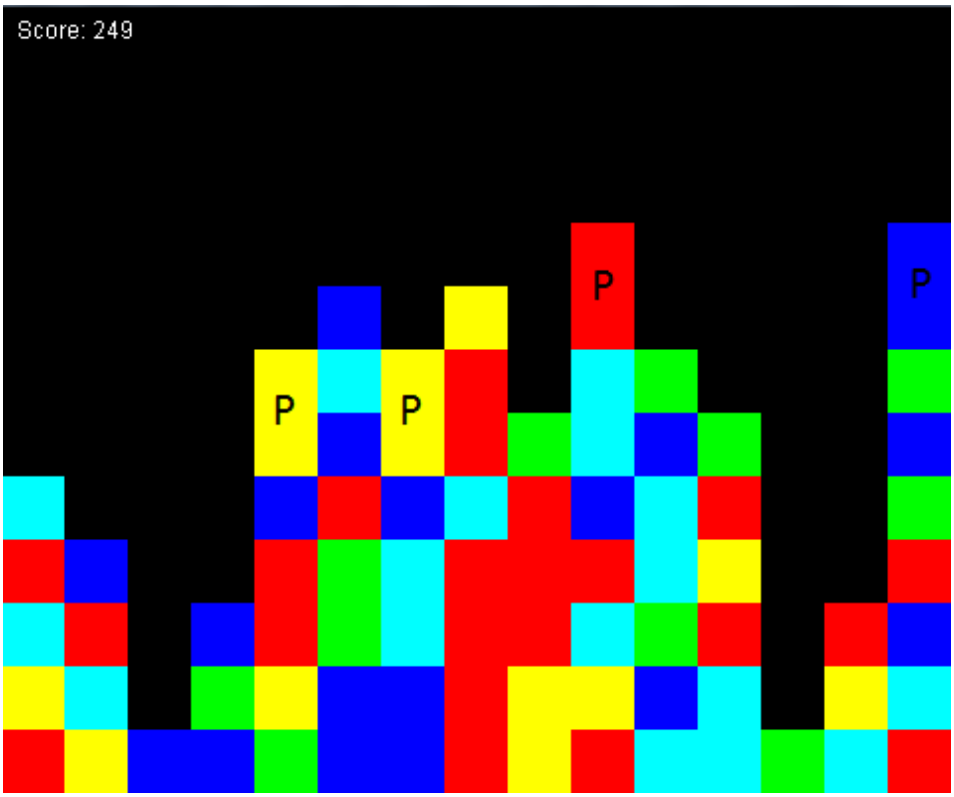

Figure 2.2. Illustration showing four vertical segments (marked with P) that may be disregarded as moves at the moment.

by reducing the strongly NP-complete 3-partition problem to it. Then Schadd et al. [24] showed that SameGame is at least as hard as Clickomania.

In most games however, the setup is fixed. Chess is for instance played on a fixed 8×8 board with a fixed number of pieces and standard SameGame is played on a 15×15 board with 5 colors. That is, the size of the board does not vary, or alternatively the input to the problem is static, so discussing the complexity class is – although interesting – of limited use in these cases.

In games one tends to talk about two other complexity metrics instead, viz. state-space complexity and game-tree complexity [2].

2.5.1 Game-tree complexity

The game-tree complexity of a game corresponds to the number of leaf nodes in the full-width search tree required to assess the value of the initial position(s), where full-width means that the tree contains all nodes at each depth. For two-player games and minimax search [34] this gives an estimate of the number of positions required to evaluate.

However, calculating the game-tree complexity exactly is often far from a simple

task and an approximation must therefore be used. A common way of approximating the game-tree complexity is by calculating the average game length L and the average branching factor B by random simulations of the game, and then letting the approximation of the game-tree complexity be B^L. Despite being a crude estimate, it has been frequently used for many problem domains, so using it allows for a reasonably fair comparison with these.

The game-tree complexity of SameGame has been approximated to 10^{85} by Schadd et al. [23] using the aforementioned method. One million instances of the game were played randomly giving an estimated average length of 64.4 and an average branching factor of 20.7. However, since SameGame does not have a specific initial position it does not make sense to talk about a single game-tree complexity, unless one tries to solve all SameGame instances simultaneously which is rarely the case. What one rather should talk about is the worst-case game-tree complexity possible for any instance or the typical game-tree complexity for a random instance. Although this distinction is merely of a semantic nature, we will in Section 4.1 present experiments conforming to this line of thought as well as results pertaining the usage of *VS-pruning*.

2.5.2 State-space complexity

The state-space complexity of a game represents the number of legal states reachable from the initial position(s), which differs from the game-tree complexity measuring the number of ways the game can be played. When this figure is too difficult to calculate exactly, an upper bound including illegal or unreachable positions can be used instead. The state-space complexity of a game is almost usually smaller than the game-tree complexity, since the latter may include a particular position several times in the tree.

Schadd et al. [23] reports a state-space complexity of 10^{159} for SameGame, which with the game-tree complexity of 10^{85} in mind may be perceived as a rather remarkable figure. The source of the unusual difference in the two complexity measures is simply put that the complexities as a matter of fact refer to two different problems. The reported state-space complexity has been found by calculating the number of different board positions you can create with a 15×15 board and 5 colors [25]. That is, 10^{159} is the state-space complexity of solving all SameGame instances, while 10^{85} is the expected game-tree complexity of a single random instance. We will address and try to resolve this issue in Section 4.2.

Chapter 3

Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) is a search algorithm based on a combination of tree search and random sampling. The algorithm itself is in its nature general and can be applied to decision processes such as games as well as combinatorial optimization problems such as the *travelling salesman problem*.

The building blocks constituting MCTS naturally predates the algorithm itself, and the most competitive variations are a result of several different fields coming together. To get a good understanding of the field of MCTS we will have a look at these subcomponents, including the over 60 years old Monte-Carlo method, the equally old multi-armed bandit problem and the recent breakthroughs that have led to the rise and success of MCTS.

3.1 The Monte-Carlo method

The basic idea of a Monte-Carlo method is to use random simulations to estimate the value of an intermediate position. The rationale being that if random simulations from an alternative A results in overall better rewards than random simulations from an alternative B, then we might hope that optimal or near-optimal play from A is better than from B. In a two-player game this corresponds to the insight that if player A wins the majority of all random simulations from a position P, then P is probably a winning position for player A as well.

Given a state s and a set of actions $A(s)$ available at s, let $f(s, a)$ denote the state reached by applying action a and $avg(s, a)$ denote the average score of all simulations started at $f(s, a)$. A typical implementation of a so called *flat Monte-Carlo method* may then proceed as follows:

```
PLAY(start)
(1)   s ← start
(2)   while s is not terminal
(3)     a ← CHOOSE(s)
(4)     s ← f(s, a)
CHOOSE(s)
(1)   for i ← 1 to n
(2)     foreach a ∈ A(s)
(3)       SAMPLE(s,a)
(4)   return maxarg_{a∈A(s)} avg(s, a)
SAMPLE(start, a)
(1)   s ← f(start, a)
(2)   while s is not terminal
(3)     choose a_2 from A(s) randomly
(4)     s ← f(s, a_2)
(5)   update avg(start, a) with score(s)
```

That is we run n – where n corresponds to some kind of computational budget – simulations from each of the child nodes of a given state, then we simply pick the one yielding the highest average score as our next move. The simulation or sampling will typically involve randomness in one way or another, though it may not necessarily be uniformly random.

There is however an inherent problem with a *flat Monte-Carlo approach*, namely run time. In most domains running a simulation is quite expensive compared to a simple heuristic estimate and running several simulations is even more computationally expensive. Furthermore, it may take very long before the estimated value through simulations starts to converge to a reasonable value. However, we are comparing alternatives and are merely interested in the best alternative, the ranking between inferior alternatives is often irrelevant. Since it may become blatantly obvious quite soon that one alternative is inferior to another, we would ideally want to perform more simulations from the promising alternatives and less from the unpromising ones. These insights lead to two directions of improving the Monte-Carlo method, speeding up simulations and informed allocation of computational resources.

3.2 Monte-Carlo with a Tree

The key insight to speed up the simulations is that certain prefixes of the simulations' solution strings are more common than others. The expensive operation during simulation is often to calculate all possible available moves at a certain state and to which states they lead. If some prefixes of moves are common, certain states are visited several times and duplicate work will be done. However, this could be avoided by caching the states and their possible moves. Obviously we cannot cache

each state encountered during simulation due to memory constraints, but we could try to cache the most frequently visited ones, which typically are states close to the initial state. Implicitly, this gives rise to a tree structure and it is this combination of a tree structure and Monte-Carlo simulations that coins the name MCTS.

So instead of simply running simulations from the immediate successors of the initial state, we maintain a tree where the root corresponds to the initial state, traverse down the tree until we hit a current leaf, run a simulation from the leaf and propagate the results back up the tree. For the time being and sake of example let say that the tree has been pre-computed prior to the simulations, although this will not be the case for MCTS. Should we allocate resources uniformly over all children as in a *flat Monte-Carlo method* and apply this principle of uniformity to each node of the tree, we would typically get a full-width search tree up to some level. E.g. if the initial state corresponding to the root has 10 children which in turn has 10 children each, and our budget is 1000 simulations; instead of running 100 simulations from each node at level 1 as would have been done in a *flat Monte-Carlo method* we run 10 simulations from each node at level 2. This would improve performance slightly since we only calculate the possible moves at level 1 once. It is however, most likely far from ideal and for the sake of generality we introduce the following two definitions:

Tree policy Prescribes how the search space that is part of the search tree should be traversed.

Simulation policy Prescribes how the search space that is not part of the search tree should be traversed, also known as *Default policy*.

We are now in position where we can describe the four steps commonly defining MCTS (see also Figure 3.1).

1. **Selection**: Starting at the root R of the search tree, recursively select a child node according to the tree policy until a node C not part of the search tree is reached.
2. **Expansion**: Attach the node C to the search tree.
3. **Simulation**: Run a simulation starting at C using the simulation policy.
4. **Backpropagation**: Update all nodes on the path from C to R using the results obtained from the simulation.

The update procedure in step 4 typically updates the number of visits, the average score and possibly some other relevant statistics. Let r be the result obtained in step 3, x a node, $n(x)$ the number of visits of x and $sum(x)$ the sum of its scores (the average follows from these two). Then for each node x on the path C to R the following is at least done:

$$n(x) \leftarrow n(x) + 1$$
$$sum(x) \leftarrow sum(x) + r$$

Starting with a single node – the root – corresponding to the initial position and iteratively applying steps 1 through 4 we have defined MCTS. The steps 1 through 4 are typically repeated until the computational budget is spent, e.g. time or memory. Thereafter the action corresponding to the best child of the root is selected. In most scenarios parts of the tree may be kept to improve upon selection of future actions.

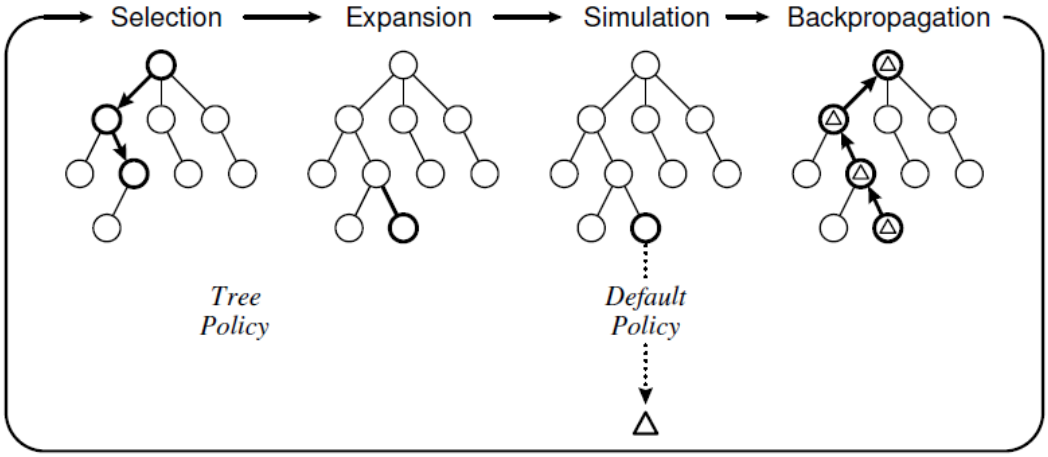

Figure 3.1. Illustration from [30] showing one iteration of MCTS.

Worth noticing is that the parent node of C does not necessarily need to be a leaf of the search tree, due to the unknown specification of the tree policy. The tree policy will highly influence the structure of the tree and typically it will grow asymmetrically towards some part of the search space. If the tree policy is focused on exploitation we will get a tree with deep fringes while a tree policy focused on exploration will result in a flatter tree. The problem of how this trade-off should be done remains.

3.3 The Bandit Problem

A well studied problem that precedes MCTS and deals with allocation of resources is the *multi-armed bandit problem* from the field of probability theory. The problem is usually formulated as follows:

> A gambler is standing in front of N slot machines, each having its own distribution which is unknown to the gambler. When the gambler plays a machine he gets a reward at random from that machine's particular distribution. All machines are independent from one another and the rewards lie in the range $[0, 1]$. What strategy should the gambler apply in order to maximize his rewards?

What the core of this problem addresses is known as the *exploitation-exploration dilemma*. The gambler wants to play the best machine as many times as possible

– exploitation, however, in order to get an idea of which machine that is the best among all of them they all have to be tried a sufficient number of times – exploration. After a number of plays the gambler typically has an estimation of each machine's expected reward, but his estimation is likely to be flawed due to randomness, so a machine that the gambler thinks is less profitable might simply be unlucky for the moment being. And therein lies the dilemma.

Regret: The regret of a strategy is the expected difference between the optimal strategy and the strategy applied so far. More formally speaking, if M is the maximum expected value of any machine and T is the number of lever pulls so far with obtained results $r_1 \dots r_T$, the regret p is defined as $p = M \cdot T - \sum_{i=1}^{T} r_i$.

The goal of a policy should be to minimize the regret. It has been shown for a large number of families of distributions that no policy can achieve a regret growing slower than $\Omega(\log T)$ [16], so a policy is considered to have resolved the dilemma if it achieves this rate of growth. Policies achieving this were devised at the same time the theorem was proved, however, the methods used were computationally expensive. But ten years later in 1995 Agrawal [1] devised a usable method, though with a larger constant on the growth. Based on this work Auer et al. [3] created in 2002 the UCB1 policy, achieving logarithmic regret. UCB standing for Upper Confidence Bound, which relates to the fact that we want an upper bound on the rewards we can hope for from a specific machine.

UCB1: Let T be the total number of plays so far, $n(x)$ the number of times machine x has been played and $avg(x)$ its average reward so far.

1. Play each machine once.

2. Loop: Play the machine x maximizing $avg(x) + \sqrt{\frac{2 \ln T}{n(x)}}$.

The first term in the policy constitute the exploitation and the second corresponds to exploration. Noteworthy is that a play of a specific machine does not only decrease the value of its explorative term but it also increases the explorative term for all other machines. The new breakthrough of UCB is that it achieves a logarithmic growth of the regret uniformly over time and not only asymptotically. This is clearly a desirable property in practice where resources are finite.

Although UCB1 exhibits marvelous theoretical properties, it does not necessarily need to be the best policy in practice. As a matter of fact Auer et al. also devised a policy called UCB1-TUNED.

UCB1-TUNED: Let x_i be the reward obtained from the ith play of machine x. Define $V_T(x) = \left(\frac{1}{n(x)} \sum_{i=1}^{n(x)} x_i^2\right) - avg(x)^2 + \sqrt{\frac{2 \ln T}{n(x)}}$.

1. Play each machine once.

2. Loop: Play the machine x maximizing $avg(x) + \sqrt{\frac{\ln T}{n(x)} \cdot \min\left(\frac{1}{4}, V_T(x)\right)}$.

This policy also takes an upper bound on the variance into consideration. The rationale being that this would tune the upper bound more finely. Auer et al. did not provide any proof for the policy's growth of regret, but it was empirically shown to outperform UCB1.

3.4 UCT

In 2006 Kocsis and Szepesvári [15] came up with the idea to use the UCB1 formula to solve the exploration-exploitation trade-off issue in MCTS. By considering the choice of a child node as the multi-armed bandit problem, the today widely used UCT (Upper Confidence bound applied to Trees) algorithm was invented. Let p denote the parent node and x the child node being considered, the algorithm then uses the formula

$$UCT(x) = avg(x) + 2 \cdot C_p \cdot \sqrt{\frac{\ln n(p)}{n(x)}}$$

in the tree policy where C_p possibly is a node-specific constant (nodes having $n(x) = 0$ are considered to have a value of ∞). That is, from a parent node p we traverse down the child node x maximizing the UCT-formula. The authors showed that although the sequence of values obtained from a node will drift in time as the sampling probabilities of descendants change, it will still be possible to choose the constant C_p while maintaining the properties of UCB1. Primarily that the sampled average value of a node x having parent p satisfies the following two inequalites (where $\mu(x)$ is the true current expected value of x, and P denotes probability):

$$P\left(avg(x) \geq \mu(x) + \sqrt{2 \ln n(p)/n(x)}\right) \leq n(p)^{-4}$$
$$P\left(avg(x) \leq \mu(x) - \sqrt{2 \ln n(p)/n(x)}\right) \leq n(p)^{-4}$$

i.e., the sampled average value is within distance $\sqrt{2 \ln n(p)/n(x)}$ of the true expected value with at least a probability of $1 - 2 \cdot n(p)^{-4}$.

There is however a distinct difference between the multi-armed bandit problem and a typical game, in the former there is randomness and an expected reward associated with each choice, but in the latter each choice is often associated with an absolute value under optimal play (in two-player games you either win or lose). So for a game the sampled average value at a node in the search tree could be perceived as pointless, we would rather like to know the optimal choice assuming optimal play. Fortunately it is in this respect UCT shines with yet another advantageous property, v.i.z. that the probability of choosing the optimal move at the root of the tree converges to one at a polynomial rate, though this is under the assumption of infinite memory. But UCT based variants have proven themselves

very successful in experiments as well as Computer Go, making it extensively used among contemporary MCTS implementations [18].

3.5 Improvements

Although the original formulation of MCTS is elegant, there are specific ways to improve it. The success of the methods may depend heavily on the problem domain.

3.5.1 AMAF

Using UCT we have a theoretical guarantee that estimates provided by the nodes of the search tree will converge to the corresponding true values as the number of simulations goes towards infinity. However, no one has infinite resources and ideally one would want the search tree to converge as fast as possible. One family of techniques to accomplish this is All Moves As First (AMAF).

The observation made by AMAF is that if a specific move, e.g. placing a stone on a specific intersection in Go, is favorable in one particular state it is probably also favorable in other similar states. This works by updating the statistics for each move used during the simulation during the backpropagation phase. That is, not only do we update statistics in the nodes used to traverse down the search tree, but also in the siblings having an associated move used during the iteration. During the selection phase this data is used, either solely or in tandem with the regular statistics.

One issue however, is what should be considered a move? This is highly domain dependent and different definitions might yield different outcomes. In SameGame for instance a move could be a removal of a specific set of blocks, but due to the drastic changes of the board this might not be appropriate and one could also want to take blocks below or above the set into consideration. In the game of Arimaa AMAF techniques incurred no benefit while they have been successfully applied in Computer Go [30, p. 31 p. 27].

3.5.2 RAVE

One of the most popular techniques from the AMAF family, mainly due to its successful applications in Computer Go, is Rapid Action Value Estimation (RAVE). As a matter of fact, RAVE is today considered a must-have extension for Computer Go solving programs [6].

The key idea of RAVE is to separate AMAF statistics and regular statistics, and use the latter when many simulations have been done and the former when few simulations have passed through the node in question. The rationale being that the regular statistics may initially be unreliable relative to the AMAF statistics. RAVE aims to balance these two aspects by using the following modified UCT-formula as

the tree policy:

$$UCT_{RAVE}(x) = (1 - \beta\left(n(x), m(x)\right)) \cdot avg(x) + \beta\left(n(x), m(x)\right) \cdot avg_{AMAF}(x) + C_p \cdot \sqrt{\frac{\ln n(p)}{n(x)}}$$

where $n(x)$ is the number of regular visits of the node, $m(x)$ is the number of AMAF visits and $\beta(n, m)$ is a function that should be close to zero for relatively large values of n and close to one for comparatively large values of m. One suggested definition of $\beta(n, m)$ is

$$\beta(n, m) = \frac{m}{n + m + 4b^2nm}$$

where b is an empirically chosen constant [27].

Worth pointing out is that the clear distinction between AMAF and RAVE outlined here may not always apply. There are variations of both of them and sometimes it can be difficult to make a precise distinction of the two.

3.5.3 From Tree to DAG

In the search tree of MCTS a single node represents a certain state and a specific way of reaching that state. In other words, several nodes in the search tree could represent the same state. The associated value of a state however, does not depend on the sequence of moves used to reach it, so if a simulation is done from one of these nodes, we could in theory propagate the result to all of them. If done correctly this would be a considerably more efficient use of the data provided by the simulations.

In practice we would also like to not waste memory, so having several nodes representing a single state is rather unnecessary. However, with one node per state in the tree, we would still have to include edges from all previous parent nodes of the specific state, which would degenerate the tree into a directed acyclic graph (DAG). All the steps of MCTS except one are still applicable using a DAG as the underlying data structure instead of a tree, the odd one out being the backpropagation step. Using a tree there is a unique path from an arbitrary node to the root, but when using a DAG this is not the case.

Many alternative definitions for the backpropagation phase do exist, but among the straightforward general ones there are essentially two.

1. Update all nodes lying on a path between the root and the simulated node.

2. Update only the nodes used to traverse down to the simulated node in the current iteration.

The second alternative can obviously lead to some inconsistencies, such as a child node having more visits than its parent. So we might be tempted to try the first alternative, but that is far from a wise choice. Because the first alternative could fail to find the optimal solution even when given infinite resources. Why is that?

Consider the following example: A node P having children X and Y, X has only one child U which is also shared with Y, in addition to this Y has two other

children V and W (see Figure 3.2). The expected reward from U, V and W are medium, low and high respectively. Assume that U has been expanded, while V and W has not. This means that X and Y have the same number of visits and the same average score, so let us assume that the next iteration favors Y and V gets expanded. Now Y will have a lower average score and a higher number of visits than X. So the next iteration will definitely favor X which only has one choice, to run simulations from a descendant of U, when this is done both X and Y will be updated, so X will still have a higher average and a lower number of visits than Y. So from now on all future iterations will favor X over Y and the state W will never be explored under UCT.

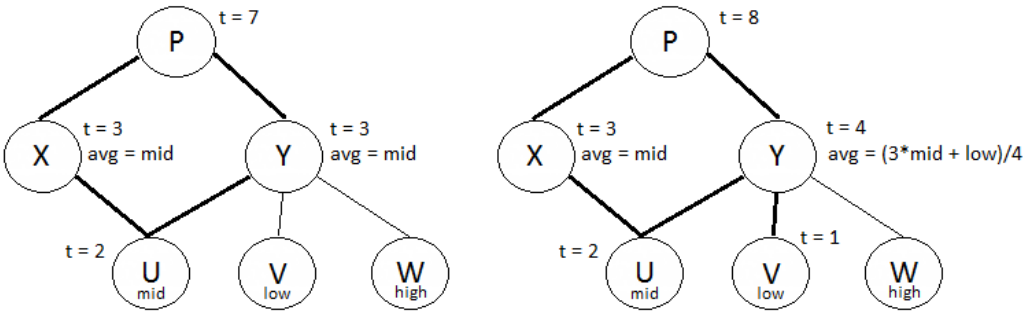

Figure 3.2. Bold lines mark edges that are already part of the search tree, whereas weak lines indicate moves leading to states not yet part of the search tree. The statistics may look differently depending on implementation, but in this example we see to the left when both X and Y has been expanded by P and both nodes have in turn expanded the child node U. To the right we see a future iteration of MCTS when Y has expanded the child node V. From now on all iterations from P will pass through X and the node W will remain unknown, or at least the move from Y to W.

The aforementioned example is obviously a very specific case, but the problem may arise for sibling nodes X and Y where $children(X) \subseteq children(Y)$ when we have $avg(x) \geq avg(y), n(x) < n(y)$ or $avg(x) > avg(y), n(x) \leq n(y)$.

3.5.4 UCT1 and UCT2

A neat way of resolving this issue is due to the work of Childs et al. [10] where they presented three new policies UCT1, UCT2 and UCT3 respectively, out of which UCT2 was the most promising one. (The UCT3 policy gave superior results, but was quite expensive.) The main idea of Childs et al. is to store information not only in the nodes but also in the edges. Let $avg(a, b)$ and $n(a, b)$ denote the average score and number of iterations respectively associated with using the move taking us from state a to state b, in graph theoretic terms the average score and number of traversals of the edge connecting node a and b. The UCT1 and UCT2 policy is then defined as follows.

$$UCT1(x) = avg(p,x) + C_p \cdot \sqrt{\frac{\ln n(p)}{n(p,x)}}$$
$$UCT2(x) = avg(x) + C_p \cdot \sqrt{\frac{\ln n(p)}{n(p,x)}}$$

So the key idea is to use $n(p,x)$ rather than $n(x)$ in the explorative term of the selection policy formula to avoid convergence to an incorrect value. The reason why UCT2 outperforms UCT1 is that $avg(x)$ is a more accurate estimate than $avg(p,x)$.

3.5.5 UCD

For the general case there luckily exists another satisfying way of resolving the issue with the backpropagation phase using a DAG, viz. Upper Confidence bound for rooted Directed acyclic graphs (UCD) developed by Saffidine et al. in 2010 [22]. The main idea behind UCD is – as in the previous section – to store the statistics in the edges rather than the nodes, e.g. the average score associated with following a specific edge. Using this model the backpropagation phase is once again unambiguously defined, provided we only update the edges selected during the current iteration. The problem now on the other hand is the selection phase. How UCD resolves this and other issues is elaborated on in Section A.1, since it is not crucial in the understanding of this work.

3.6 The single-player context

Applying the UCT algorithm to zero sum two-player games such as Go is quite straightforward. To model the average score we could use the number of wins divided by the number of simulations. If draws may arise in the game we might treat them as a half win or even a loss. Moreover, since we never can fully anticipate the next move to be made by the opponent, moving to parts of the search tree where we have a high chance of winning is of interest.

In a puzzle however, there is no opponent and we are normally dealing with a score rather than wins and losses. This complicates as well as simplifies the application of UCT to puzzles. The most striking differences are listed below.

Simplifying

- There is no assumption on the abilities of an opponent.
- There is no deadline per move, all time may be exhausted at the initial position.

Complicating

- We are dealing with a score rather than a binary win-lose situation.
- We are interested in the maximum score rather than the average.

The first two points are clearly simplifications since they both make certain aspects of the implementation easier as well as opening up for new possibilities. We could for instance remove/avoid duplicate states in the search tree without necessarily enforcing a DAG structure.

The third point is a complication. The win-ratio is in a sense a globally and universally unambiguous value. Imagine that we have done a fair number of simulations from a node and the resulting win-ratio is 99%, without even looking at the siblings of this node we may say that it definitely seems to be an advantageous position for us. Whereas the same scenario in a puzzle with a resulting average score of 2000 does not tell us anything, is 2000 good or bad? A score is merely a relative value and without any knowledge about the performance of the siblings it is of little use. Furthermore, the first term in UCT-formula should be in the range $[0, 1]$, meaning that the score would have to be normalized should a UCT variant be applied to the domain.

The fourth point is a very good objection to the use of UCT or any bandit-based tree policy. Because the multi-armed bandit problem addresses the problem of exploiting the alternative with the best expected reward as often as possible in order to maximize a cumulative reward, while in the single-player context we want to achieve the maximum possible reward, be it only once, that is picking the alternative maximizing the expected maximum reward. In other words, the alternative with the best average score is not necessarily the same as the alternative with the best maximum score. However, due to the nature of UCT the average score will converge towards the maximum score provided that we may perform a specific sequence of moves more than once.

3.7 Existing research

The single-player context is substantially different from the two player case, and therefore as a consequence a bunch of MCTS variations specialized on this domain have seen the light of day. Many of them use SameGame as a means of evaluation and some are even solely focused on SameGame. We will have a look at the most prominent ones in the literature.

3.7.1 SP-MCTS

Single-Player Monte-Carlo Tree Search (SP-MCTS) is possibly the most predominant MCTS implementation in the research literature of MCTS in a single-player context. The algorithm was devised in 2008 by Schadd et al. [23] and was one of the first of its kind.

Foundation

In its essence SP-MCTS is essentially UCT with a tweaked tree policy. The modified formula used for node selection looks as follows:

$$UCT(x) = avg(x) + C \cdot \sqrt{\frac{\ln n(p)}{n(x)}} + \sqrt{\frac{\sum_{y \in R(x)} (y - avg(x))^2 + D}{n(x)}}$$

Where $R(x)$ denotes all simulation results passing through x. The difference from the original formula being that a third term corresponding to a standard deviation has been added. A constant D is added to inflate the deviation of yet rarely explored nodes, to make sure that they will remain considered uncertain.

Taking the standard deviation into consideration makes sense when you try to attack a puzzle. Since we are only interested in achieving a good score at least once, stable positions with low variance are of smaller interest, whereas positions with high variance are more interesting. E.g. a position with average score 950 and high variance probably has a better maximum score than a position with average 970 and low variance.

In response to the changed tree policy the backpropagation phase naturally has to update the sum of all squared results in the third term as well. This is essentially all differences from ordinary UCT, but Schadd et al. has also altered the expansion phase slightly in that a node is only allowed to expand a child node if it has been visited at least T times, being a well known trick among practicers of MCTS, possibly invented by Coulom [11]. The reason behind this is to save memory and only expand children of relatively stable nodes.

Meta-search

Despite the care taken to include explorative properties in the tree policy, Schadd et al. found that the algorithm tended to easily get stuck in local maxima. This could obviously be mitigated by choosing a more explorative setting, i.e., increasing the values of C and D. However, it was found that deep search trees were important in some domains, more specifically SameGame, mainly because shallow trees were unable to reach down into the local maxima finding the necessary sequences of moves yielding the bonus. Thus a more exploitative setting was crucial as well.

This dilemma of both wanting to have exploitation and exploration at the same time, or rather exploitation of different parts of the search space, was resolved by introducing a meta-search. Instead of performing one large search, k smaller searches each using $1/k$ of the total resources are performed, and the overall solution is the best one over all smaller searches. In order to make the different searches explore different parts of the search space, each search is also initialized with a new seed for its random generator. Worth noting is that no information is shared between the searches.

Simulation policies

In general stronger play during simulations, that is a more refined simulation policy, does not necessarily have to result in an overall better search algorithm, but when the problem domain is a puzzle this is likely to be true, since there is no interference by an opponent. Schadd et al. boosted the performance of their SameGame solver by using a better so called simulation strategy, i.e., a simulation policy.

They defined three different simulation policies for SameGame.

Random Choose a move at random.

TabuRandom Prior to simulation, choose a color at random that will be the taboo color. Remove no blocks of this color during simulation unless no other move is available. Choose a move at random among the available ones.

TabuColorRandom Similar to TabuRandom, but the most frequent color prior to simulation is chosen as the taboo color.

The goal of the latter two strategies is to create large groups of one color, since this is often beneficial in SameGame. In general *TabuColorRandom* yielded the best result (about a 50% increase over *Random* in average score on a test set of 250 random levels) and has since then been used in other research aimed at SameGame as well, e.g. HGSTS [12]. Matsumoto et al. [17] also experimented with other simulation strategies, though the results showed that as the resources got large enough, there was little difference in average score and computational time between most heuristics tested.

Normalization

Schadd et al. present no general way of normalizing the score, but suggests the following three ways of resolving the issue.

Adjusted constants Adjust the values of C and D in the tree policy formula such that they are reasonable for the problem domain attacked.

Constant-scaling Assume that the maximum score for the domain is M, scale values using M.

Upper bound-scaling Calculate an upper bound on the maximum score M achievable for the problem instance at hand, scale values using M, e.g. $avg(x)/M$.

All of these methods are dissatisfactory in one way or another. The first two are not general and require the programmer to have good domain specific knowledge and also time to experimentally verify his choices. The third method is not suitable when applied to SameGame, since it is notoriously difficult to find a good such bound and the over-estimator heuristic would cause most values to end up close to zero. It is not entirely clear which solution Schadd et al. opted for, but it is one of the first two.

Improvements

A long time has passed since the genesis of SP-MCTS and naturally improvements have been done [26]. These are the major ones:

- Add a fraction W (e.g. $W = 0.02$) of the maximum score found so far (from the concerned node) to the average score in the tree policy.
- Deviate from the simulation strategy (a.k.a. simulation policy) with a small probability ϵ (e.g. $\epsilon = 0.003$).
- Allocate resources per move rather than game (as is the case for two-player games), i.e., when (total resources)$/L$ has been spent, we choose the best next move once and for all and continue our search from the resulting state. The best next move is the move with the highest maximum score. Typically $L =$ avg game length, but other choices are also possible.
- Tune the parameters involved in the algorithm, first and foremost C and D, using a machine learning algorithm e.g. the Cross-Entropy Method.

All of the stated improvements make sense. Since the parent of one strong node and several weak ones may have a low associated average score, it is reasonable to also take the maximum score achieved from the node into consideration. Moreover, the simulation strategy can in some cases be a limiting factor in that there exists some sequences of moves (possibly optimal ones) that will never be executed.

The major improvement however, is the third one. It turns out that moves closer to the root are more likely to get simulated than moves further down in the search space. This could result in weaker play during end game and thus also a suboptimal score. By moving down the search tree created this issue is mitigated. In a successful implementation of SP-MCTS, Tak [28, p. 9] let $L = 30$, since the overall best solution found usually had been accomplished by then.

Conclusions

Although the applications of SP-MCTS has been focused on SameGame there are several conclusions one may draw from the work of Schadd et al. that are generally applicable to the field of MCTS in puzzles.

- For domains with an end bonus constituting a substantial part of the final score, deep search trees are likely to be important.
- A refined simulation policy may improve performance significantly in puzzles.
- The search might get stuck in a local maximum. Try to avoid it using a meta-search approach.
- If resources (e.g. time) are limited, we might be better off exploiting a local maximum than trying to find the global one.

Question marks left open by previous research

The work behind SP-MCTS is very thoroughly done, despite this there are still some aspects of the algorithm and its implementation that may need some more clarity.

- To what extent does the third term improve performance?
- Should not the third term be weighted as well?

As future research Schadd et al. primarily suggests the following directions.

- Can we enhance SP-MCTS using an AMAF technique e.g. RAVE?
- Can we improve the meta-search by sharing information between searches?

To these we may add surveying the following aspects as an interesting option.

- So far resources per move has only been allocated using a uniform distribution. However, a uniform distribution need not be optimal, e.g. it might be interesting to allocate more resources to earlier moves. Is there another more suitable distribution?
- Can we devise a more general and still well functioning method to normalize the values involved in the UCT-formula?
- Is it possible to accomplish a better meta-search approach than simple random restarts?
- Can we use another approach than meta-search to avoid local maxima?

The fact that SP-MCTS is a simple yet efficient algorithm to attack puzzles, makes it suitable as a foundation to build upon. We will therefore in this work examine many of the the various possible improvements and question marks outlined here.

3.7.2 SA-UCT

A quite natural idea to consider in the single-player context is trying to be more explorative early on and more exploitative later. This is exactly what Ruijl et al. [21] has done in their so called SA-UCT, where simulated annealing is in a sense incorporated into UCT.

The algorithm makes use of a function $T(C_{start}, R_{left}, R_{total})$ decreasing over time, where R_{left} is the number of resources left, R_{total} is the total number of resources started with and C_{start} is the intended value of the explorative factor at the start of the algorithm's execution. This corresponds to the use of temperature in simulated annealing. Using this function the UCT-formula is augmented as follows

$$SA\text{–}UCT(x) = avg(x) + T(C_{start}, R_{left}, R_{total}) \cdot \sqrt{\frac{\log n(p)}{n(x)}}$$

where the authors chose the linear function

$$T(C_{start}, R_{left}, R_{total}) = C_{start} \cdot \frac{R_{left}}{R_{total}}$$

although they do propose as future research to study other functions, e.g. exponentially decreasing ones.

The authors of SA-UCT has applied it to the domain of expression simplification where they obtained promising results, especially regarding the range of successful (initial) values of the explorative factor which was increased tenfold.

3.7.3 Additional work

In this section we will describe algorithms that have had an impact on the field in one way or another, but that are not essential to understand the upcoming methods outlined in this work. Additional information can be found in the appendix for each of the algorithms.

MC-RWS

Monte-Carlo with Roulette-Wheel Selection (MC-RWS) is a flat Monte-Carlo method specifically aimed at SameGame created by Takes and Kosters [29]. The core of the algorithm lies solely in its strategy to perform simulations, which tries to create larger groups of several colors simultaneously. Despite being very limited in its applications, MC-RWS is an interesting case to have a look at. The reason behind this and a more in-depth explanation of the algorithm can be found in Section A.2.

NMCS and NMCTS

Nested Monte-Carlo Search (NMCS) is a variation of the *flat Monte-Carlo method* created by Cazenave [9] but inspired by many earlier techniques; Nested Monte-Carlo Tree Search (NMCTS) devised by Baier and Winands [5] is in a sense an extension of NMCS. The key idea of these two algorithms is to recursively apply themselves during simulation. In this way the estimates provided by the simulations will be of higher quality.

More about these two algorithms – including comments on their contribution to the field – can be found in Section A.3.

NRPA

A continuation on the work of NMCS is Nested Rollout Policy Adaption (NRPA) created by Christopher D. Rosin in 2011 [20]. The main idea of NRPA is to dynamically adapt the simulation policy (also known as rollout policy) during the search. Adaption of the simulation policy is done by gradient ascent [37] with the objective of a gradually increasing bias towards the best solution found so far.

More about NRPA – including comments on its impressive feats – can be found in Section A.4.

BMCTS

A relatively new algorithm in the field is Beam Monte-Carlo Tree Search (BMCTS) devised by Baier and Winands in 2012 [4]. The key idea of the algorithm is to combine MCTS with the concepts of beam search, in order to trade completeness and convergence to optimality for speed and space. This is done by permanently pruning seemingly unpromising nodes in the search tree at a per depth basis.

Finer details can be found in Section A.5.

HGSTS

In the field of SameGame, Heuristically Guided Swarm Tree Search (HGSTS) is one of the most competitive approaches. HGSTS is an algorithm based on UCT devised by Edelkamp et al. [12] relying to a great extent on hardware, parallelization and clever implementation. More on this algorithm may be found in Section A.6.

Part II

Development

Chapter 4

SameGame Complexity

Before one starts attacking a problem it is usually wise to conduct some analysis of it. In our case this has to some extent already been covered by previous research, but as was pointed out in Section 2.5 there is still room for an improved analysis. For that purpose this brief chapter will address the issues identified and present new results pertaining the complexity of SameGame.

4.1 Game-tree complexity

Approximating the typical game-tree complexity of SameGame may be achieved by estimating the game-tree complexity for a number of instances and calculating the average complexity value. But in order to get a feel for the typical complexity one ought to average the logarithm of the instances' complexities, since the average value will be heavily dominated by the instance with the greatest complexity. By estimating the complexity of an instance as mentioned in Section 2.5.1, using 256 random simulations over 8192 random instances we get the following values.

Min	Median	Average	Max	Log-average	Std-dev Log-avg
$3.6 \cdot 10^{66}$	$2.5 \cdot 10^{84}$	$5.7 \cdot 10^{96}$	$4.6 \cdot 10^{100}$	84.23	4.81

This coincides with the previous results of Schadd et al. and we may claim that the typical game-tree complexity of a SameGame instance is indeed in the region of 10^{85}. In Figure 4.1 we can see how the sampled complexity measurements are distributed around this number.

We could also look at the estimated game-tree complexity using the move pruning technique described in Section 2.4.2 with the same settings.

Min	Median	Average	Max	Log-average	Std-dev Log-avg
$7.6 \cdot 10^{60}$	$5.2 \cdot 10^{80}$	$7.5 \cdot 10^{93}$	$5.7 \cdot 10^{97}$	80.56	4.78

As we can see the game-tree complexity decreases with about four orders. It remains to see what benefits this corresponds to in the search algorithm, which we will briefly examine in Section 6.5.

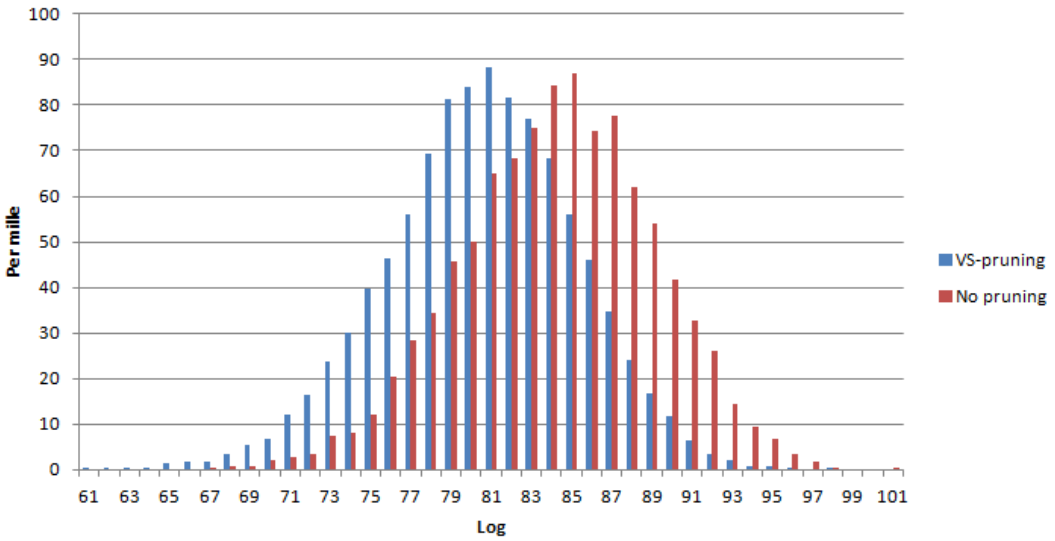

Figure 4.1. Illustration showing the distribution of the estimated game-tree complexity for random SameGame instances.

4.2 State-space complexity

In Section 2.5 we pointed out an inconsistency in the previously reported numbers on the game-tree and state-space complexity of SameGame, namely that the former concerned the complexity of a single instance whereas the latter concerned the complexity of all possible instances. In order to make the two numbers relate to each other we could either multiply the game-tree complexity figure with the number of different initial positions or we could try to calculate a typical upper bound for the state-space complexity of a random instance. Clearly the later metric is the more interesting one, since we are focusing on solving particular instances.

Upper bound calculation: There are several ways in which an upper bound for the state-space complexity of a certain instance can be calculated. A simple but yet reasonably good estimate is to calculate the number of possible distinct configurations of every column assuming that we are able to remove groups of single blocks as well. Each group of a column (a continuous monochromatic vertical segment) can either be included or excluded in a configuration, so the number can rather straightforwardly be computed in $O(2^n)$ time where n is the height of the column (in our case $n = 15$), provided we use a fast data structure, e.g. a hash table, to avoid counting duplicate configurations. Finally we multiply these numbers together to get the upper bound. Because we include the empty configuration for each column and have an ordering of them, simply multiplying the numbers counts all subsets of non-empty columns as well as the empty board configuration.

Typical upper bound for state-space complexity: Using the method previously described on 100000 randomly generated SameGame instances we get the following values.

Min	Median	Average	Max	Log-average	Std-dev Log-avg
$4.11 \cdot 10^{44}$	$2.84 \cdot 10^{52}$	$5.53 \cdot 10^{54}$	$3.98 \cdot 10^{58}$	52.42	1.53

Thus we may conclude that the typical upper bound for the state-space complexity of a random SameGame instance is 10^{53}. Although this is a quite pessimistic estimate, it asserts that the real figure is nowhere even near the previously estimated number of 10^{159} (recall Section 2.5.2).

Chapter 5

Method

In Chapter 6 we perform relatively basic experiments and present the obtained results. Prior to that it is however appropriate that we elaborate on the methodology being used, i.e., implementation details, amount of resources, how the experiments were performed and the notation used.

5.1 Implementation

Although different versions function differently there are some common implementation details that are not explicitly outlined in the definition of MCTS or UCT. For the sake of clarity we specify them here.

Null preference Nodes that have an associated value of zero visits are always chosen first, i.e., all children of a node need to be explored before the finer mechanics of the tree policy are applied.

Null randomness If there are several unexplored moves of a node, one of them is chosen uniformly at random.

Solved nodes Even in MCTS it might be the case that for some node all possible future move sequences are explored, i.e. a node gets completely solved. We keep track of these such that no move sequence is explored more than once. A node is completely solved either if it is a terminal node or if all its children are completely solved.

VS-pruning quick finish When using VS-pruning we may reach a point when all moves remaining are moves normally pruned by VS-pruning, of course these moves must be performed since they may yield points, but instead of making them available in the search we simply sum the points of all such segments and subtract it from the points about to be subtracted due to the other blocks left at the board. That is we consider such a position as a terminal position and assign it a customized score.

5.2 Resource metric

What ultimately matters is in what time frame an algorithm manages to output a solution of a certain quality and how much memory was required during the process. However, apart from the properties of the algorithm itself, there are many factors affecting the speed, such as the hardware used for execution and the programming language used for implementation. Ideally we would like to have a way to measure resources independent of such factors and giving precedence to the theoretical properties of the algorithm rather than optimizations in implementation, making comparisons more fair across the research field.

Schadd et al. [23] uses the number of nodes in the search tree as a metric to evaluate their MCTS based approaches, i.e., a solely memory focused metric. This is indeed a hardware independent metric, but it fails to capture some aspects of time in the measurement. The main issue being that nodes higher up in the search tree are more expensive than nodes lower down, since they require longer simulations, which is where many MCTS based algorithms spend their time. As an alternative to this we propose the following metric of one resource unit (ru).

- An inspection of a state, i.e. a calculation of all possible moves from that state, consumes a single ru. This also includes move generation during the simulation phase in MCTS based approaches.
- For MCTS based implementations traversing one level down the search tree consumes 1/8 ru.

This metric puts more focus on the main operation performed, i.e. move generation. Which also makes it possible to make wider comparisons, e.g. comparing MCTS and A* based implementations. However, very exploitative searches spend a lot of time traversing the search tree. In order to make them finish within a reasonable time frame a penalty is inflicted on moving down the search tree. In this way we get a metric that corresponds to time as well as memory and that is independent of hardware, programming language and other simultaneously executing processes.

The reason behind the choice of 1/8 for tree traversal was that for a given number of resources it seemed to yield quite consistent running times for exploitative and explorative searches alike on the experimental machine.

5.3 Testing

For research where conclusions are based on experiments, it is of utmost importance that the experiments are adequately performed.

Much of the previous research has based its evaluation on running the algorithm being evaluated once on a large test set of instances. This is a reasonable approach for randomized algorithms if the instances of the test set are in some sense equal, e.g. of equal difficulty or equal score distribution. If this is not the case, the result

on the whole test set will be heavily influenced by the algorithm's performance on a subset of important instances. For instance, let say that the algorithm's solution ends up uniformly between the optimal score and 80% of it, assume now that we have a test set of 100 instances of which 99 has an optimal score of 1000 and one has an optimal score of 1000000. In this case the result on the test set is basically decided by the performance on that sole instance, so what may seem as a test set of 100 instances is in fact a test set of a single instance. In order to counter such irregularities in a test set, we advocate that several independent runs of the algorithm should be done on each instance of the test set. Alternatively one could also group instances after some property such as score or complexity.

5.3.1 Our tests

If not otherwise stated we execute five independent runs on a single instance where each run uses a number of 640000 ru. This usually results in between 10000-80000 nodes of the search tree with an average number of nodes around 58000 on the Schadd test set (which will be the targeted test set if not otherwise stated). Furthermore, each implementation's parameters has been tuned in order to yield an average node depth around 30, in order to make the comparisons more fair. Here the average node depth refers to the average depth of a node in the in-memory built search tree, i.e. also the average depth from which we start running simulations if we expand one node of the search tree at each iteration of MCTS. For game positions not part of the search tree, e.g. ones merely encountered during simulations, we typically use the term state instead.

5.4 Results

Because we often execute several runs on each single instance of a test set, we can present a much more detailed view of the results than just a single average score value. In the results sections we will present the following three values.

Min The average over all instances of the worst case run (out of the five runs for the instance).

Avg Simply the average over all runs (over all instances).

Max The average over all instances of the best case run.

This is done in order to get an idea of the stability of a certain aspect of the algorithm. Often we will also present the values *Avg depth* and C, corresponding to average node depth and the value of the explorative factor used (to achieve the certain node depth).

In some cases we will also present graphs showing not only Min and Max but also the following three values for further insight.

Q1 The average over all instances of the first quartile, i.e., the run yielding the second smallest value.

Q2 The average over all instances of the second quartile, i.e., the run yielding the median value.

Q3 The average over all instances of the third quartile, i.e., the run yielding the second largest value.

So in the event of the Schadd test set we will execute five independent runs on each of the 250 instances, for each instance the five scores will be sorted in ascending order, and the values Min, $Q1$, $Q2$, $Q3$ and Max will correspond to the average of the score at position 1 through 5 respectively. Also worth noticing is that the value $Q2$ and Avg are not the same.

The average scores presented are rounded to the closest integer. For the average node depth, results presented are rounded to two decimal places.

5.5 Notation

Throughout the remaining part of this work we will use the following notation.

$n(x)$ The number of iterations that have passed through the node x, alternatively the number of simulations run from some node in the subtree rooted at x.

$avg_{global}(x)$ The average score of all iterations passing through the node x.

$avg_{local}(x)$ The average score of all iterations passing through the node x, disregarding the points obtained before reaching it, i.e. $avg_{global}(x) = S + avg_{local}(x)$ where S is the number of points obtained on the path from the root to x.

$top_{global}(x)$ The maximum score that has been achieved for any iteration passing through the node x.

$top_{local}(x)$ The maximum score that has been achieved from the node x, disregarding earlier gained points.

$sum_{global}(x)$ The score sum of all iterations passing through the node x. Do note that we have $sum_{global}(x) = avg_{global}(x) \cdot n(x)$.

$sum_{local}(x)$ The score sum of all iterations passing through the node x, disregarding earlier gained points.

$R(x)$ The vector of all iteration results passing through the node x.

$V(x)$ The variance of all results associated with the node x, i.e.,

$$V(x) = \frac{\sum_{y \in R(x)} y^2 - n(x) \cdot avg_{local}(x)^2}{n(x)}$$

best The globally best score obtained so far.

C A statically defined constant (if not stated otherwise).

Chapter 6

Examination

There are various basic concepts in the field of Monte-Carlo Tree Search in a single player context that have yet not been fully researched, of which some were mentioned in Chapter 3. The purpose of the following chapter will be to examine these, i.e., concepts that may be considered as a quite fundamental part to most MCTS based approaches. Doing this we may hope to acquire a comprehensive foundation of valuable insight that can be used by this project – as well as future projects – to build upon.

To a large extent we will test various ideas in isolation and one by one. Apart from being compared to each other, we will also often benchmark them against a reference version that will be devised and decided on in Section 6.1.

6.1 Normalization

Mapping the score to a value in the range $[0, 1]$ is an important aspect of making UCT applicable to single-player domains, and is consequently an issue suitable starting with before one moves on to examining more complex aspects. We will have a look at three different variations for normalizing the average score term in the UCT formula.

Definitions

From a parent node p choose the child node x maximizing

Local

$$UCT_L(x) = \frac{avg_{local}(x)}{top_{local}(p)} + C \cdot \sqrt{\frac{\log n(p)}{n(x)}}$$

Global

$$UCT_G(x) = \frac{avg_{global}(x)}{best} + C \cdot \sqrt{\frac{\log n(p)}{n(x)}}$$

Static

$$UCT_S(x) = \frac{avg_{global}(x)}{5000} + C \cdot \sqrt{\frac{\log n(p)}{n(x)}}$$

Do note that special care may need to be taken concerning non-positive numbers using the *Local* and *Global* policies, but this has not been addressed as it is generally not a problem.

Results

Along with the obtained average scores for each policy we also present the value of C used in the tree policy formula and the average node depth this choice of parameter resulted in.

Name	Min	Avg	Max	C	Avg depth
Local	2301	2629	2957	0.066	29.74
Global	2351	2652	2960	0.053	29.66
Static	2342	2653	2975	0.020	30.54

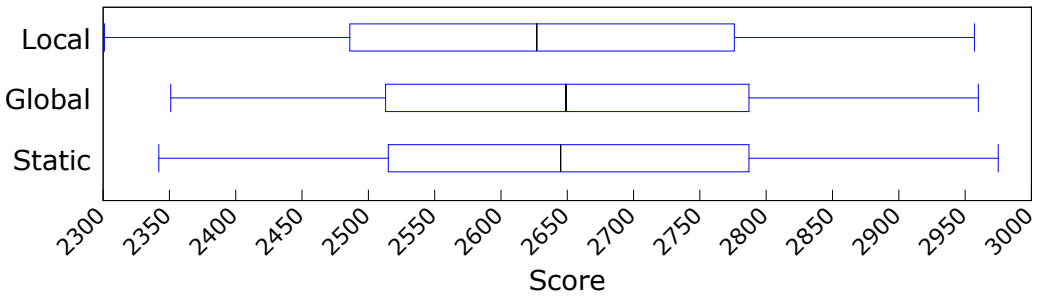

Figure 6.1. Illustration showing the average for Min, Q1, Q2, Q3 and Max for Local, Global and Static respectively (Schadd test set). Do note that the average of Q2 (the median) is not the same as the average as a whole.

Comments

Overall the results are quite similar (based on personal experience, we estimate that 10 points is within statistical deviation), but there are some tangible differences we may comment on.

Local There are at least two observations we can make concerning the *local normalization scheme*. Firstly, it is in its nature prone to attraction by local maxima and thus needs the highest explorative constant C of the three to counter this. Secondly, due to this nature it is worse than the *global scheme*

in the *Avg min category* but almost as good in the *Avg max category*, i.e., it gets easily stuck in a local maximum but is good at exploiting it.

In addition to this we may note a fundamental difference between the *local scheme* and UCT as it is normally applied in two-player games. In the *local scheme* we essentially focus equally much on exploitation regardless of where in the search tree we are, whereas UCT in two-player games uses an unambiguous win ratio resulting in less exploitative focus in suboptimal branches. One might argue that the latter qualities are desirable since the best known line of play in a suboptimal branch is simply not good enough and something drastically different needs to be found, i.e. we should be less exploitative.

Global Regarding the *global scheme* we may say that it is almost as good as the *static scheme* and has better worst case performance compared to the *local scheme*. On the other hand a rather large value of C is required which indicates that issues with local maxima do exist. One may suspect that the search may try to exploit the currently best known line of play a bit too much early on, since the globally best found solution should be quite bad early on but increase in quality as time progresses. This issue could be mitigated by scaling the value used to normalize with some value $k(R_{left}, R_{total}) \geq 1$ that converges to 1 as time goes by, i.e. as the number of resources R_{left} left decreases.

Static The *static scheme* seems to have the best overall performance and is also in its nature the least exploitative. Normalizing using a static constant suffers from the fact that it has to be set so that is "good on average", i.e. it might be the case that it's not well suited for some instances, but when it is appropriate in relation to the instance we get "early-exploration late-exploitation behavior". That is, because the solutions found initially will likely be of poor quality the explorative term will dominate, but as the solutions improve more and more focus will be given the exploitative term. One could argue that this is a desirable property. In a sense this is the opposite of the *global scheme*, where the influence of the exploitative term decreases over time as the best score found so far improves (though the average scores improve over time as well, they do not improve at the same rate as the best score found).

However, a drawback of the *static scheme* is that it requires domain-specific knowledge, i.e., the normalization constant needs to be hardcoded beforehand. This is a weakness that is not present in the other two schemes.

From now on we will to a large extent use this version as a reference version for other implementations to benchmark against.

From Figure 6.1 we can make some further observations. The most striking one is that all the three boxplots look quite symmetric, which is far from a necessary result.

6.2 Variance

An interesting question one might ask is "does the third term in the tree policy formula for SP-MCTS improve over ordinary UCT and to what extent?", i.e., what are the benefits of taking variance into consideration in the tree policy? Results concerning this has not been presented by Schadd et al. [23] and is therefore an issue worth proceeding with after one has settled for the normalization technique.

Definitions

To examine this we implement SP-MCTS (recall Section 3.7.1)using static normalization as follows. From a parent node p choose the child node x maximizing

$$UCT_{SPMCTS}(x) = \frac{avg_{global}(x)}{5000} + C \cdot \sqrt{\frac{\log n(p)}{n(x)}} + \sqrt{\frac{V(x)}{5000^2} + \frac{D}{n(x)}}$$

Results

Name	Min	Avg	Max	C	D	Avg depth
Static	2342	2653	2975	0.020	-	30.54
SP-MCTS	2411	2766	3112	0.010	0.01	30.63

Comments

We can see that SP-MCTS yields a significant improvement and seeing the results we may conclude that taking the variance into consideration is beneficial. We may also be inclined to believe that it is the properties of the variance itself that yields the improvement, but due to a mistake a discovery was made that makes one question this belief.

6.3 The Mistake - The 3rdAvg Policy

Due to an implementation error in the third term and further investigation, a new equally well (if not even better) performing tree policy was found. Because of the looks of the tree policy formula, we will refer to the scheme as *the 3rdAvg policy*.

Definition

The found policy in question chooses the child x of a parent p maximizing

$$UCT_{3rdAvg}(x) = \frac{avg_{global}(x)}{5000} + C \cdot \sqrt{\frac{\log n(p)}{n(x)}} + \frac{avg_{local}(x)}{5000}$$

Results

Apart from the results themselves, we will also present the distribution of the node depth distributions.

Name	Min	Avg	Max	C	D	Avg depth
SP-MCTS	2411	2766	3112	0.010	0.01	30.63
3rdAvg	2543	2843	3139	0.041	-	30.08

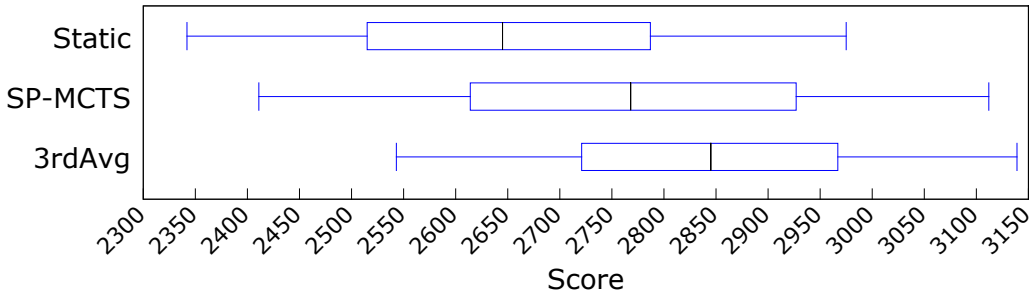

Figure 6.2. Illustration showing the average for Min, Q1, Q2, Q3 and Max for Static, SP-MCTS and 3rdAvg respectively.

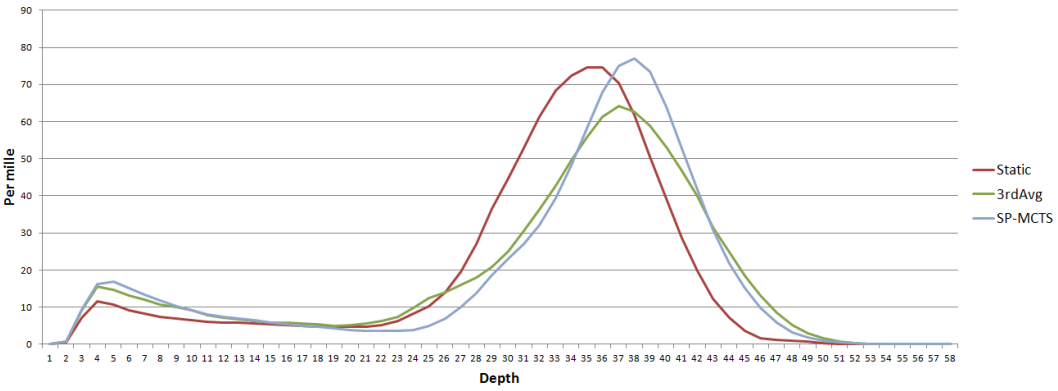

Figure 6.3. Illustration showing the distribution of nodes (in per mille) per depth for Static, 3rdAvg and SP-MCTS respectively.

Conclusions

As we can see this alternative policy is considerably better than that of SP-MCTS, and from Figure 6.2 we can clearly see that the variance (range of values) is smaller and thus yields better worst case performances. Not only are the results superior, but bookkeeping and the computational overhead of the selection policy is also

smaller. Why this policy is beneficial in the domain of MCTS and SameGame is however not entirely clear. Though we may outline some properties of the policy:

- We are generally more exploitive higher up in the search tree and more explorative lower down.
- However, the above is typically not true if the scores associated with the path traversed down are low, e.g. moves corresponding to removal of groups of size 2.
- After following an edge in the selection phase that has a high associated score, we will make a drastic shift from exploitation to exploration.

This could mean that the search is guided towards and through paths containing moves yielding a large quantity of points. These are all properties that are to some degree incorporated in the variance, although not to the same extent.

6.4 Equality of points

Before we move on to further examination of impending MCTS related question marks, we take a look at an insight that was reached through the sole use of thinking. This insight may not always result in an improvement of a SameGame algorithm, but has still an intrinsic value in its generality and possible application to other domains.

Previous attempts using UCT based algorithms to attack SameGame have all simply used the end score as described by the game mechanics internally in the search tree. However, one may argue that this is not the appropriate approach as SameGame consists of three different kinds of points.

- Points obtained through moves, i.e. removing blocks.
- Points deducted for leaving blocks at the end of the game.
- Points gained as a bonus for clearing the board.

Let say that the bonus of SameGame is increased from 1000 to 1000000, would then the present UCT based algorithms find an as good move sequence as they do when the bonus is only 1000? Most likely not. As soon as a solution clearing the board is found, the search will be relentlessly focused in this part of the search space, although there might exist some other very promising move sequence unable to clear the board at the moment that however could do so would sufficient search efforts be spent in the nearby search space. This leads to a new dilemma similar to that of exploitation and exploration, should we focus our search efforts in a part of the search space where there exists a seemingly mediocre move sequence clearing the board or should we try to find a solution clearing the board in a part of the search space where the move sequences are very promising.

Results

For the sake of completeness we examine three different settings for scaling the importance of the end game bonus in the tree policy, viz. Bonus-0.0, Bonus-0.5 and Bonus-0.8, where Bonus-x signifies that we backpropagate a score of $x \cdot 1000$ as the bonus for solutions clearing the board. Of course this is only done internally in the nodes of the search tree, the bonus of a solution is still 1000 points.

Name	Min	Avg	Max	C	Avg depth
Static	2342	2653	2975	0.020	30.54
Bonus-0.0	2287	2620	2970	0.016	29.74
Bonus-0.5	2328	2645	2979	0.018	30.96
Bonus-0.8	2352	2646	2957	0.0194	30.87

Comments

As we can see there is no significant difference between the different implementations. This is where a second insight follows, the creator of SameGame knew what he was doing and naturally he would set the end game bonus such that it is balanced with respect to the points gained by moves for a general SameGame instance. However, for an application in practice it might be the case that something corresponding to an end game bonus is not balanced in relation to the cumulative gains. In such a case the mentioned insight could be of a higher importance.

6.5 Various experiments

For the sake of completeness and the chance of obtaining some valuable insight before we move on to tackle bigger tasks, we will have a look at minor issues such as the effectiveness of *VS-pruning* and state pruning as well as usage of the maximum score and the UCB1-TUNED policy.

Definitions

All of the following variations use static normalization with 5000 and *VS-pruning* if not otherwise stated.

StatePruning Prunes a node in the search tree if the gained score together with the *over-estimator* heuristic gives a score less or equal to the best score found so far.

NoPruning Uses no *VS-pruning*.

UCB1-Tuned Uses the following formula (as mentioned in Section 3.3) in the tree policy.

$$UCT_{TUNED}(x) = \frac{avg_{global}(x)}{5000} + C \cdot \sqrt{\frac{\log n(p)}{n(x)} \cdot \min(V_T(x), 1/4)}$$

where

$$V_T(x) = \frac{V(x)}{5000^2} + \sqrt{\frac{2 \log n(p)}{n(x)}}$$

TopScore Uses the maximum score rather than the average score to guide the search according to.

$$UCT_{TOP}(x) = \frac{top_{global}(x)}{5000} + C \cdot \sqrt{\frac{\log n(p)}{n(x)}}$$

Results

Name	Min	Avg	Max	C	Avg depth
Static	2342	2653	2975	0.020	30.54
UCB1-Tuned	2324	2637	2954	0.038	30.93
NoPruning	2234	2566	2907	0.017	31.05
StatePruning	2343	2657	2996	0.020	27.01
TopScore	1969	2317	2723	0.195	29.60

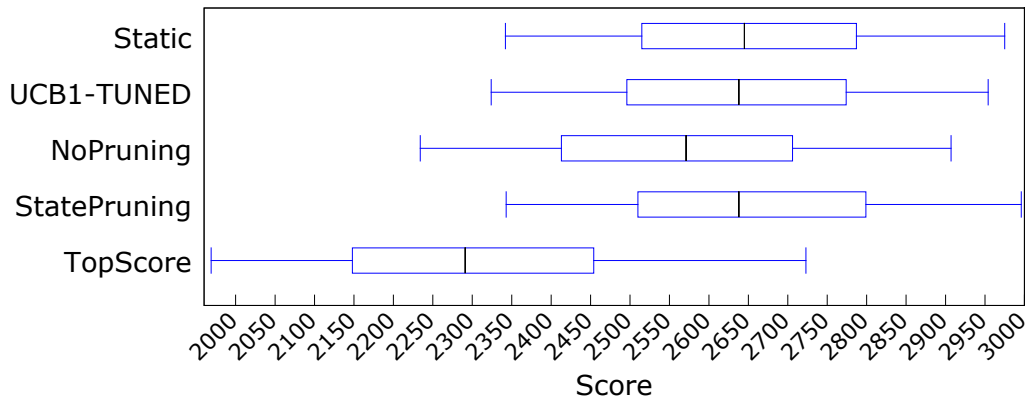

Figure 6.4. Illustration showing the average for Min, Q1, Q2, Q3 and Max for the respective policies on the Schadd test set (64×10^4 ru).

Conclusions

The most noteworthy result is the fact that we are still performing relatively well without using *VS-pruning*, although as expected worse compared to using it; though

the difference in the min category is larger than in the max category, suggesting that move pruning primarily mitigates the worst case scenarios. We also needed to use a smaller explorative factor in order to counter the increased branching factor. It might be difficult to relate to the results and determine what a drop by around 70 points (see the max category) really corresponds to. On one hand it is rather big compared to most differences we have seen so far, on the other hand it is rather insignificant compared to the improvements of the *3rdAvg* policy and SP-MCTS. We could thusly say that for SameGame, a decrease of the game-tree complexity by an order of four (c.f. section 4.1) is only worth the trouble if the effort put in is relatively small.

Furthermore UCB1-TUNED seems to give no benefit over UCB1, in fact it seems to perform worse and on top of that it is more complex which makes it a less appealing alternative. We must however, bear in mind that these conclusions are more or less limited to a short time setting, the benefits of pruning should increase over time and UCB1-TUNED could possibly be more suited for a longer time setting.

Regarding the state pruning we can see that it provides a small yet clear improvement in score. We can also see that the average node depth is reduced by 3-4 as a natural consequence of the pruning. In addition to this the memory usage is reduced, being a favorable side effect.

A bit surprisingly on the other hand is the poor performance of *TopScore*, which once and for all shows that the maximum score achieved as an indicator of the maximum score achievable from a state is a bad choice. Most likely due to the fact that it is quite unstable, partly due to the endgame bonus.

6.6 The standard test set

We are now in a position where we have examined and reflected upon a number of basic as well as fundamental aspects of MCTS and the problem domain addressed, and we have also by accident devised a variation – the *3rdAvg* policy – performing quite well. Considering that the results presented so far has used a quite small amount of resources as well as no simulation strategy (c.f. section 3.7.1) they may be regarded as quite promising. As a comparison Schadd et al. achieves a score of 2069, 2737 and 3038 points with their SP-MCTS and the simulation policies *Random*, *TabuRandom* and *TabuColorRandom* respectively, using a computational budget of 10 million in-memory search tree nodes [23].

However, it might be of interest to see what the same settings would yield on a different test set, not only for the sake of evaluating the algorithms but also comparing and evaluating the test sets. An obvious choice for this task is the standard test set.

Results

This time we will display some supplementary statistic in addition to the score values and the average node depth. The two new categories *Node* and *Terminal* denote the total number of nodes and terminal nodes in the search tree respectively, and the category *Max depth* denotes the maximum depth achieved for a node on an instance. All values presented are naturally the average over all instances of the test set.

Name	Min	Avg	Max	C/D	Avg depth	ru
Static	1053	1357	1649	0.020	8.05	64×10^4
Global	1309	1614	1907	0.053	14.76	64×10^4
SP-MCTS	1049	1355	1758	0.010/0.01	7.29	64×10^4
3rdAvg	1143	1446	1772	0.041	8.56	64×10^4
Static	1674	1979	2289	0.020	40.01	64×10^5
Global	1526	1915	2377	0.053	45.65	64×10^5
SP-MCTS	1831	2240	2625	0.010/0.01	41.83	64×10^5
3rdAvg	1956	2290	2638	0.041	41.25	64×10^5

Figure 6.5. Primary statistics on the Standard test set in a shorter (upper half) and longer (lower half) time setting respectively.

Name	Node	Terminal	Max depth	ru
Static	12450	0.03	12.77	64×10^4
Global	15493	211	23.63	64×10^4
SP-MCTS	12051	0.00	12.21	64×10^4
3rdAvg	12431	29.49	14.19	64×10^4
Static	443462	83482	49.44	64×10^5
Global	519987	106949	52.10	64×10^5
SP-MCTS	418666	76805	52.33	64×10^5
3rdAvg	364441	63562	52.51	64×10^5

Figure 6.6. Supplementary statistics on the Standard test set in a shorter (upper half) and longer (lower half) time setting respectively.

Comments

We can clearly see that the standard test set is considerably more difficult than the Schadd test set. Using the same settings, not only are the scores lower but also the average depth. This is essentially the case for almost every instance in the test set. Because of this one may question the alleged randomness of the 10 first instances [29], since they seem to exhibit vastly different properties compared to known random instances.

What was appropriate parameter settings for the Schadd test set turned out to be improper settings for the standard test set. Furthermore, when the amount of resources is tenfold the search trees get too focused around local maxima (c.f. 20% of the nodes are essentially terminal nodes), i.e., it is too exploitative, whereas in the shorter time setting it was too explorative. This clearly demonstrates that not only do we need to take the instances we are attacking into consideration when we tune our parameters but also the amount of resources we have at our disposal. Bearing this in mind one may question the whole concept of having a static explorative factor. This is all rather unsurprising, but surprisingly this is a line of thought that is rather unexplored in the field.

On a separate note we can also see that the *3rdAvg* policy uses about 10% less memory than SP-MCTS in spite of having similar values of average node depth etc.

6.7 Duplication detection

We have now come to a point where we can tackle a more complex issue, one that has easier and more efficient solutions in the single-player context, v.i.z., duplication detection. By avoiding duplicate states in the underlying structure in MCTS, the search tree is compressed into a DAG. Unlike the two-player context, in the single-player context we may remove edges from this graph without any loss of generality. More specifically we can remove edges such that a tree structure is once again imposed. There are however, various ways in which this can be done, some more advanced than others, and it remains to see which implementation gives you the most value for your effort.

Definitions

Let a and b be nodes in the search tree where b corresponds to a state reachable from a. Define $f(a, b)$ as the cumulative gained score of moving down from a to b as described by our current search tree, and $c(a, b)$ as the score of performing the associated move taking us from a to b when b is an immediate successor of a.

Do note the subtle difference between these two functions, $f(a, b)$ is only defined when b is a descendant of a in the search tree and is uniquely defined even when there are several ways of reaching b from a, whereas $c(a, b)$ is defined even if the parent node of b in the search tree is not a.

Naive If we try to expand a child node x from p_2 that has already been expanded by p_1, we cut away x from p_2 and try to expand another child node instead.

Dynamic Similar to *Naive*, but if $f(root, p_2) + c(p_2, x) > f(root, p_1) + c(p_1, x)$ we cut away x from p_1 and add it to p_2.

Consistent Similar to *Dynamic*, but this time we subtract the statistics of x from each ancestor of p_1 and add them to each ancestor of p_2.

Let a and b be ancestors of p_1 and p_2 respectively (possibly with equality). For each such a and b the following happens when a collision occurs.

$$\begin{aligned} sum_{local}(a) &\leftarrow sum_{local}(a) - n(x) \cdot (f(a, p_1) + c(p_1, x)) - sum_{local}(x) \\ n(a) &\leftarrow n(a) - n(x) \\ avg_{local}(a) &\leftarrow \frac{sum_{local}(a)}{n(a)} \end{aligned}$$

$$\begin{aligned} sum_{local}(b) &\leftarrow sum_{local}(b) + n(x) \cdot (f(b, p_2) + c(p_2, x)) + sum_{local}(x) \\ n(b) &\leftarrow n(b) + n(x) \\ avg_{local}(b) &\leftarrow \frac{sum_{local}(b)}{n(b)} \end{aligned}$$

The subtraction step is also applied to each ancestor of a node y when it gets completely solved (c.f. section 5.1). We call this *statistics retraction*.

Worth adding is that in the *Dynamic* and *Consistent* versions, when we encounter a completely solved node x to which we would improve the cumulative score, we update this information in the node. If then the cumulative score plus the best achievable score from x improves the overall best solution found, we record this solution by moving down optimally from x, store it and cease the current MCTS iteration. This solution is not recorded as statistics in any node of the search tree.

Results

All versions use static normalization and $C = 0.020$, in order to see the effect of duplication detection on different parameters such as average node depth. For the additional statistics we introduce the category *Hit-depth* denoting the average depth where we discovered a duplicate state. Additional data can be found in Section B.1.

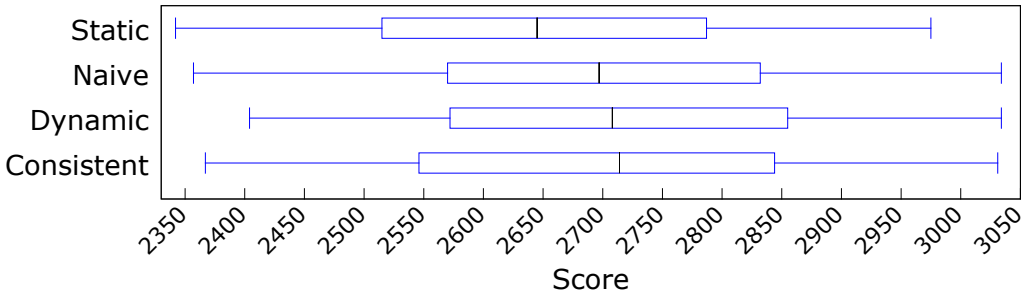

Figure 6.7. Illustration showing the average for Min, Q1, Q2, Q3 and Max for Naive, Dynamic and Consistent respectively on the Schadd test set (64×10^4).

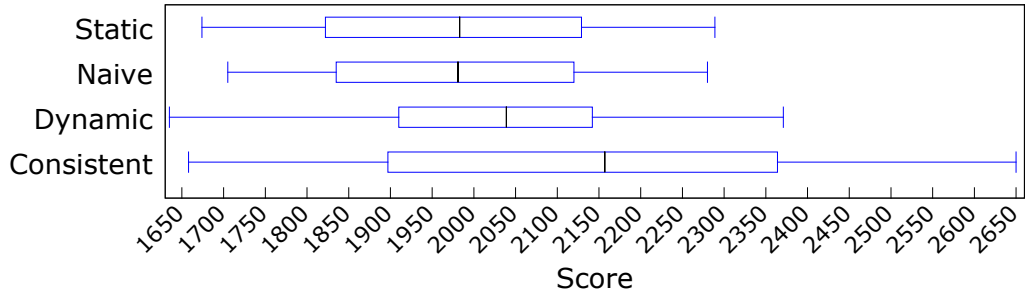

Figure 6.8. Illustration showing the average for Min, Q1, Q2, Q3 and Max for Dynamic and Consistent respectively on the Standard test set (64×10^5).

Name	Depth	Hit-depth	Max depth	Test set	ru
Naive	27.20	31.32	39.87	Schadd	64×10^4
Dynamic	27.20	30.40	39.86	Schadd	64×10^4
Consistent	26.91	29.78	39.78	Schadd	64×10^4
Naive	41.38	???	50.68	Standard	64×10^5
Dynamic	39.21	43.13	55.19	Standard	64×10^5
Consistent	39.53	43.22	55.77	Standard	64×10^5

Figure 6.9. Additional statistics on the two experiments.

Conclusions

We can see that all versions perform almost equally well in the short time setting, though *Dynamic* has a slight upper hand overall. A bit unusual is also the unsymmetrical position of Q1 and Q3 around Q2 for *Consistent* (c.f. Figure 6.7) and the comparatively low value of Q2. What is interesting however, is that avoiding duplications does not quite give the performance boost one might initially have expected, especially in the short time setting. There is a clear improvement, but it is not even near that of *3rdAvg*. This gives some perspective on the importance of a sound tree policy in MCTS-based algorithms.

Interestingly the average depth is decreased by about 3 (compared to the prescribed 30 of the reference version). This might come across as a bit unintuitive, but the *hit-depth* figure tells us that it is exactly around the previous average depth where duplicate states are typically encountered, which in turn results in nodes being forced back up to a lower depth (c.f. upper half of Figure 6.9).

In the longer time setting however, *Consistent* really outperforms the *Dynamic* (and *Naive*) version, the Q2 of the former even outperforms the Q3 of the latter (c.f. Figure 6.8), and also gives similar performance to that of *3rdAvg* although with a considerably larger range of score. From this we may possibly conclude that the usefulness of duplication detection improves as the resources grow larger, and that

being a bit inconsistent and unpredictable might be good in a short time setting but in a longer time setting structure and patience pay off.

6.8 Allocation of resources per move

One major advancement of the SP-MCTS was to allocate resources per move rather than game [26], i.e., once a certain amount of resources has been spent we choose a move once and for all and proceed our search from the resulting state. This is an idea worth investigating further in. It has been empirically shown that proceeding with the move associated with the highest maximum score is a good choice, which is one move selection policy we will examine along with a slight variation of it. Whether the search tree should be discarded or kept after a move is however unclear.

Previous research [28] has allocated resources uniformly over the first 30 moves, but it might be worthwhile to examine other allocation strategies.

Definitions

Let say that we are given a total of R ru. Then we allocate these resources over all moves $m_0, m_1...$ accordingly. Do note that we execute an additional iteration as long as there is a positive number of resources left, thus a move may slightly exceed its prescribed resources. This is done at the expense of later moves.

We consider the following resource allocation strategies.

Uniform

$$m_i = \begin{cases} R/30 & i < 30 \\ 0 & i \geq 30 \end{cases}$$

Linear Let $A = R/12.2$, $B = R/64$ and $C = R/30 - (A + B)/2$ then

$$m_i = \begin{cases} A + C - i(A - B)/(30 - 1) & i < 30 \\ 0 & i \geq 30 \end{cases}$$

Exponential If resources are left over they are assigned to move m_{30}.

$$m_i = \begin{cases} \max\left(2^{11}, R \cdot (6/7)^i \cdot 1/7\right) & i < 30 \\ 0 & i \geq 30 \end{cases}$$

The rationale behind $2^{11} = 64 \times 32$ is that it is an upper estimation of the average game length times the average branching factor, meaning that if we always make the optimal choice in the tree policy we should be able to grow the search tree to the associated terminal position.

Move selection: Once we have run out of resources for the current move m_i we must finalize it, i.e., start each future iteration of MCTS at a descendant of the node associated with the state resulting from performing the move sequence $m_0...m_i$. A

sound policy for selecting this move is to choose the one associated with the best solution found so far, i.e. $m_i \leftarrow argmax_{x \in children(p)} \left(top_{global}(x) \right)$.

However, other policies might be interesting to try out as well. One variation that we will try out is the following $m_i \leftarrow argmax_{x \in children(p)} \left(top_{local}(x) \right)$, i.e., we ignore the score of the move we are choosing and only concern ourselves with the resulting state. We will, for the sake of brevity, indicate the experiments using this policy by appending a plus sign to the name.

Results

Name	Min	Avg	Max	C	Avg depth	Test set	ru
Static	2342	2653	2975	0.020	30.54	Schadd	64×10^4
Uniform	2181	2557	2956	0.033	30.43	Schadd	64×10^4
Linear	2304	2647	3009	0.020	30.56	Schadd	64×10^4
Exponential	2378	2707	3040	0.020	29.61	Schadd	64×10^4
Uniform+	2184	2600	3005	0.032	30.43	Schadd	64×10^4
Linear+	2380	2728	3063	0.020	29.92	Schadd	64×10^4
Exponential+	2431	2751	3068	0.020	29.12	Schadd	64×10^4
Uniform	1720	2097	2438	0.033	30.47	Standard	64×10^5
Linear	1764	2078	2410	0.020	39.08	Standard	64×10^5
Exponential	1708	2081	2491	0.020	39.24	Standard	64×10^5
Uniform+	1765	2120	2484	0.032	29.38	Standard	64×10^5
Linear+	1790	2105	2467	0.020	37.47	Standard	64×10^5
Exponential+	1797	2169	2576	0.020	40.05	Standard	64×10^5

Figure 6.10. Statistics on the Schadd test set in a shorter time setting (upper half), and the Standard test set in a longer time setting (lower half).

Conclusions

With the exception of *Uniform* all variations outperform *Static* – the reference version allocating all resources at the first move – in the *max* category for the short time setting. In the two other categories the two *Uniform* versions perform notably worse, meaning that they make premature suboptimal choices early on (this also comes through in the larger range of scores for *Uniform*, c.f. Figure 6.11), a consequence not only due to the allocation strategy but also the short time setting. The *Linear* and *Exponential* versions seem to not suffer from this issue as three of them outperform the default version in all categories. For the plus versions, the two strategies perform essentially equally well in the *max* category, but it is clear that *Exponential* has significantly better worst case performance. This is most likely due to the fact that we are allocating more resources higher up in the search tree, which should lead to better stability.

An interesting fact however, is that both *Linear* and *Exponential* versions gets a lower value of average depth than or similar to that of the reference version, this

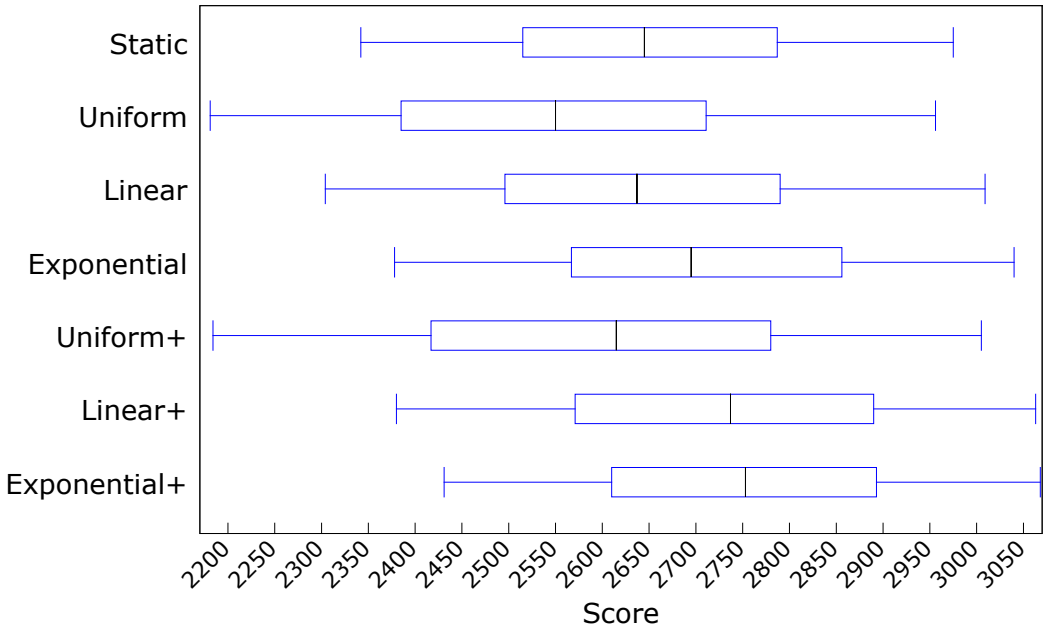

Figure 6.11. Illustration showing the average for Min, Q1, Q2, Q3 and Max for the respective policies on the Schadd test set (64×10^4 ru).

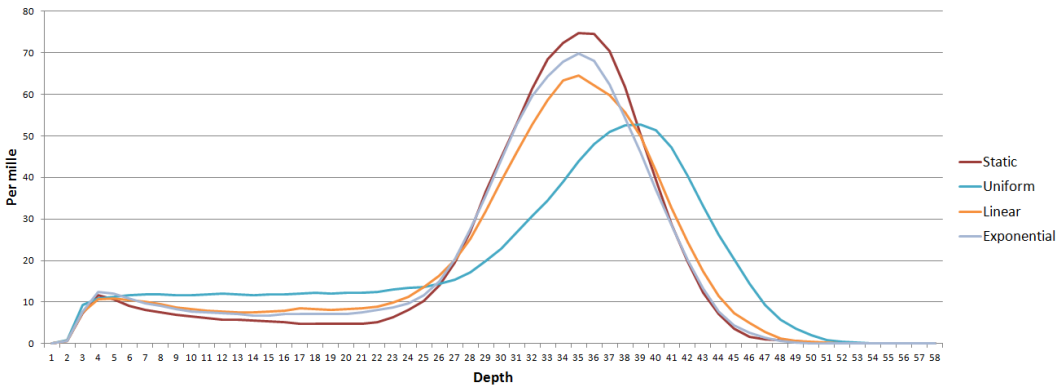

Figure 6.12. Illustration showing the distribution of nodes (in per mille) per depth for the respective policies on the Schadd test set (64×10^4 ru).

despite the fact that we are permanently moving down the tree. In spite of being counter-intuitive, this may be due to that we at some node p make a permanent choice $x = maxarg_{x \in children(p)} top(x)$ such that $x \neq maxarg_{y \in children(p)} avg(y)$, forcing us to explore a part of the search space where there are more equally promising nodes according to the tree policy. This can also be offered as an explanation to the plus versions superiority, since they are more susceptible to this phenomenon. Furthermore, the move selection policy of the plus versions will also bias parts of

the search space where there is yet a lot of points to be retrieved, typically a lot of blocks left which typically implies a greater search space, i.e., we may claim that we now also to some extent take the size of the search space into consideration in the search.

In the longer time setting the differences are smaller, although *Exponential* is clearly the best overall. However, the values of average depth are so vastly different that any conclusive conclusions are hard to make.

6.9 Dynamic explorative factor

When we ran experiments on the *Standard test set* using the same settings as the *Schadd test set*, we noticed primarily two issues. Firstly, each instance is unique and requires a different value of the explorative factor C for MCTS/UCT to work well. Secondly, in a longer time setting we can afford to be more explorative than in a shorter time setting, i.e., we must take the amount of resources into consideration as well when we set the value of C. If the value is too low we will get stuck in a local maximum and if it is too high the search tree will be too shallow and unable to discover good moves deep down.

We may refer to these two issues as *local maxima syndrome* and *shallowness*. Obviously it is impossible to have a static value of C that is suitable for all instances and time settings, thus a policy to dynamically adjust C is called for.

In this section, instead of normalizing simulation results by 5000, we will incorporate this value into C. So the previously commonly used value of $C = 0.020$ will now correspond to $C = 0.020 \cdot 5000 = 100$. This makes it easier as a human to relate to the value of C, but it could also in some cases mitigate possible technical issues such as underflow.

$$\frac{avg_{global}(x)}{5000} + C' \cdot \sqrt{\frac{\log n(p)}{n(x)}} \iff avg_{global}(x) + 5000 \cdot C' \cdot \sqrt{\frac{\log n(p)}{n(x)}}$$
$$\implies avg_{global}(x) + C \cdot \sqrt{\frac{\log n(p)}{n(x)}}$$

6.9.1 Ideal depth based approaches

One statistical indicator to identify local maxima syndrome and shallowness is the average depth of a node in the search tree. If the average depth is low we may consider decreasing C whereas if it is too high increasing C is an option. The question of what the average depth should ideally be then naturally arises.

For this sake we introduce the concept of ideal depth, a certain value we want the average node depth to have. If the average depth is below the ideal depth it is deemed too low and if it is above it is deemed too high. However, simply decreasing and increasing C indiscriminately in these cases respectively would yield a search alternating between BFS and greedy search.

Ideally we would want the average node depth to end at the ideal depth when the resources have run out. That is, there is a time aspect we ought to take into consideration as well. Estimating what the average depth should be at a given time in order to end at the ideal depth is far from trivial, therefore we will start off by introducing a simpler scheme that we will later extend to get an almost complete and satisfactory policy.

Depth block

Given an initial value of C (preferably reasonable), we keep this value as is and run the search as normal (c.f. *Static*). However, should the average node depth exceed the ideal depth we increase C by multiplying it by some number $f_1 > 1$. That is, we prevent the average depth from excessively exceeding the ideal depth, but we do not take any measures to push it there, i.e. we introduce a depth block. By doing this we will prevent the search from getting too focused around a local maximum.

In our previous experiments we tuned the constant C to yield an average depth of 30 over several instances. Since the results were rather promising this is a possible candidate for the ideal depth. Another approach is to set the ideal depth in relation to the average game length, which we can sample as we run simulations, i.e., we introduce a dynamic but yet stable ideal depth value.

Results

All versions use a value of $f_1 = 1/0.9995$ and initially $C = 100$. The version without any ideal depth is *Static* – the reference version.

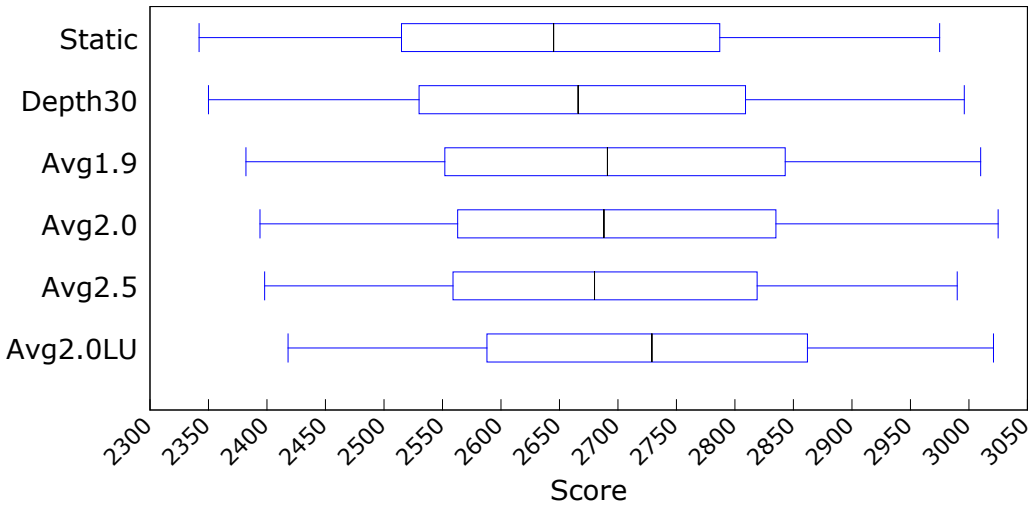

Figure 6.13. Illustration showing the average for Min, Q1, Q2, Q3 and Max for the respective policies on the Schadd test set (64×10^4 ru).

Ideal depth	Min	Avg	Max	Avg depth
None	2342	2653	2975	30.54
30	2350	2670	2996	28.27
$avg(gamelen)/1.9$	2382	2696	3010	23.03
$avg(gamelen)/2.0$	2394	2701	3025	21.16
$avg(gamelen)/2.5$	2398	2689	2990	16.39

Figure 6.14. Statistics on the Schadd test set using 64×10^4 ru.

Comments

We can see that all new versions outperform the reference version and that setting the ideal depth to 50% of the average game length seems to be optimal in terms of a depth block approach (c.f. Figure 6.13). Naturally, as a consequence of the depth block, the average node depth is lowered as well.

Late urgency

So how can we extend the depth block approach to also resolve the issue with shallowness? One indicator of shallowness is obviously if the average node depth is low, however, it is perfectly normal for the average node depth to be low initially, so the question is when can we with confidence assess that the search tree is not deep enough?

To deal with this we introduce a better late than sorry approach, that is when we only have 30% of the resources left and the average node depth is less than half of the ideal depth, then we start decreasing the explorative factor C through multiplication by $f_2 < 1$ and keep doing so as long the average node depth is below 50% of the ideal depth. This accounts for the time aspect in a lazy way in that we stop decreasing C when the half ideal depth has been reached, so that we avoid excessive exploration and let the average node depth increase steadily over the remaining time.

The policy formalized by combining *Depth block* (DB) and *Late urgency* (LU) then results in the following code snippet being executed in every iteration.

(1) **if** $R_{left} < 0.30 \cdot R_{total}$ **and** $avg\_depth < 0.50 \cdot ideal\_depth$
(2) $\quad C \leftarrow C \cdot f_2$
(3) **if** $avg\_depth > ideal\_depth$
(4) $\quad C \leftarrow C \cdot f_1$

Results

This policy yields the following results, where we compare it with the corresponding DB-version without LU. The values used are $f_1 = 1/0.9995$, $f_2 = 0.9995$ and initially $C = 100$.

LU	Ideal depth	Min	Avg	Max	Avg depth
No	$avg(gamelen)/2.0$	2394	2701	3025	21.16
Yes	$avg(gamelen)/2.0$	2418	2724	3021	22.09

Figure 6.15. Statistics on the Schadd test set using 64×10^4 ru.

Comments
Interestingly there is no improvement in the max-category, but for the two other there is an improvement. This is most likely due to that the *Late urgency* strategy is more likely to come into effect during the less successful runs. Looking at Figure 6.13, we can see that there is a clear improvement for each of the Q1, Q2 and Q3 categories as well.

6.9.2 Terminal node based approaches

An issue with the ideal depth based approaches is that there is a latency in the information the average node depth communicates. Although the average depth is low, it could be the case that we have already found a promising solution making the search too exploitative, and even if we are expanding nodes at a depth twice the ideal depth, it will take some time before the average node depth value has reached the so called ideal depth.

An alternative or complementary indicator of local maxima and shallowness is terminal nodes. If we expand a terminal node in our search tree we can conclusively say that we are too exploitative, whereas if we are soon done and have not expanded a single terminal node we may suspect that we are too explorative.

Global explorative factor

The corresponding terminal node based version to the previously seen *Depth block* is to simply increase the explorative factor whenever we expand a terminal node. This is once again done through multiplication of C by a constant $f_1 > 1$. We will call this version *LeafBlock*.

Results
LeafBlock uses $f_1 = 1/0.99$ and *DepthBlock* uses $f_1 = 1/0.9995$. Both versions have initially $C = 100$.

Name	Min	Avg	Max	Avg depth	Terminal nodes
DepthBlock	2394	2701	3025	21.16	2010.9
LeafBlock	2394	2713	3025	19.16	207.8

Figure 6.16. Statistics on the Schadd test set using 64×10^4 ru.

Comments
As we can see the results are of equivalent or marginally better quality than that of *Depth block*. The number of terminal nodes is naturally decreased as well.

LU extension

Although we now do not use the average node depth to avoid local maxima we can use the *Late urgency* (LU) strategy to mitigate issues with shallowness. An alternative version of the *Late urgency* strategy is to decrease the explorative factor when we have less than 30% resources left and there is not a single terminal node in the search tree, let us call this version *Late urgency - no terminal node* (LUntn).

Results
All versions have initially $C = 100$, $f_1 = 1/0.99$ and $f_2 = 0.9995$.

Name	Min	Avg	Max	Avg depth	Terminal nodes
LeafBlock	2394	2713	3025	19.16	207.8
LeafBlockLU	2398	2723	3048	19.61	232.6
LeafBlockLUntn	2404	2725	3048	19.54	232.0

Figure 6.17. Statistics on the Schadd test set using 64×10^4 ru.

Comments
Interestingly, while the *Late urgency* strategy yielded no improvement in the max-category for the *Ideal depth* based approaches it yields a clear improvement for *Terminal node* based ones. This can very well be due to the fact that we are in the *Terminal node* based approaches able to discover quickly when we have decreased C too much and also counter the still ongoing decreases using larger increases, though the latter is of smaller influences due to the similar performance of the second version.

Node specific explorative factor

An issue with using a single global explorative factor is that it does not account for the different needs of different branches of the search tree. When we expand a terminal node the local maximum is highly associated with the corresponding branch of the search tree, so if we increase the explorative factor for all branches, apart from mitigating an issue with a local maximum we also prevent the tree from growing deep in less explored branches. This is needless to say not a desired property.

A solution to this issue is rather than having a single explorative factor, we instead have a unique one for each node in the search tree, which is actually in line with the original definition of UCT although many implementations use a single

global one. So when we expand a terminal node in the search tree we only increase the explorative factor in all ancestors of the node. Naturally ancestors closer to the terminal node should get a larger increase than ancestors higher up in the search tree. These ideas yield the following algorithm.

terminalNodeHit($terminal\_node$, P, f, L)
(1) $current\_node \leftarrow terminal\_node$
(2) **while** $current\_node \neq NULL$
(3) $current\_node.C \leftarrow current\_node.C \times P$
(4) $P \leftarrow \max(P \cdot f, L)$
(5) $current\_node \leftarrow parent(current\_node)$

That is we start with a scaling factor $P \geq 1$ used for scaling the explorative factors in the nodes and a factor $f \leq 1$ which is used to decrease P as we move closer to the root. We also make use of a lower limit $L \geq 1$ of P, i.e., we scale the C values of all ancestors of the terminal node by at least L. However, do note that we still have to assign an initial start value of C for all nodes.

Results

The version implementing this idea will be called *CNode* for brevity. It uses $P = 1.05$, $f = 0.999$ and $L = 1.0$. All nodes will initially have $C = 100$.

Name	Min	Avg	Max	Avg depth	Terminal nodes
Static	2342	2653	2975	30.54	12106.0
LeafBlock	2394	2713	3025	19.16	207.8
CNode	2451	2767	3079	19.97	97.4

Figure 6.18. Statistics on the Schadd test set using 64×10^4 ru.

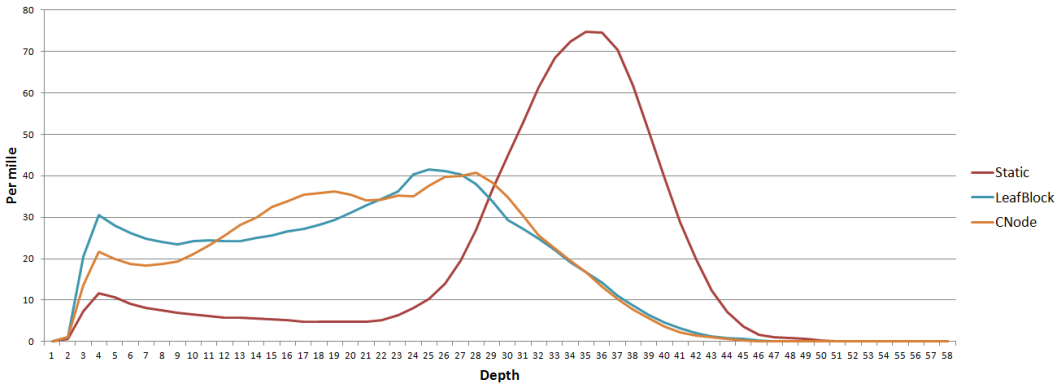

Figure 6.19. Illustration showing the distribution of nodes (in per mille) per depth for Static, LeafBlock and CNode respectively.

Comments
We can see that using this strategy we get a significant increase in performance, about 100 points more than the reference version, and this in spite of not applying any techniques to avoid shallowness. Moreover, compared to *LeafBlock* the number of terminal nodes is significantly decreased (more than halved) despite having a higher average node depth, which is arguably a healthy sign.

Mitigating shallowness

To apply techniques of avoiding shallowness to approaches having node specific C is not trivial. Intuitively we would like to be more exploitative in rarely visited branches, especially when the resources are about to run out.

A more straightforward and possibly more naïve application of *Late urgency* is simply to apply it to each node individually, i.e., if less than 30% of the resources are remaining and there is no terminal node in the subtree rooted at the specific node, then we decrease the value of C each time we visit the node.

Results
The version using this strategy will be named *CNodeLU* for brevity. The results can be found in Figure 6.20.

Comments
Do note that this strategy implies that often visited branches (without any terminal node) will have a faster decrease of the explorative factor, which is in contrast with our initial intuition. However, as we can see, this gives about the same improvement as it did for the *Ideal depth* based approaches. The values for Min, Q1 and Q2 are essentially the same, but both Q3 and Max can enjoy a boost (c.f. Figure 6.21).

Rapid start

Due to the difficulty of estimating when we suffer from shallowness, we might consider a quite different take on the issue. Instead of trying to initiate the search using an explorative factor being reasonable or a bit on the high side, we can start off exploitative and start increasing the C values once we identify local maxima. This way we do not need to worry about any issues with shallowness in any branch of the search tree. Furthermore, setting C to get exploitation does not require any prior domain specific knowledge.

Results
Both *CNodeLU* and *Rapid* uses $P = 1.05$, $f = 0.999$ and $L = 1.001$. The latter has however $C = 4$ initially for all nodes whereas the former uses $C = 100$.

Name	Min	Avg	Max	Avg depth	Terminal nodes
CNode	2451	2767	3079	19.97	97.4
CNodeLU	2453	2774	3100	20.03	103.3
Rapid	2368	2713	3054	18.65	311.5

Figure 6.20. Statistics on the Schadd test set using 64×10^4 ru.

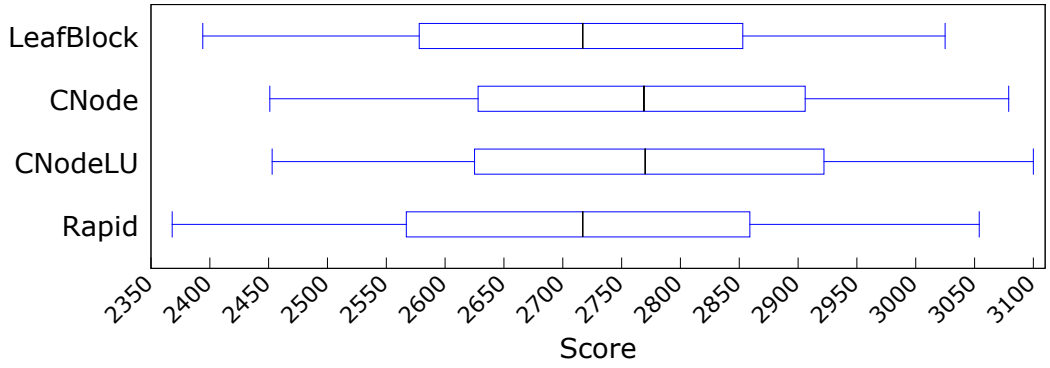

Figure 6.21. Illustration showing the average for Min, Q1, Q2, Q3 and Max for the respective policies on the Schadd test set (64×10^4 ru).

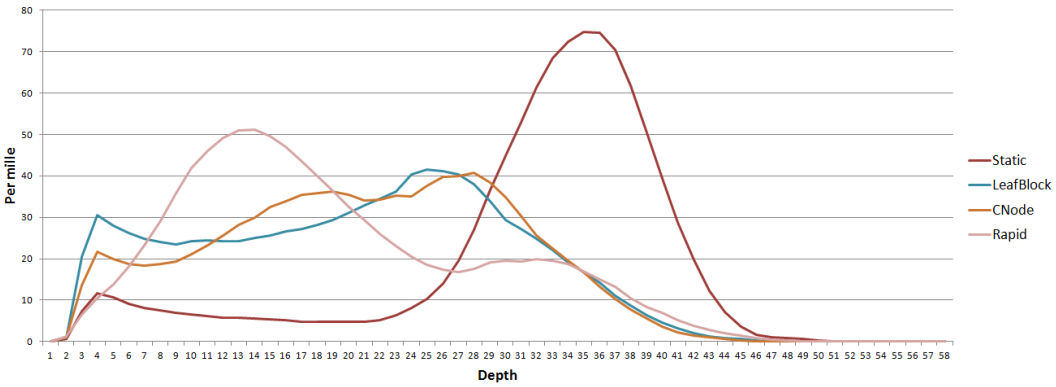

Figure 6.22. Illustration showing the distribution of nodes (in per mille) per depth for Static, LeafBlock, CNode and Rapid respectively.

Comments

As we can see the results are not favorable. This could be due to that we end up focused around a local maximum too early on. Despite the early exploitativeness the average node depth is lower than the corresponding version starting with a reasonable value of C, which is a bit counter intuitive. However, this might be related to the fact that we are finding inferior solutions. Because if we start of

exploitative we are more likely to find solutions with shorter move sequences, and thus converge around a shallower local maximum and consequently get a lower average node depth.

These results give us two insights. Firstly, the initial value of the explorative factor has a considerable impact on the performance of the search algorithm. Secondly, although *Rapid* yields in comparison inferior solutions, it still works reasonably well and could be a competitive alternative in domains where knowledge is scarce.

Duplication detection and node specific explorative factor

Applying the technique of node specific explorative factors to a version of MCTS that avoids duplicate states in the search tree is a bit more complicated in order to achieve good performance, since the number of terminal nodes in such a tree is drastically decreased, e.g. there can only be one node representing the cleared board which otherwise would be a quite frequent terminal position. Thus we need to find an alternative condition for when we should increase the explorative factor in a set of nodes.

One way of doing this is trying to imagine when a terminal node hit would occur should the duplication detection not be present. This requires just a slight modification of the original approach, which is if the most recent child node we expanded was already marked as completely solved (i.e. we found it in the transposition table for duplication detection rather than created a new node) we know that there is an execution of *terminalNodeHit* passing through our node that has not been accounted for. Therefore we accommodate for this by running *terminalNodeHit* starting from our current node, which will also result in a slightly more and more aggressive increase of the C values as larger branches of the search tree become completely solved.

Do note that the there will exist some set of nodes where the update associated with one specific terminal node is accounted for more than once, but this is all in order since the mentioned node will exist in several locations in a search tree without duplication detection. However, in some cases we might miss to account for a terminal node hit in some possible parent node of it, e.g. if some child nodes have been expanded but then taken over (in terms of ownership) from other nodes and these child nodes then become completely solved. Upon the next visit in the originally considered parent node, it will be impossible to tell whether these (all of a sudden) completely solved child nodes have been accounted for or not without any extra bookkeeping, a bookkeeping which is hardly likely to be worthwhile. However, if we in this situation discover that all child nodes have been completely solved (and thus also the parent node itself) we may run *terminalNodeHit* once from the node.

Results

We will for the moment refer to this version as *ConsistentCNode*. It uses $P = 1.05$, $f = 0.999$ and $L = 1.001$, and all nodes start with a value of $C = 100$.

Name	Min	Avg	Max	Avg depth	Terminal nodes
Static	2342	2653	2975	30.54	12106.0
CNode	2451	2767	3079	19.97	97.4
CNodeLU	2453	2774	3100	20.03	103.3
Consistent	2367	2700	3031	26.91	326.9
ConsistentCNode	2503	2842	3164	16.42	19.4

Figure 6.23. Statistics on the Schadd test set using 64×10^4 ru.

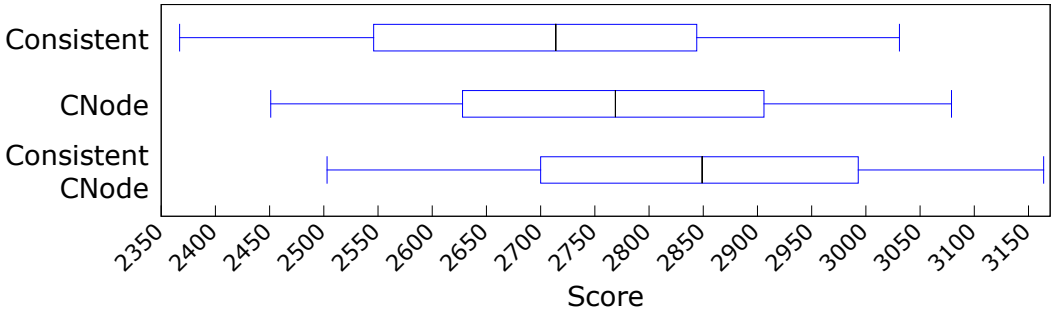

Figure 6.24. Illustration showing the average for Min, Q1, Q2, Q3 and Max for Consistent, CNode and ConsistentCNode respectively.

Comments

We can see that combining *CNode* with duplication detection in this way boosts the performance maybe more than one would have initially expected. A key stone to the performance boost may be the successively more aggressive increase of C values that take place, which could explain the remarkably low number of terminal nodes on average. From Figure 6.24 we can tell that it is an overall improvement and that the improvement is more or less the same for each of the five categories.

Combining CNode and SA-UCT

The main idea of SA-UCT prescribes that we should keep decreasing C over time. However, if we have reached a point where we have started creating terminal nodes, then there is little meaning in decreasing the explorative factor of concerned nodes any further. Therefore combining SA-UCT with *CNode* could be a fruitful approach.

The modification is simple. We maintain a global value C_{start}, at each iteration when we encounter a node, we set its individual C value to that of $T(C_{start}, R_{left}, R_{total})$ unless a terminal node hit has occurred in the subtree rooted at the node (when we combine this with duplication detection, we generally mean "in any subtree that has been rooted at the node"). In this way we will keep decreasing the explorative factor of all nodes equally much until a terminal node is expanded upon which some nodes will start getting their explorative factor increased instead.

The original formulation of SA-UCT only considers a linear decrease of the explorative factor. However, if the initial value of C_{start} is large it is likely that spending more than 50% of our resources using an explorative factor less than $C_{start}/2$ will yield more promising results compared to a 50/50 distribution around $C_{start}/2$. We therefore will define a simple exponential scheme to compare aside the linear scheme where we also prevent the explorative factor from getting too low.

Linear

$$T(C_{start}, R_{left}, R_{total}) = C_{start} \cdot \frac{R_{left}}{R_{total}}$$

Exponential

$$T(C_{start}, R_{left}, R_{total}) = \max\left(C_{start} \cdot e^{-5 \cdot \frac{R_{total} - R_{left}}{R_{total}}}, 8\right)$$

Results

All versions use $P = 1.05$, $f = 0.999$ and $L = 1.001$. The numeric suffix of the abbreviation of the name indicates the value of C_{start} being used.

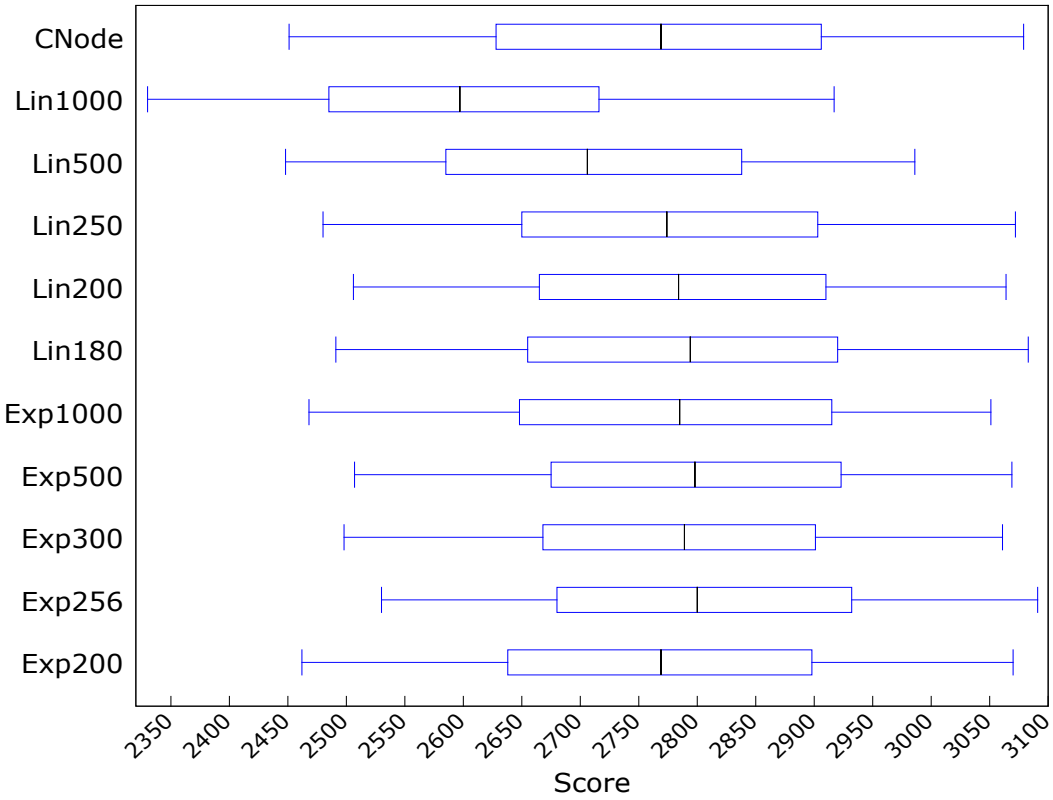

Figure 6.25. Illustration showing the average for Min, Q1, Q2, Q3 and Max for the respective policies on the Schadd test set (64×10^4 ru).

Comments
Sadly neither of the two present any major improvement over the original *CNode* or *CNodeLU*. However, both perform relatively well with the exception of *Linear* using a too large C_{start} value. Especially the *Exponential* scheme has an overall good performance regardless of C_{start}, which may render it a viable alternative in domains where it is difficult to manually tune the explorative factor. The worst case performance is also for some variations clearly improved, resulting in a smaller range of values (c.f. Figure 6.25).

6.9.3 Final remarks

Looking at the still substantial discrepancy in score between the min- and max-categories, we may conclude that none of the techniques presented in this section solves the issue of getting stuck in local maxima completely. The overall increase in score and decrease of number of terminal nodes however, suggests that the issue is at least mitigated and would also most likely improve upon a naïve meta-search approach using random restarts. The results of *CNodeLU*, *Rapid start* and the various schemes of SA-UCT suggest on the other hand that the initial choice of the explorative factor has significant impact on the performance. This initial choice is not only limited to a global choice, but can also be done individually per node based on some parameters e.g. the depth or the parent node.

6.10 Deepened insights

The analysis of some of the concepts presented so far in this chapter only scratched on the surface of why they perform differently. In this section we will give a more thorough understanding of the mechanics working behind the scene.

6.10.1 3rdAvg

One obvious question is "why does the 3rdAvg policy perform so well?". We have already outlined similarities between this policy and the standard deviation, but the main point behind the success remains to be unveiled.

An interesting aspect of the 3rdAvg policy is that it in the third term does not take the score of the move leading to the child state into consideration. If this is bad or not has yet to be established, let us define a policy called 3rdAvg+ that also incorporates the score of the move into the third term as follows.

$$UCT(x) = \frac{avg_{global}(x)}{5000} + C \cdot \sqrt{\frac{\log n(p)}{n(x)}} + \frac{c(p,x) + avg_{local}(x)}{5000}$$

Results

More precise figures can be found in Section B.3.

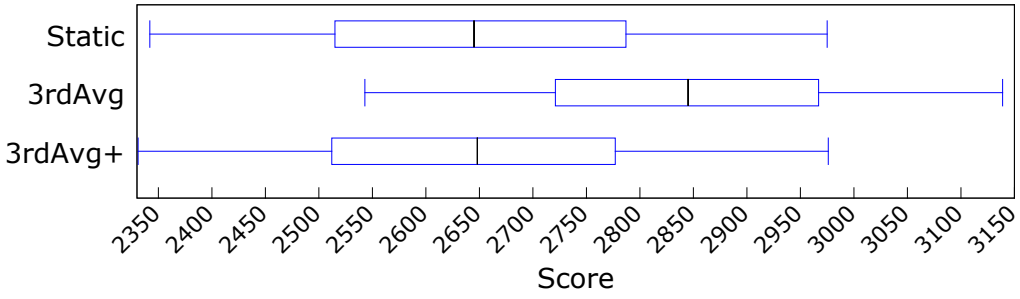

Figure 6.26. Illustration showing the average for Min, Q1, Q2, Q3 and Max for the respective policies on the Schadd test set (64×10^4 ru).

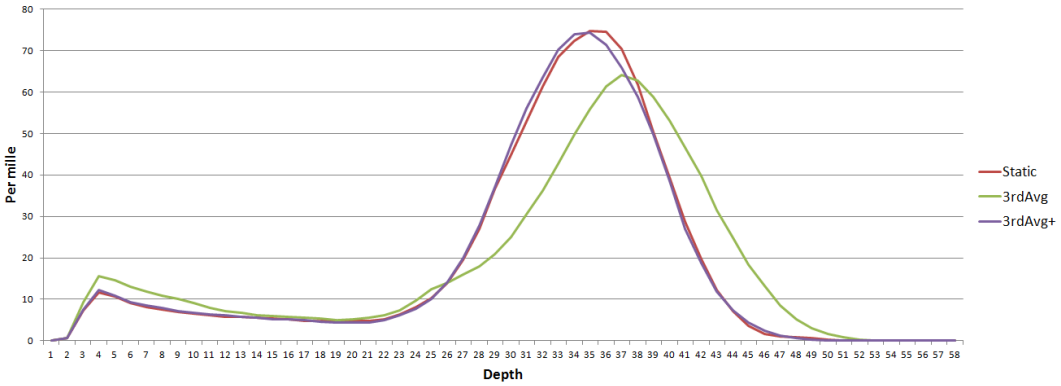

Figure 6.27. Illustration showing the distribution of nodes (in per mille) per depth for Static, 3rdAvg and 3rdAvg+ respectively.

Comments

The change may have seemed small and insignificant at first glance, but the results of it are rather striking. We can see that this version performs at the same level as *Static*, i.e. significantly worse than the original *3rdAvg* policy (c.f. Figure 6.26). These results show that the whole performance gain of *3rdAvg* lies in the omission of the score of the move. This is because by doing so, we will focus on the moves keeping the largest group of blocks intact alternatively making it larger, since the score of removing the large block will only have influence in the third term for the child states where it is still available as a move. In other words we have incorporated to some extent domain specific knowledge in the tree policy in a quite subtle but yet elegant way.

An interesting observation is that looking at the node distribution per depth for *Static* and *3rdAvg+*, although being different policies with different values of C, the curves of the two almost coincide, possibly indicating some relation between node distribution and score.

6.10.2 TopScore

A question possibly arising is, if we are interested in the maximum value rather than the average value, why do we not use the former to guide the search? To answer this concern with confidence an experimental examination is required.

$$UCT(x) = \frac{top_{global}(x)}{5000} + C \cdot \sqrt{\frac{\log n(p)}{n(x)}}$$

As we saw in Section 6.5 using the maximum score to guide the search performs surprisingly poorly. This could be due to the high variance and drastic shift in the maximum score as a solution clearing the board is found. However, we could apply our earlier insight of ignoring the endgame bonus in order to possibly achieve a better performance (these versions will be suffixed with a plus sign).

Results

Name	Min	Avg	Max	C	Avg depth
Static	2342	2653	2975	0.020	30.54
StaticTop	1969	2317	2723	0.195	29.60
StaticTop+	2329	2750	3138	0.120	24.88
StaticTop+	2256	2675	3054	0.090	29.79

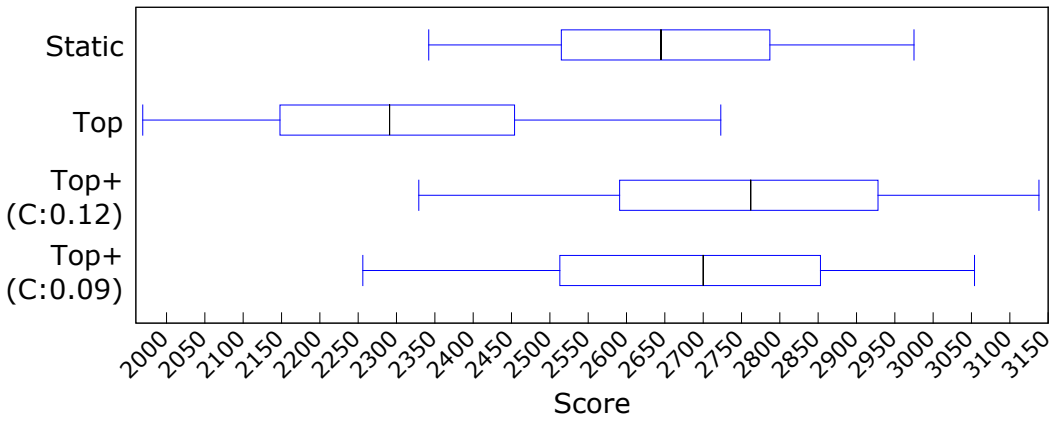

Figure 6.28. Illustration showing the average for Min, Q1, Q2, Q3 and Max for the respective policies on the Schadd test set (64×10^4 ru).

Comments

Shokingly this does not only yield an improvement, but the new version even outperforms the corresponding version (*Static*) using the average score to guide the

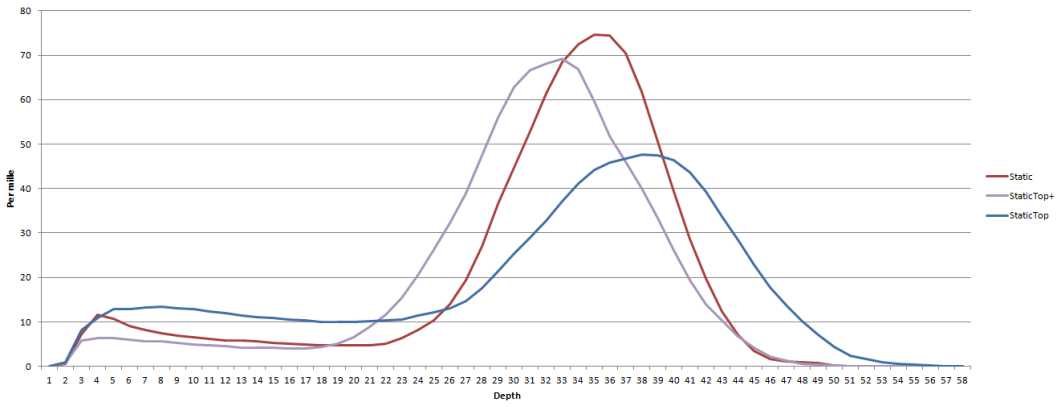

Figure 6.29. Illustration showing the distribution of nodes (in per mille) per depth for Static, StaticTop+ ($C = 0.090$) and StaticTop respectively.

search. The worst case perfomance is worse, but the range of score is larger, resulting in a better top performance. This shows that our insight that was previously fruitless resulted in a significant performance boost for a different score measure with a different distribution, and consequently we have established its importance.

Combining TopScore and 3rdAvg

One question concerns in what way we can combine this with the previous *3rdAvg* policy. One variation is to simply add a third term corresponding to the best score achievable from the considered child node, let us call this *3rdTop*. We also introduce a variation called *Top3rdAvg* where we add the average score as the third term, but in the average score we also take the endgame bonus into consideration.

Results

Name	Min	Avg	Max	C	Avg depth
3rdTop	2268	2664	3048	0.195	31.43
3rdTop	2319	2699	3053	0.210	29.88
Top3rdAvg	2408	2782	3123	0.141	30.94
Top3rdAvg	2365	2786	3145	0.148	30.06

Comments

As we can see none of the so called “third term” strategies give any massive performance boost, as was the case with the original *3rdAvg* policy. This could be because much of the benefits provided by such a strategy (focus on creating large groups) is already to some extent covered by ignoring the endgame bonus and using the maximum score to guide the search.

Chapter 7

Combining techniques

Up until now, we have largely focused on trying out a myriad of techniques and ideas in isolation, and evaluating their performance. The results of section 6.10.2 however, gave us one insight, namely that trying out combination of techniques is worthwhile, even when the techniques themselves individually have yielded inferior results. So in order to achieve top performance the following chapter will adress the issue of combining ideas. Typically we would aim to combine the best techniques (as outlined by the experimental results of Chapter 6), though it might be the case that some techniques work poorly together, so some afterthought is required in the experimental process.

The natural version to build upon is the one from section 6.9.2 combining duplication detection and CNode using the LU extension, since these two are techniques that first of all worked well together and second of all are quite general in their nature and thus unlikely to affect other techniques negatively. For the sake of brevity we will call this version *Base*.

7.0.3 Adding the 3rdAvg policy

The first extension to this base version will be the incorporation of the 3rdAvg policy, which in all is a straightforward task.

Results

Name	Min	Avg	Max	C	Avg depth
Base	2503	2842	3164	0.020	16.42
3rdAvg	2543	2843	3139	0.041	30.08
Base3rdAvg	2735	3003	3262	0.041	17.07

Comments

As we can see the overall improvement is significant, but perhaps not as much as expected in the max-category. This may be because there is a limit on how good solutions can get before they get very hard if not impossible to improve. Consequently the best-case performance improves less than the worst case performance.

7.0.4 Adding allocation per move

Incorporating allocation per move together with duplication detection as done according to the consistent strategy is however not as easy as it first may sound, especially if it is done using a so called plus strategy (as defined in Section 6.8). Some special cases one must bear in mind are the following:

- There might exist several equally good solutions and in some cases their solution strings may have a shared suffix. But using duplication detection every node has only one parent or owner, thus it is possible that we traverse down a branch from which the so far best known solution cannot be exploited.
- Using a plus strategy it might be the case that the starting node of iterations is the parent of no child in the search tree, since it is possible that we have chosen a suboptimal path.
- If too few resources are allocated for a series of moves, we might have to deal with null-children.
- When we get deeper down we might have to deal with completely solved children.

The latter two can be quite straightforwardly handled, but regarding the former two which are strongly linked to duplication detection there is room for various design choices. In our solution we have solved issue one by simply changing the parent and ownership of the child node in question, and issue two is solved by undoing the last "final move choice", i.e. we start iterations at a lower depth.

Results

Name	Min	Avg	Max	C	Avg depth
Base3rdAvg	2735	3003	3262	0.041	17.07
Base3rdAvgExp+	2722	3026	3312	0.041	19.26

Comments

As we can see performance in the max category is clearly improved, so also for the avg category although not to the same extent. On the other hand worst case

performance seems to be made worse, this is in all likelihood due to what we have previously assessed about allocation per move strategies, that premature choices may happen to a larger extent.

7.0.5 Simulation policy

A quite important and effective technique to improve MCTS based solvers, that we have neglected so far in this work, is the use of a enhanced simulation policy also known as a simulation strategy. A simple yet quite effective simulation policy for SameGame is *TabuColorRandom* (section 3.7.1). Experiments using this simulation policy will be suffixed by "Tabu" in their name.

Since there is reason to believe that much of the benefits granted by the 3rdAvg policy also comes with a simulation policy focusing on larger groups, it is also wise to try a variation using the ordinary UCB as tree policy.

Results

For the sake of space in the graph the following abbreviations will be used *B3A* (Base3rdAvg), *B3AE+* (Base3rdAvgExp+), *B3AE+T* (Base3rdAvgExp+Tabu) and *BE+T* (BaseExp+Tabu). More detailed data can be found in Section B.4.

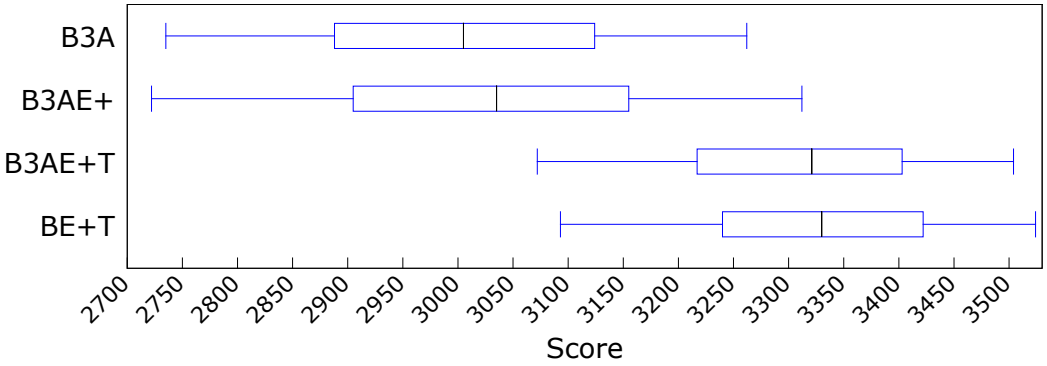

Figure 7.1. Illustration showing the average for Min, Q1, Q2, Q3 and Max for the respective policies on the Schadd test set (64×10^4 ru).

Comments

Partially as a surprise we can see that the version using ordinary UCB has superior performance, which confirms our initial suspicion of the 3rdAvg policy working badly together with a simulation strategy focusing on larger groups, since they do much of the same work. And in the case of such a simulation strategy, a tree policy more focused on clearance of the board is preferable. Furthermore we can see that the simulation strategy boost the performance considerably, though not as much as 50% [23], which may be due to the fact that the method was already quite competitive.

Interestingly the scope of score values is also considerably smaller, suggesting a more reliable and predictable performance (c.f. Figure 7.1).

7.0.6 Augmenting the simulation strategy

Inspired by [17] we propose an easily implementable slight modification of the *TabuColorRandom* strategy, namely switching to a complete random simulation when the number of blocks falls below a certain threshold. So if we set the threshold to x, we will refrain from using the simulation strategy when only x blocks or fewer are left. The rationale for this is that it could possibly increase our chances of clearing the board.

Results

In the experiments we will suffix the name of the method by the number of blocks used as a threshold. And once again for the sake of space we will use an abbreviation, this time *Tabu* in place of BaseExp+Tabu. Additional detailed data can be found in Section B.5.

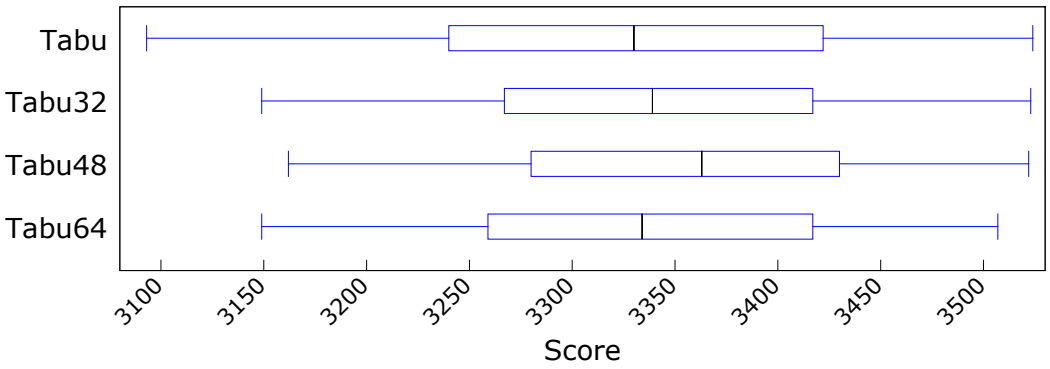

Figure 7.2. Illustration showing the average for Min, Q1, Q2, Q3 and Max for the respective policies on the Schadd test set (64×10^4 ru).

Comments

In the results of Matsumoto et al. [17] no big difference between their similar variation and the original *TabuColorRandom* was seen. This is also the case for us, in the max-category. In the min-category however, we can see that there is a significant boost in performance, which naturally influences the results of the avg-category. We can also see that one should be careful not to set the threshold too high, which may have a negative impact on best case performance. The results of the min-category tell us that using a threshold, the probability of clearing the board is likely increased, whereas the results of the max-category tell us that in the best case there is no need for such an extension.

Chapter 8

Epilouge

8.1 Comparison

In order to assess the level of contribution of the work, we will compare some of the methods presented with previous research.

8.1.1 Schadd test set

Apart from the creators of the set, Baier et al. [5] has also used it to evaluate their NMCTS implementation. However, for the evaluation of NMCTS only the 100 first levels were used, but since the levels are randomly generated the average score on all 250 levels should be comparable to the average score on merely the first 100 ones. Alongside a level-2 NMCTS implementation, a MCTS variation using randomized restarts was also evaluated.

In their experiments they have tried various number of restarts for the MCTS implementation and various numbers of calls to level-1 NMCTS for the level-2 NMCTS implementation. The results presented here will be the best ones over all settings. They also present separate results depending on whether the simulations have been purely random or used their enhanced simulation strategy based on *TabuColorRandom*. Each position was given a computational time of 9120 seconds (approximately 2.5 hours).

Results

In the comparisons we will use both the result presented as "Avg" and "Max" for our work, but for the latter we will append the suffix "(Meta 5)" since it is essentially a meta search taking the best result from five independent runs. For the case of random simulations, we have also included a method not relying on the 3rdAvg policy, since one may argue it is unfair to incorporate game knowledge in such a way.

The methods in this work do not rely on time as a resource measurement but so called resource units. For the Schadd test set 64×10^4 ru has been exclusively

used throughout. This corresponds roughly to just a few seconds of computational time for a reasonable computer as of year 2014.

Name	Avg	Computational time
MCTS (Meta 2280)	<2700	9120 sec
NMCTS	<2900	9120 sec
Base3rdAvgExp+	3026	<10 sec
Base3rdAvgExp+ (Meta 5)	3312	<50 sec
ConsistentCNode	2842	<10 sec
ConsistentCNode (Meta 5)	3164	<50 sec

Figure 8.1. Comparison of methods using random simulations. The upper half is due to Baier et al., the lower half is from this work. (The time is per instance.)

Name	Avg	Computational time
MCTS (Meta 2280)	3396	9120 sec
NMCTS	3466	9120 sec
BaseExp+Tabu48	3351	<10 sec
BaseExp+Tabu48 (Meta 5)	3522	<50 sec

Figure 8.2. Comparison of methods using weighted simulations. The upper half is due to Baier et al., the lower half is from this work.

Comments

As we can see the methods of this work clearly outperforms the ones presented in [5]. Not only do we generally achieve a higher score, but we do this in literally just a fraction of the time. Interestingly the methods of this work are more superior in the case of random simulations than in the case of weighted simulations. A possible explanation for this is that a considerable part of the development of the methods was done using random simulations, whereas weighted simulations was added relatively late on to the work.

8.1.2 Standard test set

The most commonly used test set among previous research is the so called standard test set. Although the number of instances is small and the number of resources used by previous research is unknown, the vast number of results available for comparison makes it an interesting test set to use for evaluation.

The method up for comparison from this work will be the same as in the previous section, i.e., *BaseExp+Tabu48*, but this time we will use an initial value of $C = 0.022$. We will run two separate executions each using a total of 448×10^7 ru, but the first will divide them uniformly over 16 independent runs whereas the second

one divides them over 14 runs (both 14 and 16 yield an amount of resources per run that was deemed suitable during pre-testing), so that we get two similar versions that ought to be comparable in performance, and for each of them we pick the run yielding the highest score as our solution. After each run we will also reset the seed used by the random number generator to a new one.

Results

For naming purposes the first version will be referred to as *448/16* and the second version as *448/14*. As an interesting comparison we will also introduce a column *Max* taking the best final solution provided by the two.

Furthermore, since the Standard test set consists of merely 20 instances, it allows us to present the results individually for each instance and thus give us a deeper understanding of the performance.

Instance	448/16	448/14	Max	SP-MCTS	MC-RWS	NMCS
1	2731	2835	2835	2919	2633	3121
2	3843	3853	3853	3797	3755	3813
3	3487	3369	3487	3243	3167	3085
4	3825	3799	3825	3687	3795	3697
5	3969	3741	3969	4067	3943	4055
6	4419	4719	4719	4269	4179	4459
7	2945	3025	3025	2949	2971	2949
8	4261	4053	4261	4043	3935	3999
9	4861	4859	4861	4769	4707	4695
10	3253	3519	3519	3245	3239	3223
11	3625	3269	3625	3259	3327	3147
12	3181	3171	3181	3245	3281	3201
13	3337	3243	3337	3211	3379	3197
14	2833	2899	2899	2937	2697	2799
15	3613	3565	3613	3434	3399	3677
16	5151	5109	5151	5117	4935	4979
17	4661	4853	4853	4959	4737	4919
18	4993	5185	5185	5151	5133	5201
19	5027	4971	5027	4803	4903	4883
20	4921	4897	4921	4999	4649	4835
Avg	3947	3947	4007	3901	3838	3897
Total	78936	78934	80146	78012	76764	77934

Figure 8.3. The respective score on each instance as well as the average and total score on the test set as a whole. To the left is the results of this work, and to the right is the results of previous research. The results of the competitors' approaches has been fetched from [26].

Comments

Sadly we do not reach our goal of 80000 in a single execution, although we are not too far off, and by combining the best results of the two executions we manage to achieve the desired score. Thus we may conclude that using more resources would probably make us reach our goal in a single execution. Allocating the resources over a different number of multiple independent runs could possibly also affect the performance, but looking at the total score of *448/16* and *448/14* which are uncannily close, this must be considered unlikely (though 14 and 16 are after all close in absolute terms). Luckliy the two executions produce quite different results on some instances, which beneficially influences the maximum of the two. And comparing the results of *Max* to SP-MCTS we can see that there is still room for improvement on some instances.

Interestingly the results of SP-MCTS were obtained using 1000 independent runs each using 100000 in-memory nodes (where each node must be simulated 11 times before it can expand child nodes), in total corresponding to about $1000 \cdot 100000 \cdot 11 \cdot I$ ru where I is the expected number of inspected states per simulation (let us say 20 as an underestimation), i.e., approximately 2200×10^7 ru. In contrast we only divide our resources over 16 and 14 runs respectively, which could imply that thanks to the methods developed in this work we perform better during long time runs compared to SP-MCTS. Needless to say, the number of resources used is also (by estimation) considerably lower, even when we consider our two executions combined. Therefore it is fair to consider the maximum of the two as a single execution in comparison to other algorithms.

There are in fact algorithms achieving better results than those presented in this work (see Section 2.2.3). However, they are all rather badly (or not at all) documented or presumably quite resource intensive. In that respect we believe that the methods presented in this work have made a quite tangible contribution to the field of MCTS-based algorithms.

8.2 Contribution and Conclusion

A quite tangible contribution of this work, although of small scientific value, is an analysis of SameGame. A new take on the concept of complexity is applied, with a new figure for the state-space complexity (although being a very rough upper bound) and deeper insight of the typical game-tree complexity. Furthermore, as far as this work is concerned, a yet undocumented pruning technique for moves was introduced, which is evaluated using the concept of game-tree complexity and empirically shown to be quite fruitful.

Although not finally used, the introduced 3rdAvg policy shows the power of a tree policy in order to harness the power of randomness in the search. However, as for SameGame, the power of a simulation policy is superior. And we have seen that there is a risk that a tree policy improving performance over UCT using purely random simulations might be inferior to UCT in the case of weighted simulations.

Due to the similarities between the 3rdAvg policy and SP-MCTS, one may suspect that SP-MCTS too is inferior to UCT in combination with the *TabuColorRandom* simulation policy, though one should be careful of jumping to conclusions and to such a great deal dismiss the usefulness of previous innovations in the field.

One of the main points of originality of this work lies in the usage of a dynamic explorative factor. There are very few papers out there describing the use of an individual C per node and even fewer describing a way to dynamically adjust them. Few papers even adopts the concept of a dynamic value at all, in most cases a single static global explorative factor is the rule. From that point of view, this work is fairly unique. Furthermore we manage to apply the idea successfully resulting in improved performance. Already present and future work could probably apply something similar to enhance their results, and this work shows that it is possible and worth the effort trying.

The techniques related to a dynamic explorative factor has not always increased the best case performance, but almost always enhanced worst case performance. The majority of previous research in the field, has not concerned itself with worst case performance although some interesting findings may be done. An example of that is presented in this work is the modification of the *TabuColorRandom* simulation strategy such that it swaps to purely random simulation after a time, which significantly improved worst case performance. Another technique developed in this area was the combination of *CNode* and SA-UCT, where we for the latter experimented with three different policies of decreasing the explorative factor over time.

Integration of many different techniques has been one of many challenges addressed in this work. In isolation a technique may be straightforward to implement, but when combined with others special cases, sometimes only occurring at rare occasions, may arise. An example of this is the, for the single player context exclusively and newly developed, duplication detection combined with MCTS, which has turned out to then in turn not be completely easily to integrate with state pruning and the various strategies for a per move allocation of resources that has been examined in this work.

A big part of the contribution lies however not in the integration of various techniques, but in the sheer examination of basic MCTS related concepts and ideas in the single player context. Most experiments related to the basics are however conducted under a short time setting, meaning that the conclusions may not necessarily extend to a longer time setting in all cases. But the results provided should at least provide a hint for future researchers which techniques that may be worth trying out for their problem.

In relation to the basics some parts of the previous research that seemed illogical or contradictory was brought up for discussion and questioned, although related experiments were not always conducted. One concrete example of rejection of information from previous research following from experiments is the alleged randomness of the 10 first levels of the standard test set. They might be handpicked as difficult levels from a set of random levels, but the likelihood of them being simply randomly generated must be considered as low. We also questioned previous

experimental procedure of only using a single run per instance, and fundamentals such as how to adopt the target domain i.e. SameGame to UCT and usage of the acquired statistics, from which the ideas of ignoring the end game bonus and using the maximum score instead of the average score was conceived. Two ideas which independently provided no performance gain but combined boosted performance significantly, an insight that can be of value for other similar domains lacking any suitable simulation heuristics.

8.3 Future research

One of the main issues discovered and addressed in this work has been the explorative factor C. We have seen that different values for different parts of the search tree has been beneficial. We have also seen that the initial value chosen for these different explorative factors affects the performance, even if we adopt techniques to dynamically adjust them. Therefore one possible future line of research could be to examine various techniques of setting these initial values. In this work a constant has been used, but this is likely far from optimal.

The techniques applied to mitigate the effects of local maxima has been quite successful. The techniques used for mitigating issues of shallowness however, has not been as elegant or successful. Needless to say, there is definitely room for improvement and ideas to address this issue.

A suggested line of future research suggested by [26] is to apply an AMAF technique such as RAVE to SameGame. This was originally part of the scope of this work, but was in the end dropped because it was estimated to give a relatively small improvement for SameGame in relation to the required work and increased complexity of the MCTS implementation. However, there might be value in attempting to develop an AMAF technique solely focused on a single-player domain and combining this with other techniques e.g. duplication detection.

We have seen in previous research that nested methods (e.g. NMCTS [5]) improves upon a traditional MCTS. A suitable question is to what extent the techniques in this work may be successfully integrated and adopted by such a method and how it ought to be achieved.

An obvious further line of research to follow limited to SameGame is to investigate how to weigh in the end game bonus. Or more formally how to most efficiently adopt the target problem to MCTS. This may include ingenious ways of combining the average score and the maximum score in the tree policy, since we have seen in this work that the latter can be successfully used if the domain is just adopted appropriately.

Bibliography

[1] R. Agrawal Sample mean based index policies with $O(\log n)$ regret for the multi-armed bandit problem Advances in Applied Probability, 27, 1054–1078, 1995.

[2] L. V. Allis Searching for Solutions in Games and Artificial Intelligence Ph.D. thesis, Department of Computer Science, Rijksuniversiteit Limburg, Maastricht, The Netherlands, 1994.

[3] P. Auer, N. Cesa-Bianchi and P. Fischer Finite-time Analysis of the Multiarmed Bandit Problem Machine Learning, pp. 235–256, 2002.

[4] H. Baier and M. H. M. Winands Beam Monte-Carlo Tree Search 2012 IEEE Conference on Computational Intelligence and Games (CIG).

[5] H. Baier and M. H. M. Winands Nested Monte-Carlo Tree Search for Online Planning in Large MDPs In 20th European Conference on Artificial Intelligence, ECAI 2012. IOS Press, 2012, pp. 109–114.

[6] P. Baudis Balancing MCTS by Dynamically Adjusting Komi Value ICGA Journal 34, pp. 131–139, 2011.

[7] T. Biedl, E. Demaine, M. Demaine, R. Fleischer, L. Jacobsen and J. Munro The Complexity of Clickomania More Games of No Chance (ed. R. Nowakowski), pp. 389–404, Cambridge University Press, 2002.

[8] Y. Björnsson and H. Finnsson CADIAPLAYER: A Simulation-Based General Game Player IEEE Trans. Comp. Intell. AI Games, vol. 1, no. 1, pp. 4–15, 2009.

[9] T. Cazenave Nested Monte-Carlo Search In IJCAI, 2009, pp. 456–461.

[10] B. E. Childs, J. Vermaseren, J. H. Brodeur and L. Kocsis Transpositions and Move Groups in Monte Carlo Tree Search Proc. IEEE Symp. Comput. Intell. Games, Perth, Australia, pp. 389–395, 2008.

[11] R. Coulom Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search 5th International Conference on Computer and Games, Turin, Italy, 2006.

[12] S. Edelkamp, P. Kissmann, D. Sulewski and H. Messerschmidt Finding the Needle in the Haystack with Heuristically Guided Swarm Tree Search Multikonf. Wirtschaftsinform., Gottingen, Germany, 2010, pp. 2295–2308.

[13] D. S. Johnson and L. A. McGeoch The travelling salesman problem: a case study in local optimization In Local Search in Combinatorial Optimization, E. H. L. Aarts and J. K. Lenstra, Eds. New York: Wiley: New York, 1997.

[14] JS–Games.de SameGame – Highscores Retrieved from http://www.js-games.de/eng/highscores/samegame/lx on 2014–01–19

[15] L. Kocsis and C. Szepesvári Bandit based Monte-Carlo Planning European Conference on Machine Learning, pages 282–293, 2006.

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

[17] S. Matsumoto, N. Hirosue, K. Itonaga, K. Yokoo and H. Futahashi Evaluation of Simulation Strategy on Single-Player Monte-Carlo Tree Search and its Discussion for a Practical Scheduling Problem In Proceedings of the International MultiConference of Engineers and Computer Scientists 2010, vol. 3, 2010, pp. 2086–2091.

[18] mcts.ai About Retrieved from http://mcts.ai/about/index.html on 2014–02–01

[19] MorpionSolitaire.com Morpion Solitaire – Rules Retrieved from http://www.morpionsolitaire.com/English/Rules.htm on 2014–07–01

[20] C. D. Rosin Nested Rollout Policy Adaptation for Monte Carlo Tree Search Proc. 22nd Int. Joint Conf. Artif. Intell., Barcelona, Spain, pp. 649–654, 2011.

[21] B. Ruijl, J. Vermaseren, A. Plaat and J. van den Herik Combining Simulated Annealing and Monte Carlo Tree Search for Expression Simplification Unreviewed arXiv:1312.0841v1, December 2013.

[22] A. Saffidine, T. Cazenave and J. Méhat UCD: Upper Confidence bound for rooted Directed acyclic graphs Knowledge–Based Systems, December 2011.

[23] M. P. D. Schadd, M. H. M. Winands, H. J. van den Herik, G. Chaslot and J.W. H.M. Uiterwijk Single-Player Monte-Carlo Tree Search In Computers and Games, volume 5131 of Lecture Notes in Computer Science, pages 1–12. Springer, 2008.

[24] M. P. D. Schadd, M. H. M. Winands, H. J. van den Herikand H. Aldewereld Addressing NP-Complete Puzzles with Monte-Carlo Methods In Proceedings of the AISB 2008 Symposium on Logic and the Simulation of Interaction and Reasoning. Vol. 9. The Society for the Study of Artificial Intelligence and Simulation of Behaviour, Brighton, United Kingdom, pp. 55–61, 2008.

[25] M.P.D. Schadd Selective Search in Games of Different Complexity, Chapter 3: Single-Player Monte-Carlo Tree Search Ph.D. Thesis. Department of Knowledge Engineering, Maastricht University, Maastricht, The Netherlands, 2011.

[26] M. P. D. Schadd, M. H. M. Winands, M. J. W. Tak and J. W. H. M. Uiterwijk Single-Player Monte-Carlo Tree Search for SameGame Knowledge-Based Systems. In press, 2012.

[27] David Silver Reinforcement Learning and Simulation-Based Search in Computer Go PhD thesis, University of Alberta, Edmonton, Canada, 2009.

[28] M. J. W. Tak The Cross-Entropy Method Applied to SameGame Bachelor thesis, Maastricht University, Maastricht, The Netherlands, 2010.

[29] F. W. Takes and W. A. Kosters Solving SameGame and its Chessboard Variant In Proceedings of 21st Benelux Conference on Artificial Intelligence, pages 249–256, 2009.

[30] S. Tavener, D. Perez, S. Samothrakis and S. Colton A Survey of Monte Carlo Tree Search Methods Transactions on Computational Intelligence and AI in Games, March 2012.

[31] Wikipedia, The Free Encyclopedia SameGame Retrieved from http://en.wikipedia.org/wiki/SameGame on 2014–01–19

[32] Wikipedia, The Free Encyclopedia 15–puzzle Retrieved from http://en.wikipedia.org/wiki/15%5Fpuzzle on 2014–02–26

[33] Wikipedia, The Free Encyclopedia Sokoban Retrieved from http://en.wikipedia.org/wiki/Sokoban on 2014–02–26

[34] Wikipedia, The Free Encyclopedia Minimax Retrieved from http://en.wikipedia.org/wiki/Minimax on 2014–02–26

[35] Wikipedia, The Free Encyclopedia Computer Go Retrieved from http://en.wikipedia.org/wiki/Computer%5Fgo on 2014–02–26

[36] Wikipedia, The Free Encyclopedia Deep Blue Retrieved from http://en.wikipedia.org/wiki/Deep%5FBlue%5F(chess%5Fcomputer) on 2014–02–26

[37] Wikipedia, The Free Encyclopedia Gradient Descent Retrieved from http://en.wikipedia.org/wiki/Gradient%5Fascent on 2014–08–04

Appendix A

Additional background material

The algorithms and ideas listed in this chapter will not be imperative to understand the gist of this work. They have however had an impact on the field and are therefore of interest as background material to anyone determined on aquiring a deeper knowledge of MCTS-based research on single-player domains. Furthermore, they also serve as a reference over ideas and concepts that have already been tried.

A.1 UCD

In this section we will provide finer detail of the algorithmic ideas of UCD. The main problem of storing statistics in the edges rather than the nodes is the selection phase. Since all information is no longer available locally, we must rethink our way of selecting which child node/edge to follow when traversing down the search tree. Saffidine et al. suggests using a so called adapted score and exploration factor, where we take edges up to a certain depth d into consideration (possibly a different d for score and exploration). The formula used has the following characteristic look:

$$UCD_{d1,d2,d3}(e) = avg_{d1}(e) + c \cdot \sqrt{\frac{\ln p_{d2}(e)}{n_{d3}(e)}}$$

where c is as customary the exploration constant. The function $avg_d(e)$ calculates the average score by taking all edges up to at most d further levels down in the tree into consideration, $n_d(e)$ does the same but for visits. $p_d(e)$ calculates the sum of all $n_d(f)$ where f is an edge from the origin of e, i.e. either e itself or a sibling of e.

One has to bear in mind that the tree in MCTS is built progressively and it therefore exists information available at an edge that is not available at any descendant of that edge. This information, i.e., information obtained by initiating a simulation at the edge, needs to be separately kept track of in order to make the definition of the adapted score sound. Let us denote these statistics by $avg'(e)$ and $n'(e)$ for an edge e. Let us also introduce the notations $a(e)$ and $b(e)$ denoting the

set of edges emanating from the destination and origin of the edge e respectively.

$$
\begin{aligned}
avg_0(e) &= avg(e) \\
avg_d(e) &= \frac{avg'(e) \cdot n'(e) + \sum_{f \in a(e)} avg_{d-1}(f) \cdot n(f)}{n'(e) + \sum_{f \in a(e)} n(f)} \\
n_0(e) &= n(e) \\
n_d(e) &= n'(e) + \sum_{f \in a(e)} n_{d-1}(f) \\
p_d(e) &= \sum_{f \in b(e)} n_d(f)
\end{aligned}
$$

Although the definition allows for certain inconsistencies, e.g. the statistics of an edge may be accounted for several times, the UCD policy has empirically been shown to outperform UCT in games such as *Hex* and *LeftRight* [22].

A.2 MC-RWS

From a broader research perspective MC-RWS may appear to be of little interest. That is partially true, the algorithm itself is rather trivial, but the results of it and the results related to it, are interesting in sheer performance but also for the fact that they create a possible contradiction in the research field.

Algorithm

The structure of MC-RWS is remarkably simple.

1. Let $X \leftarrow root$, the initial state.
2. Run R simulations from each child of X.
3. Let Y be the child of X with the highest average score. Set $X \leftarrow Y$.
4. If X is not a terminal state return to step 2 otherwise quit.

Apart from the more specified step 3 this definition coincides with that of *flat of Monte-Carlo search*. The contribution of MC-RWS lies in its sophisticated simulation strategy.

Simulation strategy

The insight realized is that the *TabuColorRandom* heuristic devised by Schadd et al. is likely to perform pointless moves just before the resulting large group is removed. Focusing solely on one color limits the possibility of larger groups of other colors. MC-RWS aims to resolve this dilemma.

Given a set of C colors $\{c_1 \dots c_C\}$ let $f(c)$ be the number of blocks left of color c. The probability $P(c_i)$ of removing a group of color c_i and our focus on larger groups α is defined as follows.

$$P(c_i) = \frac{1}{C-1}\left(1 - \frac{(f(c_i) - \theta)^\alpha}{\sum_{k=1}^{C}(f(c_k) - \theta)^\alpha}\right)$$
$$\alpha = 1 + (\beta/N) \cdot \sum_{k=1}^{C} f(c_k)$$

Where $N = 15 \times 15 = 225$, $\beta = 4$ and θ is some threshold to prevent issues with large values. Then during simulation we use the function P to weight the probability of removing a certain color. We will for the sake of brevity call this simulation strategy RWS.

Results and discussion

MC-RWS outperformed the first version of SP-MCTS. This showed that a sophisticated simulation policy can yield great results despite being combined with a naïve tree policy. One obvious question is then of course, can this simulation strategy improve SP-MCTS? Because MC-RWS should be improvable by UCB as well as MCTS/UCT, one is inclined to believe that so must be the case. Interestingly the answer is in fact considered to be No [26, p. 9] based on research by Tak [28].

The reason is that RWS is too expensive in comparison to TabuColorRandom and that we are better off doing more simulations in SP-MCTS. However, one may question this conclusion, because if one looks at the results presented by Tak [28, p. 6] (Table 3, compare the 5s and 30s setting) one will find that even when the version using RWS has generated a higher number of nodes on average than the version using TabuColorRandom, the average score of the former is still lower than that of the latter. This is a contradictory result under the assumption that RWS outperforms TabuColorRandom as a simulation strategy.

A possible explanation could be an existing flaw in Tak's experimental setup or presentation. Another explanation could simply be that RWS does not outperform TabuColorRandom, and that the sole reason of MC-RWS beating the first version of SP-MCTS was that the former allocated resources per move rather than game. This would however lead to a number of follow-up issues, e.g. how good would flat Monte-Carlo using TabuColorRandom be and does MCTS/UCT really give any substantial improvement in SameGame?

A.3 NMCS and NMCTS

In this section we will outline in greater details the contents of the NMCS algorithm. We will then in a subsequent section briefly mention how the ideas are adapted into the NMCTS algorithm.

Algorithm - NMCS

The algorithm in its naïve version is rather simple:

```
NMCS(state, level)
(1)    score ← 0
(2)    while state is not terminal
(3)       best ← −∞
(4)       move ← RANDOM(A(state))
(5)       if level > 0
(6)          foreach a ∈ A(state)
(7)             sample ← points(state, a) + NMCS(f(state, a), level − 1)
(8)             if sample > best
(9)                best ← sample
(10)               move ← a
(11)      score ← score + points(state, move)
(12)      state ← f(state, move)
(13)   return score
```

If we have reached the base level, i.e. 0, we simply choose random moves. Otherwise we choose moves suggested by the lower level search. However, it is possible that a higher level search will not improve upon the best score found by any of the lower level searches, thus it is prudent to cache the globally best sequence of moves and always use the best move as the final move choice. This improved NMCS considerably and gave the algorithm the following look.

```
NMCS(state, level)
(1)   score ← 0
(2)   while state is not terminal
(3)     best ← −∞
(4)     move ← RANDOM(A(state))
(5)     if level > 0
(6)       foreach a ∈ A(state)
(7)         sample ← points(state, a) + NMCS(f(state, a), level − 1)
(8)         if sample > best
(9)           best ← sample
(10)          move ← a
(11)    if best > best_global
(12)      UPDATE(best_global, move_global, best, move)
(13)    if level = 0
(14)      score ← score + points(state, move)
(15)      state ← f(state, move)
(16)    else
(17)      score ← score + points(state, move_global)
(18)      state ← f(state, move_global)
(19)  return score
```

Results and discussion

NMCS bested the score of MC-RWS (and thus also SP-MCTS) in SameGame. A possible intuition to why NMCS performs well is that it manages to reach deep by performing smaller (lower level) searches and utilize this knowledge high up in the search tree.

Naturally the idea of NMCS can be moved to a tree context, thus defining Nested Monte-Carlo Tree Search (NMCTS) as coined by Baier and Winands [5]. The algorithm works analogously to MCTS, but during the simulation step, instead of running a simulation from the concerned node, a NMCTS search of a lower level is executed. At the base level moves are chosen according to the simulation policy (e.g. random). Thus a level-0 NMCTS corresponds to a simple simulation and a level-1 NMCTS is identical to an ordinary MCTS.

NMCTS has not been evaluated on the standard test set of SameGame, but on the 100 first levels of the Schadd et al. test set, where it outperformed NMCS despite the fact that various parameters in the algorithm had not been tuned. The authors also employed a new simulation strategy based on *TabuColorRandom* combined with bandit based methods (UCB1-TUNED) trying to learn which color to most favorably use as the taboo. Furthermore it appeared to be the case that NMCTS performs well in domains where a multi-start MCTS (meta-search) beats a single large MCTS.

A.4 NRPA

In this section we will explain more thoroughly the quirks of the NRPA algorithm and its impact on the field.

Algorithm

NRPA does not explicitly navigate any search tree or uses the nested iterations to choose a move, but uses instead the nested iterations to solely update the simulation policy. Each nested level inherits the policy of its parent level and uses the solutions found by running N iterations of a lower level nested search to adapt it. The policy adapted after the N iterations is not returned to the parent level, but only the best solution found. The policy passed down from each nested level is then finally used at level 0 in a simulation starting from the initial state.

NRPA(*policy*, *level*)
(1) **if** $level = 0$
(2) $state \leftarrow initial\_state$
(3) $solution \leftarrow \{\}$
(4) **while** *state* is **not** terminal
(5) $move \leftarrow a \in A(state)$ with probability proportional to $e^{policy[code(state,a)]}$
(6) $solution \leftarrow solution \cup move$
(7) $state \leftarrow f(state, move)$
(8) **return** $(score(state), solution)$
(9) **else**
(10) $best \leftarrow -\infty$
(11) **for** $i \leftarrow 1$ **to** N
(12) $(result, new) \leftarrow$ NRPA(*policy*, $level - 1$)
(13) **if** $result \geq best$
(14) $best \leftarrow result$
(15) $solution \leftarrow new$
(16) $policy \leftarrow$ ADAPT(*policy*, *solution*)
(17) **return** $(best, solution)$

ADAPT($policy$, $level$)
(1) $state \leftarrow initial\_state$
(2) $policy' \leftarrow policy$
(3) **for** $i \leftarrow 0$ **to** $length(solution) - 1$
(4) $policy'[code(state, solution[i])] \leftarrow policy'[code(state, solution[i])] + \alpha$
(5) $z \leftarrow \sum_{a \in A(state)} e^{policy[code(state,a)]}$
(6) **foreach** $a \in A(state)$
(7) $policy'[code(state, a)] \leftarrow policy'[code(state, a)] + \alpha \cdot e^{policy[code(state,a)]}/z$
(8) $state \leftarrow f(state, solution[i])$
(9) **return** $policy'$

The only required domain dependent knowledge in NRPA is the definition of $code(state, a)$ associating a specific code (number) with the action a used from the given state. This is the only aspect of the algorithm where one may incorporate some custom design choices, e.g. mapping a specific action to the same code for different states, though the code function must still adhere to some properties concerning uniqueness. In addition to this the value of N the number of iterations and α the step size in the direction of the gradient to adapt the policy can be chosen freely. The algorithm is initiated by a call `NRPA(`*policy*`, L)` where *policy* is an all-zero vector and L is the desired nesting level.

The definition of *Adapt* follows from the derivative of the probability (when viewed as a function) of simulating a specific sequence. The probability can be increased exponentially quickly by *Adapt*, roughly a factor close to $\alpha \cdot e$ on actions with low probability. Thus given enough iterations NRPA ensures that the best score found so far will be eventually returned. Do note that we do not require strict improvements on the best solution found in order to change it. The rationale for this design choice is mainly empirical as it proved to be superior in preliminary experiments by the author, though it preserves the aforementioned property whilst aiding exploration.

Results

The results of NRPA belong to the more promising ones in the field. Using a value of $\alpha = 1.0$, $N = 100$ and a nesting level of either 4 or 5, new records were achieved in the domains of Crossword Puzzle Construction and Morpion Solitaire using less computational time compared to previous approaches. The most impressive one being an improvement upon a human-generated Morpion Solitaire record that had stood for over 30 years.

A.5 BMCTS

We will in this section further describe BMCTS.

Algorithm

Maintain for each depth of the tree the number of simulations that have passed through any tree node at that depth. When this number reaches a predetermined parameter L for a depth d, the tree will be pruned by retaining the W most visited nodes at depth d including their ancestors, all other nodes will be discarded even descendants of the W nodes. Furthermore, from now on no new nodes will be added up to depth d. Although not specified in the original article, it has to be assumed that the counter for all depths larger than d is reset when depth d pruning takes place. Moreover, it is unclear what benefits discarding the descendants in the pruning step yield.

Results

For SameGame BMCTS was shown to yield a speedup over MCTS of a factor 2 and 4 for random- and informed simulations respectively. That is, BMCTS could find an equally good solution as MCTS in half the time.

A.6 HGSTS

The specification of the HGSTS algorithm is rather complex and unclear, but we will have a look at the key ideas used.

Non-uniform sampling During simulation, instead of choosing one of the moves uniformly at random, each move is assigned a heuristic value e.g. the score of the move, and chosen with a probability proportional to the heuristic value.

Look-ahead Apply a one step look-ahead every l:th iteration where $l \in [5, 20]$. Doing a look-ahead each iteration turned out to be too costly. It is not clear where or how this look-ahead is applied, but presumably it is done during simulation.

Duplication and symmetry detection HGSTS employs detection of duplicate and symmetric states in order to reduce redundant work. How the details of this is done, especially in the context of UCT, is not described in detail.

Node count update In the parallelized version of UCT the number of node visits are updated while traversing down the tree instead of during the backpropagation phase. This increases the probability of different threads exploring different parts of the tree.

Set-based parallelization The possibly most unique idea of HGSTS is the so called set-based parallelization being an alternative to a parallelization of UCT. Perform a complete BFS up to a certain layer. For each state in this layer a node in the UCT tree is created. Every such node is then inserted into a priority queue Q sorted on the UCT value. Then the following procedure is

repeated an arbitrary number of times: Perform a UCT iteration on each of the k first nodes in Q possibly using n threads (k/n is typically small), update the UCT values of the nodes in Q and restore its order.

Furthermore, the simulation strategies TabuColorRandom and TabuColor were applied, where the chosen taboo color was tuned manually for each board in the standard test set. This might have been one contributing factor of HGSTS's success. Unfortunately it is not known which time settings were used to obtain the results and the results have not been independently verified [25, p. 38].

Appendix B

Additional data

In order to not overwhelm the reader with data, we will aim to present any results that is not thoroughly discussed in this section instead. For most readers a graph is sufficient to grasp the results, but for the interested reader exact figures will be presented here.

B.1 Duplication detection

This section will give additional data on the experiments of section 6.7.

For the supplementary statistics the category *Hit-count* denotes the number of times a duplicate state was detected (and we refrained from creating a new node in the search tree), *Suboptimal* denotes the number of times we reached a state with a corresponding node already in the search tree and the new path reaching it improved upon the old one, and *Hit-depth* denotes the average depth where we discovered a duplicate state.

Name	Min	Avg	Max	Avg depth
Naive	2357	2698	3034	27.20
Dynamic	2404	2714	3034	27.20
Consistent	2367	2700	3031	26.91

Figure B.1. Primary statistics on the Schadd test set (64×10^4 ru).

Name	Hit-count	Suboptimal	Hit-depth	Max depth
Naive	55277	4461	31.32	39.87
Dynamic	55374	1575	30.40	39.86
Consistent	50068	1813	29.78	39.78

Figure B.2. Supplementary statistics on the Schadd test set (64×10^4 ru).

Name	Min	Avg	Max	Avg depth
Naive	1705	1984	2280	41.38
Dynamic	1635	2019	2371	39.21
Consistent	1658	2145	2650	39.53

Figure B.3. Primary statistics on the Standard test set (64×10^5 ru).

Name	Node	Terminal	Hit-count	Hit-depth	Max depth
Naive	228078	1533	511201	44.07	50.68
Dynamic	233379	1872	508889	43.13	55.19
Consistent	236730	1557	486840	43.22	55.77

Figure B.4. Supplementary statistics on the Standard test set (64×10^5 ru).

B.2 Combining CNode and SA-UCT

This section will give additional data on the experiments in Section 6.9.2.

Name	Min	Avg	Max	C_{start}	Avg depth	Terminal nodes
Linear	2330	2609	2917	1000	8.91	60.8
Linear	2448	2713	2986	500	13.48	85.1
Linear	2480	2776	3072	250	17.62	105.0
Linear	2506	2786	3064	200	18.45	112.2
Linear	2491	2789	3083	180	18.91	109.7
Exponential	2468	2773	3051	1000	17.36	128.0
Exponential	2507	2794	3069	500	18.19	143.4
Exponential	2498	2783	3061	300	18.64	154.0
Exponential	2530	2807	3091	256	18.63	159.0
Exponential	2462	2768	3070	200	18.76	163.9

Figure B.5. Statistics on the Schadd test set using 64×10^4 ru.

B.3 3rdAvg+

This section will give additional data on the experiments in Section 6.10.1.

Name	Min	Avg	Max	C	Avg depth
Static	2342	2653	2975	0.020	30.54
3rdAvg	2543	2843	3139	0.041	30.08
3rdAvg+	2331	2649	2976	0.042	29.92

Figure B.6. Standard test set, 64×10^4 ru.

B.4 Simulation strategy

This section will give additional data on the experiments in Section 7.0.5.

Name	Min	Avg	Max	C	Avg depth
Base3rdAvg	2735	3003	3262	0.041	17.07
Base3rdAvgExp+	2721	3026	3312	0.041	19.26
Base3rdAvgExp+Tabu	3072	3303	3504	0.041	20.03
BaseExp+Tabu	3093	3322	3524	0.021	20.06

B.5 Augmenting the simulation strategy

This section will give additional data on the experiments in Section 7.0.6.

Name	Min	Avg	Max	C	Avg depth
BaseExp+Tabu	3093	3322	3524	0.021	20.06
BaseExp+Tabu32	3149	3339	3523	0.021	19.55
BaseExp+Tabu48	3162	3351	3522	0.021	19.40
BaseExp+Tabu64	3149	3333	3507	0.021	19.41

Appendix C

Vocabulary

The reader is not expected to memorize all terms and definitions by just reading them once, for that sake we introduce a vocabulary listing specific – possibly common and reoccuring – words and abbreviations together with a brief description or reference to where they are defined/explained.

3rdAvg A MCTS variation, see Section 6.3.

AI Artificial Intelligence.

AMAF All Moves As First, a family of techniques aiming to speed up the convergence rate of the estimates provided by UCT, see Section 3.5.1.

Backpropagation The fourth phase of MCTS, see Section 3.2.

BFS An exhaustive search algorithm enumerating all states by the number of transitions from the initial position.

Clickomania A game identical to SameGame with the difference that the only objective is to clear the board.

CNode A MCTS variation, see Section 6.9.2.

DAG Directed Acyclic Graph.

Default policy See Section 3.2.

Expand See Section 1.2.1.

Go A two-player board game, where the two players take turns placing white and black stones respectively on the board.

Group Blocks of the same color being orthogonally connected in SameGame.

Local Maxima The unfavorable effect of having too many nodes of the search tree in one or more subtrees compared to the other subtrees.

LU Late Urgency, a technique to mitigate shallowness. It comes in several minor variations, see Section 6.9.1, 6.9.2 and 6.9.2.

MCTS Monte-Carlo Tree Search.

Move The removal of a specific group in SameGame.

NMCTS Nested Monte-Carlo Tree Search.

Node See Section 1.2.1.

RAVE Rapid Action Value Estimation, a certain AMAF technique, see Section 3.5.2.

SameGame A tile-matching puzzle high score game.

Shallowness Refers to the situation when the average node depth of a search tree is unfavorably low.

Solution string A sequence of moves yielding a solution to SameGame, i.e. all moves are valid when applied in order and no more move is possible when the last move in the sequence has been performed.

Solved node See Section 5.1.

SP-MCTS Single-Player Monte-Carlo Tree Search.

State See Section 1.2.1.

Static A variation of UCT applying normalization using a constant, see Section 6.1.

Terminal node A node in a search tree corresponding to a terminal state.

Terminal state A state from which there exists no available move.

Transition See Section 1.2.1.

Tree policy See Section 3.2.

UCB Upper Confidence Bound.

UCB1 A specific algorithm/policy adressing the *multi-armed bandit problem*, see Section 3.3.

UCB1-TUNED An empirically refinement of UCB1, see Section 3.3.

UCT Upper Confidence bound applied to Trees, a specific variation of MCTS using UCB1.

VS-pruning A move pruning strategy for SameGame, see Section 2.4.2.

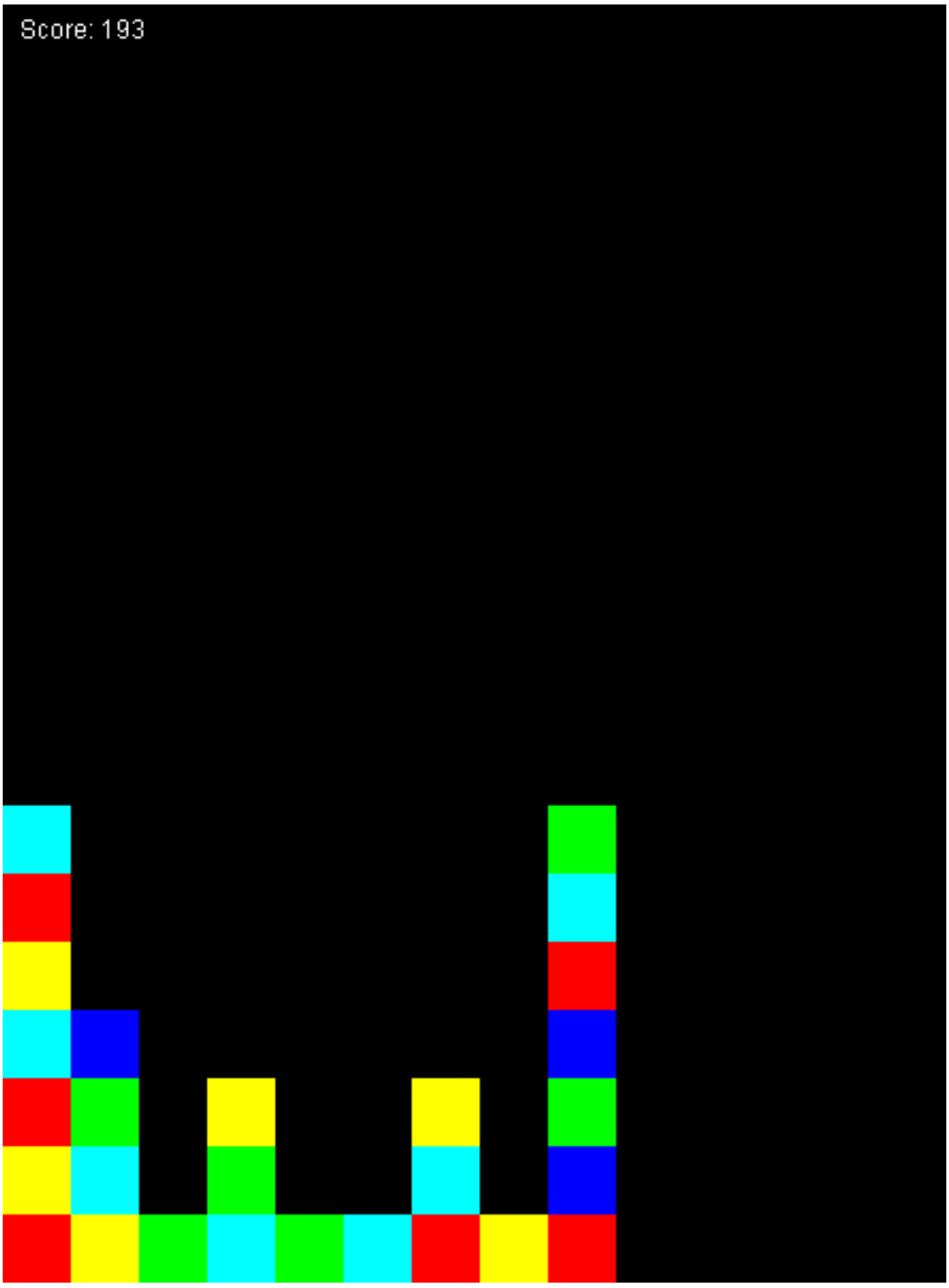

Figure C.1. Illustration showing a terminal state in SameGame.

C.1 MCTS variation cheat sheet

Keeping track of all abbreviations of MCTS based variations introduced in this work might be quite difficult. Therefore we introduce a brief cheat sheet to be used at such times.

Name	Tree policy formula
Static	$\frac{avg_{global}(x)}{5000} + C \cdot \sqrt{\frac{\log n(p)}{n(x)}}$
3rdAvg	$\frac{avg_{global}(x)}{5000} + C \cdot \sqrt{\frac{\log n(p)}{n(x)}} + \frac{avg_{local}(x)}{5000}$
SP-MCTS	$\frac{avg_{global}(x)}{5000} + C \cdot \sqrt{\frac{\log n(p)}{n(x)}} + \sqrt{\frac{V(x)}{5000^2} + \frac{D}{n(x)}}$
UCT1-TUNED	$\frac{avg_{global}(x)}{5000} + C \cdot \sqrt{\frac{\log n(p)}{n(x)} \cdot min(V_T(x), 1/4)}$
TopScore	$\frac{top_{global}(x)}{5000} + C \cdot \sqrt{\frac{\log n(p)}{n(x)}}$

Figure C.2. Tree policy cheat sheet.

where

$$V(x) = \frac{\sum_{y \in R(x)} y^2 - n(x) \cdot avg_{local}(x)^2}{n(x)}$$

$$V_T(x) = \frac{V(x)}{5000^2} + \sqrt{\frac{2 \log n(p)}{n(x)}}$$

Name	Description
Dynamic	Duplication detection assigning the nodes with higher cumulative score as parents.
Consistent	As *Dynamic* but updates the statistics of nodes as parents are reassigned.
CNode	Assigns an individual explorative factor C_p to each node p in the search tree, and whenever a terminal node x is expanded to the search tree, all ancestors of x get their explorative factor increased.
CNodeLU	As *CNode* but starts decreasing C_p of a node p when only a fraction of the initial resources remains and there is no terminal node in the subtree rooted at p.
Base	Combines *Consistent* and *CNodeLU*.

Figure C.3. Additional cheat sheet.

Appendix D

Code

A fair amount of time was spent on implementing in code the game mechanics of SameGame, the various algorithms to be evaluated, the program to measure the complexity, and so on. Consequently we simply present the most significant pieces of code in this chapter. The programming language used for all aspects was Java.

D.1 SameGame

In this section we present the part of the code which is linked to SameGame without any strong connection to a particular algorithm.

D.1.1 Game mechanics

In this section we present the implementation of the underlying game logic, such as generating moves, make blocks fall, shift columns to the left, etc.

Board.java

`https://bitbucket.org/BwaKell/mcts_samegame/downloads/Board.java`

D.1.2 Game visualization

Early on a program for either playing or watching an algorithm play SameGame was constructed. The program implements some parts of the SameGame game logic to manipulate the graphical board, which is a duplication of functionality that could be avoided, but it serves to some extent as a double check of the correctness of a solution and the game mechanics.

GameFrame.java

`https://bitbucket.org/BwaKell/mcts_samegame/downloads/GameFrame.java`

Game.java

`https://bitbucket.org/BwaKell/mcts_samegame/downloads/Game.java`

D.2 MCTS

In this section we present the code used for the final MCTS implementation.

MCTS.java

`https://bitbucket.org/BwaKell/mcts_samegame/downloads/MCTS.java`

D.3 Complexity

In this section we present the code used for various complexity measurements of SameGame.

D.3.1 State-space complexity

The upper bound calculation of the state-space complexity of a SameGame instance.

```
import java.util.*;

public class StateSpaceComplexity
{
  static final HashSet<Long> set = new HashSet<Long>();

  static void rek(final int i, final long hash, final int[] col)
  {
    if(i<0){ set.add(hash); return; }

    int j = i; long nxt = hash;
    for(; j>=0 && col[j]==col[i]; j--) nxt = nxt*6 + col[j];

    rek(j, hash, col); //Segment removed.
    rek(j, nxt, col); //Segment left.
  }

  //Returns an estimate of the state-space complexity of the board.
  public static double ssc(final int[][] board)
  {
    final long[] cnt = new long[15];
    for(int i = 0; i<15; i++)
    {
      set.clear();
      rek(14,0,board[i]);
      cnt[i] = set.size();
    }

    double dspace = 1;
```

```
    for(int i = 0; i<15; i++) dspace *= cnt[i];

    return dspace;
  }
}
```

D.3.2 Game-tree complexity

The estimation of the game-tree complexity of a SameGame instance.

```
import java.util.*;
import java.math.*;

public class GameTreeComplexity
{
  static final Random rnd = new Random();
  static double avg_branch;
  static int cnt_branch;

  public static int simulate(final int[] board)
  {
    final int[] mvs = Board.mvs;
    for(int len = 0; ; len++)
    {
      //final int d = Board.moves(board); This line for VS-pruning.
      final int d = Board.rawMoves(board);
      avg_branch += (d/2 - avg_branch)/++cnt_branch;
      if(d==0) return len;
      final int mv = 2*rnd.nextInt(d>>1), i = mvs[mv];
      Board.doMove(board, i, mvs[mv+1], 0);
    }
  }

  //Returns an estimate of the game-tree complexity of the board using
    simlim number of simulations.
  public static double gtc(final int[] board, final int simlim)
  {
    avg_branch = cnt_branch = 0;

    double avg_len = 0;
    for(int n = 0; n<simlim; )
    {
      final int sample = simulate(Arrays.copyOf(board,15*15));
      avg_len += (sample - avg_len)/++n;
    }

    return Math.pow(avg_branch,avg_len);
  }
}
```