POLITECNICO DI MILANO
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

Monte Carlo Tree Seach for Sokoban

Relatore:
Prof. Pier Luca LANZI

Tesi di Laurea di:
Fabio MAROCCHI, Matr. 858408
Mattia CRIPPA, Matr. 854126

Anno Accademico 2016-2017

Sommario

Monte Carlo Tree Search (MCTS) è uno degli algoritmi più utilizzati nel campo dell'intelligenza artificiale applicata ai giochi da tavolo e ai giochi di carte. Lo scopo di questa tesi è di valutare le prestazioni dell'algoritmo MCTS applicato al puzzle game Sokoban, confrontandolo con un altro algoritmo ben noto, *Iterative Deepening A**, che ha dimostrato di avere un discreto successo in questo puzzle game. In questa tesi applichiamo MCTS e IDA* a Sokoban e Samegame, un altro puzzle game. Sviluppiamo anche una serie di ottimizzazioni conosciute per MCTS e IDA* e ne presentiamo alcune nuove. Infine, valutiamo gli effetti delle ottimizzazioni su entrambi i domini. I nostri risultati mostrano che in Samegame la formula UCB1-Tuned ottiene prestazioni migliori rispetto a SP-MCTS, una versione single player di MCTS che ha ottenuto buoni risultati in quel dominio in passato. In Sokoban, la migliore configurazione del MCTS usa l'UCT standard con l'aggiunta delle ottimizzazioni proposte denominate Node Elimination e Cycles Avoidance, che portano ad un drastico aumento del numero di livelli risolti dall'algoritmo MCTS in Sokoban. Nonostante ciò, anche con una serie di miglioramenti che possono essere trovati in letteratura, che sono stati ampiamente utilizzati e hanno raggiunto risultati di successo, l'algoritmo MCTS non ha potuto eguagliare le prestazioni di IDA* in termini di numero di livelli risolti. Per questo motivo IDA* rimane ancora il miglior algoritmo per Sokoban.

Abstract

Monte Carlo Tree Search (MCTS) is one of the most used algorithm in the field of Artificial Intelligence applied to board and card games. The aim of this thesis is to evaluate the performance of MCTS algorithm applied to the puzzle game Sokoban, comparing it to another well known algorithm, *Iterative Deepening A**, which has been proven to be quite successful in this puzzle game. In this thesis we apply MCTS and IDA* to Sokoban and Samegame, another puzzle game. We also develop a series of known optimizations to MCTS and IDA* and present some new ones. Finally, we evaluate the effects of the optimizations on both domains. Our results show that in Samegame the UCB1-Tuned formula performs better than SP-MCTS, a single player version of MCTS that obtained good results in that domain in the past. In Sokoban, the best MCTS configuration uses the standard UCT with the addition of the proposed optimizations called Node Elimination and Cycles Avoidance, which lead to a drastic increase in the number of levels solved. Despite this, even with a set of enhancements that can be found in the literature, which have been widely used and have achieved successful results, the MCTS algorithm could not match IDA* performance in terms of number of solved levels. For this reason IDA* still remains the best algorithm for Sokoban.

Acknowledgments

We would like to thank our supervisor Prof. Pier Luca Lanzi for the guidance provided during our research and for helping us with feedbacks and suggestions.

We also thank our families and friends for their support. We wouldn't have been able to reach this goal without them.

Fabio and Mattia, 06/04/2018

Contents

1 Introduction **1**
1.1 Thesis outline 2

2 Puzzles and AI **5**
2.1 Sokoban 6
2.1.1 Rules 6
2.1.2 Complexity 8
2.2 Iterative Deepening A* 9
2.2.1 Overview 10
2.2.2 The Algorithm 11
2.2.3 State of the art 11

3 Monte Carlo Tree Search **17**
3.1 State of the art 18
3.2 The Algorithm 20
3.3 Upper Confidence Bounds for Trees 22
3.4 Characteristics 23
3.5 MCTS Enhancements 25
3.6 MCTS for puzzles 27
3.6.1 Samegame 28

4 Our Approach **33**
4.1 IDA* optimizations 33
4.1.1 Transposition Tables 33
4.1.2 Move ordering 37
4.2 MCTS configuration 37
4.3 MCTS optimizations 38
4.3.1 Object Pooling 39

4.3.2 Node Recycling 39
4.3.3 SP-MCTS UCB 41
4.3.4 UCB1-Tuned 41
4.3.5 Rapid Action Value Estimation (RAVE) . . 41
4.3.6 Node Elimination 43
4.3.7 Cycles Avoidance 44
4.4 Sokoban optimizations 45
4.4.1 Push Level Search 45
4.4.2 Deadlock Detection 46
4.4.3 Tunnel Macros 46
4.4.4 Goal Macros 47
4.4.5 Goal Cuts 48

5 Experimental results 49
5.1 Samegame 49
5.1.1 Node Elimination & Cycles Avoidance . . . 50
5.1.2 Parameters tuning 51
5.1.3 Results 53
5.2 Sokoban 55
5.2.1 Reward types 55
5.2.2 Node Elimination & Cycles Avoidance . . . 61
5.2.3 Parameters tuning 64
5.2.4 Simulation policy 65
5.2.5 Sokoban complexity 66
5.2.6 Results 67

6 Conclusions 69
6.1 Future research 70

Bibliography 71

List of Figures

2.1 Example of a Sokoban level 6
2.2 Deadlock example 8

3.1 MCTS steps 21
3.2 Move sequence example 29

4.1 Hash Table 34
4.2 AMAF heuristic 42
4.3 Node elimination 44
4.4 Tunnel Macro 47
4.5 Tunnel Macro Exception 47

5.1 UCB1-Tuned constants scores 52
5.2 InverseBM solved levels rate and tree depth 56
5.3 NegativeBM solved levels rate and tree depth . . . 57
5.4 R0 solved levels rate and tree depth 58
5.5 Boxes solved levels rate and tree depth 59
5.6 SP-MCTS and UCB1-Tuned results with InverseBM 60
5.7 SP-MCTS and UCB1-Tuned results with NegativeBM 60
5.8 SP-MCTS and UCB1-Tuned results with Boxes . . 61
5.9 SP-MCTS and UCB1-Tuned results with R0 61
5.10 Node Elimination evaluation 62
5.11 Node Elimination and Avoid Cycles solved levels . . 63
5.12 IDA* solved levels 66
5.13 MCTS solved levels 67

List of Tables

5.1 Our SP-MCTS results 50
5.2 Node Elimination outcome 51
5.3 RAVE thresholds scores 52
5.4 Node Recycling memory budgets scores 52
5.5 SP-MCTS versus UCB1-Tuned in Samegame 53
5.6 Optimized MCTS versus IDA* in Samegame 54
5.7 Node Elimination and Avoid Cycles evaluation . . . 63
5.8 RAVE thresholds evaluation 64
5.9 Node Recycling memory budgets evaluation 65
5.10 Results of ϵ-IDA* 65
5.11 Results of the different simulation policies. 66
5.12 Methods comparison 67

List of Algorithms

1 IDA* 12
2 General MCTS approach 22
3 UCT Algorithm 24

Chapter 1

Introduction

This thesis focuses on the application of Artificial Intelligence (AI) to puzzle games. This field was born in the '50s and the first AI algorithms, developed for two-players-board games (like Checkers and Chess), were able to play only final moves of the game or they could only play at beginners level. In the following years, these programs could compete against human-expert players due to more advanced techniques that have been developed. In some cases, it has been possible to solve a game, i.e. predict the result of a game played from a certain state in which all the players did the optimal moves.

The aim of this thesis is to evaluate the performance of *Monte Carlo Tree Search* (MCTS) applied to the puzzle game Sokoban, a single-player computer game in which the player inside a maze has to push boxes to assigned positions. Only one box at a time can be pushed and the boxes can not be pulled.

MCTS has been introduced in 2006 by Rèmi Coulom [1], combining tree search with Monte-Carlo evaluations which introduced independence from domain knowledge and a fine-grained control of the growth of the tree. Shortly after, Kocsis and Szepesvàri [2] formalized this approach into the *Upper Confidence Bounds for Trees* (UCT) algorithm, which nowadays is the most used algorithm of the MCTS family. In contrast with the classical AI algorithms (like Minimax), that completely explore the search tree, MCTS builds up a tree in an incremental and asymmetric manner guided by

many random simulated games. In this way it can explore only the most promising areas of the tree. Moreover, the exploration can be stopped at any time returning the current best result, this make MCTS very efficient in terms of time and memory. Kocsis and Szepesvàri were also able to prove that, with enough iterations of the algorithm, MCTS converges to the same result of Minimax.

Due to the successful application of MCTS algorithm to many board games, in this thesis we evaluate its efficiency in Sokoban comparing it to another well known algorithm, *Iterative Deepening A**, which has been proven to have a huge success in this puzzle game [3].

1.1 Thesis outline

The structure of the thesis is describe in the following:

- Chapter 2 provides a background of Sokoban, outlining its rules and complexity. It also provides a description of the used algorithm *Iterative Deepening A** and its enhancements for the chosen puzzle.

- Chapter 3 reports the state of art of the *Monte Carlo Tree Search* (MCTS) algorithm, a description of it and enhancements that have been developed. It also describes a case study, represented by Samegame, in with MCTS has been successfully used.

- Chapter 4 presents our solution to the puzzle solving problem, describing the domain dependent/independent enhancements used to improve performance.

- Chapter 5 outlines all the experiments used in this thesis in order to compare the efficiency of the different algorithms in the different domains. Then compares the results obtained by the algorithms in the proposed experiments in order to determine which algorithm has better performance in each domain.

- Chapter 6 presents an analysis of the results with respect to the aim of the thesis and also set out possible further improvements that can be integrated in this work.

Chapter 2

Puzzles and AI

In this chapter we provide a background of Sokoban, describing the game rules and its complexity. We then give a short introduction of the classic algorithm *Iterative Deepening A** and some of its enhancement which have been used in the selected game and that can be found in literature.

Puzzles have been popular since the dawn of mankind, and are known for stimulating brain activity and mental welfare as well as simply being fun. A puzzle is a pastime that consists of a problem or a riddle that tests the ingenuity of those who are called to solve it. There are many types of puzzles that can test different problem-solving skills including logic, pattern recognition, sequence solving, and word completion. The various types of solutions may require structuring a form or creating a certain order. The term puzzle usually refers to single-player games which are enjoyable to play. By definition a puzzle should have a solution which is aesthetically pleasing and gives the user satisfaction in reaching it [4].

In the last decades puzzles have become a more interesting field in computer science due to the fact that finding solutions for them, often requires the recognition of patterns. This makes puzzles suitable to be solved using artificial intelligence agents. A lot of different approaches have been used trying to find a solution to the puzzle solving problem and it has been discovered that specific artificial intelligence methods can work better in some domains rather than in others.

A large amount of different puzzles and artificial intelligence applied to them can be found in literature, and one of the most used algorithm is *Iterative Deepening A**. This algorithm has been employed to solve different puzzle games such as n-Puzzle and Sokoban obtaining good results [4].

2.1 Sokoban

Sokoban is a single-player computer game created by Hiroyuki Imabayashi in 1981 and published in December 1982 by Thinking Rabbit, a software house based in Takarazuka, Japan (the japanese name "Sokoban" can be translated as "warehouse keeper"). A level in Sokoban, as shown in Figure 2.1, consists of a series of rooms and passageways on a grid, in which are scattered a series of boxes (also called stones) and goals (the dashed area). The number of boxes must match the number of goals.

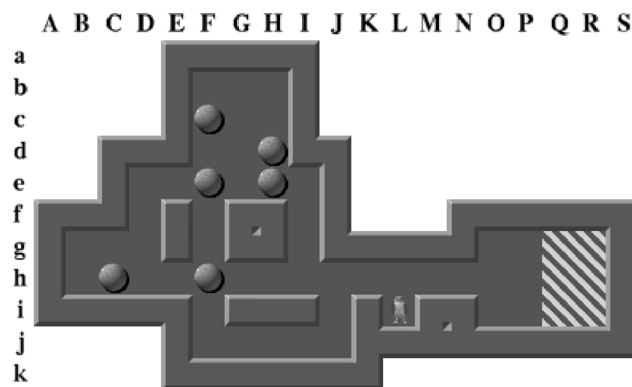


Figure 2.1: Example of a Sokoban level [3]

2.1.1 Rules

The goal of the game is to move every box on a goal square. The player controls a character (which we will refer to as the *pusher*) that can move around the level orthogonally on empty squares and goal squares. It cannot cross boxes or walls. If the player tries to move on a square occupied by a box, the box is pushed by one

square in the direction of the movement, provided that the target square is either empty or an unoccupied goal. The boxes cannot be pulled, so some moves can be irreversible.

Once every box is on a goal square, the level is solved. In addition to solving the level, the player can also try to find an optimal solution. In Sokoban, the optimality of the solution is usually measured by minimizing one of two metrics:

- The total number of pushes, i.e. moves which cause the box to change its position;
- The total number of moves, whether they are simple pusher movements or actual box pushes.

The canonical metric used is the number of pushes. An optimal solution for a sokoban level can range from a minimum of one push and one move (see level 44 of the Microban set [5]) to hundreds of pushes and thousands of moves. Sokoban uses a standard format for level files and many level sets have been created by users and enthusiasts[1].

One aspect that differentiates it from most puzzles studied in the literature is that an irreversible move can lead to a state that we call *deadlock*, from which no solution can be found. There are two main types of deadlock:

- A *simple deadlock* happens when a box is moved on a square from which it cannot reach any goal, regardless of the position of other boxes.
- A *freeze deadlock* happens when a box is moved on a non goal square and becomes immovable. That is, when it can't be pushed again. This often includes other boxes that become immovable in turn.

Simple deadlocks only depend on the position of a single box, so they can be computed once per level and are fairly easy to identify. Freeze deadlocks instead require interaction between multiple boxes

[1]Most level sets can be found at http://www.sourcecode.se/sokoban/levels

so they must be dealt with at run time. Anyhow, the game is not equipped with deadlock detection mechanisms, so the player can actually continue to play and interact with other boxes despite the deadlock situation.

In addition to the two main deadlock types, there are other situations that will eventually lead to one of the mentioned deadlocks, but might be recognized earlier. *Corral deadlocks* are defined as a situation in which a portion of the board can't be reached by the player because its access is blocked by boxes and there's no way to get all of those boxes to a goal square. *Bipartite deadlocks* happen when all boxes could still reach a goal, but there's an overlap on the goals they can reach, meaning that not all goals can be occupied at the same time. In order to solve a Sokoban problem, an algorithm (as well as a human player) must be able to recognize these situations early to prune them and reduce the search space. A few examples of deadlocks are represented in Figure 2.2.

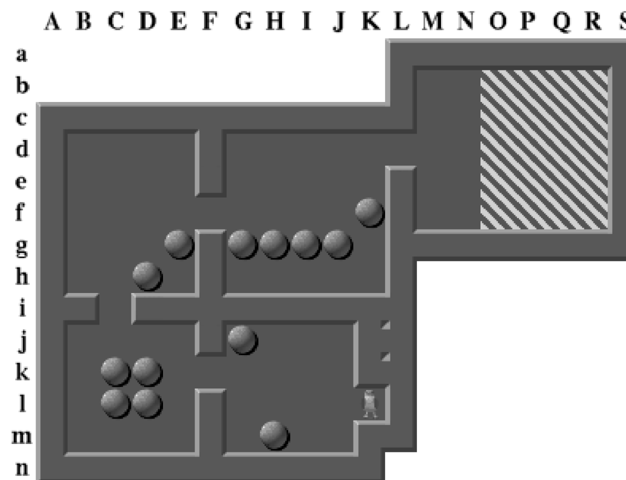


Figure 2.2: Deadlock examples: upper-left: Corral deadlock; upper-right: Corral deadlock; lower-left: Freeze deadlock; lower-right: Simple deadlock [3]

2.1.2 Complexity

Assuming that our goal is to minimize the number of pushes, we can analyze the game by considering only push moves on boxes

that are reachable from the player current position. Considering that in the standard set the number of boxes ranges from 6 to 34 and that we can have up to 4 moves per box, the branching factor can potentially reach a maximum of 136. Of course part of the complexity of Sokoban relies on the constraints that come from the risk of creating deadlocks, so in most of the levels the boxes are densely packed, meaning that the number of available moves is considerably lower. Level 1 (Figure 2.1), with only 6 boxes is one of the easiest levels. We can compute an average branching factor b of 3.07, and with a solution length d of 97, the game-tree complexity becomes approximately 10^{47}. Suppose that we have a similar boxes-to-moves ratio for all levels, with an average number of boxes of 16 and an average solution length of 284.8, we can roughly estimate the average game-tree complexity as 10^{260}. Since the game rules don't contemplate a limit in the number of moves, and the pusher can retrace its steps, the game length can be considered potentially unlimited. Therefore, to solve a Sokoban puzzle, one should avoid cycles during the search. Junghanns et al. [3] computed the upper bound for the state-space complexity of Sokoban on a 20×20 board with walls on the perimeter, as $\binom{s}{b}m$ where s is the total number of squares, b is the maximum number of boxes and m is the number of possible pusher positions. This yields a result of 10^{98}. With an average number of squares of 77 (excluding squares that would cause simple deadlocks) and an average number of boxes of 16, the average state-space complexity of the standard levels suite is approximately 10^{18}. Sokoban has also been shown to be NP-hard and P-space complete [6] and is considered challenging for both humans and programs.

2.2 Iterative Deepening A*

*Iterative Deepening A** is one of the most successful methods for puzzle solving that can be found in the literature and it is based on a classical planning approach.

2.2.1 Overview

*Iterative Deepening A** (IDA*) is a variant of *Iterative Deepening Depth-First Search* (IDDFS) that relies on heuristic evaluation to determine the threshold for the cut point of the current iteration. It was described by Richard Korf in 1985 [7]. IDDFS is an uninformed search method in which a depth limited depth-first search is repeatedly executed with increasing depth limit until a solution is found. At each iteration a full depth-first search is performed until either a solution is found or the entire search tree has been explored up to the depth limit d. If no solution has been found, d is increased and the search is repeated.

The method inherits the low memory complexity of depth first search, while retaining completeness even in presence of unlimited trees. Optimality is preserved if all actions have constant cost.Similarly to IDDFS, IDA* proceeds in a depth-first manner with a depth bound, and once the whole tree for that bound has been explored, the search restarts with an increased bound. Unlike IDDFS, the threshold is not defined in terms of tree depth, but of heuristic evaluation. The method used to determine if the search can continue is based on A* evaluation. A* is an informed search algorithm first proposed in 1968 by Hart et al. [8] as an improvement over Dijkstra's algorithm for finding the minimum cost path in a weighted graph. At each iteration it explores the node that minimizes

$$f(n) = g(n) + h(n) \tag{2.1}$$

where $g(n)$ is the cost of reaching node n from the initial node and $h(n)$ is a heuristic estimate of the cost from node n to the goal.

In IDA* as in A*, in order to guarantee the optimality of the solution, the evaluation function h must satisfy the following conditions:

- Admissibility: $h(n) < h^*(n)$ with $h^*(n)$ being the perfect heuristic (the actual cost from n to the goal in the optimal solution.

- Consistency: $h(n) \leq g(n') + h(n') - g(n)$ where n' is a successor of n.

If these condition are satisfied, and a solution is found, that solution is guaranteed to be optimal [8].

2.2.2 The Algorithm

IDA* initializes the search threshold as the heuristic value of the initial state $h(n_0)$. It then proceeds in a depth-first exploration until either the value $f(n) = g(n) + h(n)$ exceeds the threshold or there are no more successors. When one of these condition is satisfied, the search returns to the parent node and expands the siblings of the last explored node, exactly like in IDDFS. This produces an asymmetric search tree that grows guided by the heuristic function. The main advantage of this approach with respect to classic A* algorithm is that it runs in space that is linear in the maximum search depth, rather than exponential. Pseudocode for IDA* is showed in Algorithm 1.

2.2.3 State of the art

IDA* has reached good results in Sokoban [3] and in the 15-Puzzle (and derivatives) [9]. In the context of Sokoban, the only documented method that can be considered state of the art is the program *Rolling Stone* [3]. It implements domain independent enhancements [10] to prune the search tree and domain specific enhancements that make use of lower level searches to avoid deadlocks, obtain a tighter lower bound and reduce the search space.

Transposition Tables

Transposition tables consists of data structures used to store information about visited states and are used to avoid cycles during the search and to reduce the branching factor. In Rolling Stone the heuristic evaluation of nodes is stored in the transposition table and updated according to the result of previous iterations. This allows the algorithm to improve the heuristic values and prune sub-trees more efficiently.

Algorithm 1 IDA*

```
function IDA*(s_0)
    create root node n_0 with state s_0
    create solution path with n_0
    threshold ← h(n_0)
    while result not found do
        value ← Search(path, 0, threshold)
        if value = RESULT then return path
        if value = ∞ then return failure
        threshold ← value
function Search(path, g, threshold)
    n ← Last(path)
    f ← g + h(n)
    if f > threshold then return f
    if IsGoal(n) then return RESULT
    min ← ∞
    successors ← GenerateSuccessors(n)
    for all child in successors do
        Push(path, child)
        cost ← ActionCost(child, n)
        t ← Search(path, g + cost, threshold)
        if t = RESULT then return RESULT
        min ← MIN(min, t)
        Pop(path)
    return min
```

Move Ordering

Children of a node are ordered based on the likelihood of leading to a solution. The move ordering scheme proposed was the following:

1. *Inertia* moves are considered first. Inertia moves are those moves that preserve inertia, meaning that they act on the same stone as the previous move.
2. Then all moves that decrease the lower bound are tried (optimal moves), sorted by distance from the stone to its target goal.
3. Finally, all non-optimal moves are tried, also sorted by distance to target goals.

Deadlock Tables

Deadlock tables make use of pattern database [11] to match the current situation to precomputed tables of possible deadlock configurations. The tables are computed offline with all possible combinations of walls, stones and empty squares for a fixed-size region. The deadlock tables are implemented as decision trees, with internal nodes representing subpatterns and leaves representing whether the pattern is a deadlock or not. Junghanns et al. [3] built two tables for regions of roughly 5x4 squares, that differed in the order the squares in the maze are queried.

Tunnel Macros

A *tunnel* is defined as a part of the maze where the maneuverability of the pusher is restricted to a width of one. These regions cannot have more than one box inside, otherwise they would cause a deadlock. In order to reduce the branching factor, we can collapse all pushes of a box inside a tunnel in a single move that makes the box exit that tunnel.

Goal Macros

When the levels have all goal squares concentrated in a small area, if there are few entrances to this area, the problem can be decomposed

in two sub-problems:

- moving boxes to the entrances
- placing boxes on goals

In some cases these two parts can be solved separately. With Goal Macros, the order in which goal squares must be filled to avoid deadlocks is precomputed, and when a box reaches an entrance the only move it can perform is the Goal Macro, that pushes it directly into its assigned goal.

Goal Cuts

The move pruning of Goal Macros is backpropagated to previous states when a stone is pushed to a square with a Goal Macro at the end without interleaving other stone pushes.

Pattern Search

Pattern search uses sub-problem searches to improve the lower bound and to identify deadlock patterns. It consists of repeated IDA* searches with patterns of more and more boxes. When a deadlock pattern is found, it's saved and used throughout the search in addition to those in the deadlock tables. The basic algorithm performs the following steps:

1. Create a test maze with the same walls and goals configuration but containing only the last moved box.
2. Try to find a solution to the test maze.
3. If no solution is found, the pattern is a deadlock. Return pattern.
4. If a solution is found, add a box that is on a square that is needed for the solution. A square is needed for the solution if either the box or the man had to go through it in the solution found.
5. If search effort is not exhausted, repeat from 2.

Therefore the pattern search can terminate if the effort is reached, a deadlock was detected or no more stones can be added. In particular, three types of specialized search has been implemented:

- Deadlock Search: specialized in finding deadlocks, it ignores states that are less likely to contain a deadlock to reduce the cost and be able to include more boxes in the search.

- Penalty Search: specialized in finding conflicts between boxes, is not allowed to take shortcuts, hence it might discover patterns ignored by the deadlock search. It's therefore more expensive and it evaluates less boxes with the same search effort.

- Area Search: specialized on finding deadlocks, it focuses on areas that are not accessible to the man. It incrementally includes all boxes that surround those areas, trying to find a deadlock. This is similar in concept to the PI-Corral pruning of the YASS solver [12].

Relevance Cuts

Relevance cuts is a forward pruning methods that removes moves that are considered not relevant. A move is relevant only if the previous m moves influence it. The influence metric is defined based on the position of the boxes as follows:

- Alternatives: The more alternatives exist on a path between two squares, the less the squares influence each other.

- Goal-Skew: for a given square sq, squares on the optimal path from sq to its goal have a stronger influence than those off the optimal path.

- Connection: Two adjacent squares between which a stone can be moved freely influence each other more than two squares between which only the man can move freely.

- Tunnel: Influence remains constant inside a tunnel.

Overestimation

Overestimation is based on the idea that a good non admissible heuristic might be closer to the optimal value than a poor admissible heuristic. Given the complexity of the problem, non optimal solution are acceptable. Overestimation allows every pattern found during pattern search to increment the lower bound. This will postpone "difficult" situations, giving up optimality but preserving completeness.

Chapter 3

Monte Carlo Tree Search

In this chapter we present the second chosen algorithm *Monte Carlo Tree Search*, giving a description of this algorithm and of its improvements which can be found in literature. We also present and briefly describe rules and complexity of Samegame, a puzzle domain where *Monte Carlo Tree Search* achieve good results.

Traditional artificial intelligence methods require a good heuristic function to evaluate moves and could be unsuitable for those domains in which defining an heuristic using expert knowledge is particularly difficult. In these situations, Monte Carlo methods can make up for the lack of a solid heuristic by obtaining an approximate evaluation of the game-theoretic value of a move, by relying on the use of simulated playouts. The value of action a in state s can be expressed as

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{N(s)} I_i(s,a) z_i$$

Where $Q(s,a)$ is the action-value function, $N(s,a)$ is the number of times action a has been selected from state s, $N(s)$ is the number of visits through state s, $I_i(s,a)$ has value 1 if action a was selected from s in the $i-th$ playout and z_i is the final reward of simulation i. Essentially, the value of an action is computed as the average reward obtained in the playouts in which that action was taken.

Monte Carlo evaluation was initially applied to tree search as an alternative to ad-hoc evaluation functions to prune the search tree

[13] [14]. These methods had the drawback of having no game-theoretic guarantees on the optimality of the solution. Monte Carlo evaluation was also applied to Markov Decision Processes with better asymptotic properties [15] [16]. First visit Monte Carlo evaluation for example, is an unbiased estimator for the action-value function, meaning that given enough samples, it will converge to the actual function.

The algorithm known as Monte Carlo Tree Search (MCTS) [1] was first proposed as an alternative to min-max trees with Monte Carlo evaluation and implemented in *Crazy Stone*, a Go-playing program that won the 10th KGS computer-Go tournament. Over the years it has been successfully used in two-player zero-sum games with high branching factor like Go, Hex and Lines of Action [17].

The next major step in MCTS development was the proposal of combining the *Upper Confidence Bound* (UCB1) algorithm of Multi-Armed Bandit problems to MCTS. This created what we now know as *Upper Confidence Bounds for Trees* (UCT) [2]. The UCB1 formula is used to balance exploration and exploitation during the tree search. Then, once reached a leaf node, a simulation is executed and the result is backpropagated to the root. UCT is now the most used among the Monte Carlo methods.

3.1 State of the art

Most of the research on MCTS has been carried out in the context of the game Go, an high branching factor two-players zero-sum game with no reliable heuristics for non-terminal positions. In 2016 Silver et al. [18] proposed a method that combined MCTS with convolutional neural networks and their program *AlphaGo* was able to defeat 18-times human world champion Lee Sedol 4-1. They continued their research and while AlphaGo used expert knowledge to train the neural networks, their *AlphaGo Zero* [19] program was developed with no supervised learning, and was able to beat the original Alpha Go 100-0.

Another board game in which MCTS has been successfully used is Hex that, unlike Go, has a robust evaluation function for the

intermediate states; that is the reason why is possible to create good artificial intelligence using alpha-beta pruning techniques [20]. In 2007, Arneson et al. [20] developed a program based on Monte Carlo Tree Search, able to play the board game Hex. The program, called *MoHex*, was able to win the silver and the gold medals at Computer Olympiads in 2008 and 2009 respectively, showing that it is able to compete with the artificial intelligence based on alpha-beta pruning.

In the context of puzzle games, the research in MCTS methods is less developed and relatively recent. In the Samegame puzzle (Section 3.6.1) the top score in a 20 levels set [21] is currently held by tcooke with an undocumented method while the best score among documented algorithms was obtained by Edelkamp et al. with their *Heuristically Guided Swarm Tree Search* [22] algorithm, a parallelized version of MCTS. Great results were also obtained by Schadd et al. [23] with a variant of the classic UCT algorithm called *Single-Player Monte Carlo Tree Search*.

Another puzzle in which MCTS methods excel is the Morpion solitaire. The current top scores of 178 and 82 respectively on the 5-T and 5-D versions of Morpion are held by Rosin and his *Nested Rollout Policy Adaptation* [24], an extension of *Nested Monte Carlo Search* [25] in which the rollout policy is tuned adaptively starting from a uniform policy.

Guez et al. [26] proposed a method that combined neural networks and MCTS and trained it on the outcome of a MCTS algorithm on a set of very simple Sokoban levels (10×10 grid with a maximum of 4 boxes). As a result, after the offline supervised training, their method was able to reach the same performance as the baseline MCTS method but with far fewer iterations. The paper publication was eventually rejected due to the fact that the trained NN was considered to unlikely be able to generalize to other level configurations, in addition to the fact that the complete application of the method (offline and online phases) required far more resources than the baseline MCTS.

3.2 The Algorithm

The MCTS algorithm builds an asymmetric search tree based on the results of Monte Carlo simulations. The tree growth is guided by the estimates it provides. It's an anytime algorithm, meaning that it can be stopped at any point in its execution and it will provide the best action for the root state so far. Its estimate values for the action become more precise as the algorithm continues its execution. The execution can be divided in four steps per iteration, as shown in Figure 3.1:

1. *Selection*: starting from the root node, the algorithm recursively selects a child node according to some policy that should balance exploration and exploitation, until it reaches a node that has not been fully expanded, meaning not all of its moves lead to another node. The selection phase might also end if it reaches a terminal state;

2. *Expansion*: According to the expansion policy, one or more nodes - corresponding to the execution of the current node unexplored actions - are created and added to the tree;

3. *Rollout*: a simulated game is played, starting from the newly created node (the *leaf node*), until a terminal state. This simulation is executed according to the default policy and it produces a reward;

4. *Backpropagation*: the reward obtained during the rollout phase is backpropagated through the tree, starting from the leaf node, upwards towards the root node. Each node contains the sum of the rewards of its children and the visit count (the number of times the node has been visited during the search).

The algorithm repeats this four steps until the end condition is met: this can be either a time constraint, a memory constraint or a limit in terms of number of iterations. At this point, the algorithm selects the best action for the root node according to a chosen criteria. Schadd [28] describes four criteria for selecting the winning action, based on the work of Chaslot et al. [27]:

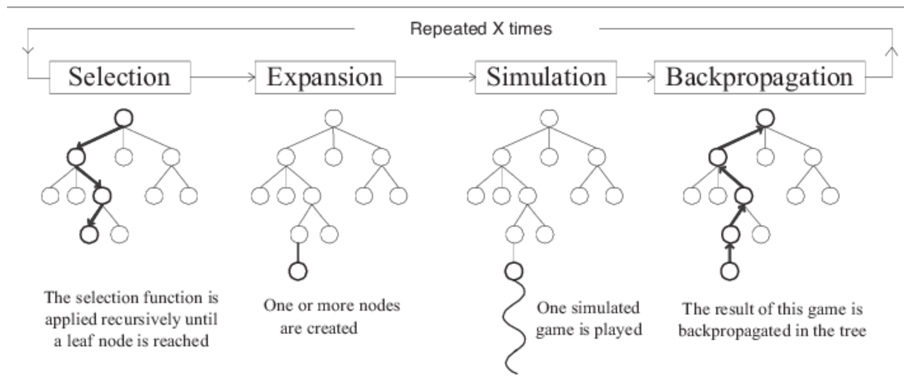


Figure 3.1: Steps of the Monte Carlo tree search algorithm [27]

- *max child*: selects the child with the highest reward;
- *robust child*: selects the most visited root child;
- *max-robust child*: select the root child with both the highest visit count and the highest reward;
- *secure child*: select the child which maximizes a lower confidence bound.

A MCTS algorithm is thus defined by the mentioned criterium, and by the policies used during the search. In the four steps of the algorithm we can identify four different policies:

- *Selection policy*: used to determine the way in which the tree is explored in the selection phase;
- *Expansion policy*: used to determine which nodes are created in the expansion phase (usually one or all);
- *Simulation policy*: used to determine the rollout behaviour. The usual default policy is a random policy, in which moves are sampled from a uniform distribution among those available in the current state, but often a policy handcrafted for the specific domain can obtain better performance;
- *Backpropagation policy*: used to determine how the rewards are propagated in the tree. In the classical version of the

method the reward is added to the sum of the rewards of a node and the number of visits is incremented.

A general MCTS approach is summarized in Algorithm 2. In this algorithm v_0 is the root node corresponding to state s_0, v_l is the last node reached during the tree policy stage and corresponds to state s_l, and Δ is the reward for the terminal state reached by running the default policy from state s_l. The result of the overall search a(BESTCHILD(v_0)) is the action a that leads to the best child of the root node v_0, where the exact definition of "best" is defined by the implementation.

Algorithm 2 **General MCTS approach**

```
function MctsSearch(s0)
    create root node v0 with state s0
    while within computational budget do
        vl ← TreePolicy(v0)
        Δ ← DefaultPolicy(s(vl))
        Backup(vl, Δ)
    return a(BestChild(v0))
```

3.3 Upper Confidence Bounds for Trees

Upper Confidence Bounds for Trees (UCT) is a version of the MCTS algorithm that uses UCB1 as a tree policy. The selection of a child node is therefore treated as a multiarmed bandit problem, in which the rewards correspond to random variables with unknown distributions. A child node i is thus selected to maximize

$$UCT = \frac{v_i}{n_i} + C \times \sqrt{\frac{2 \ln n_p}{n_i}} \quad (3.1)$$

where v_i is the sum of the rewards obtained in all rollouts that have passed through node i, n_i is number of times the node i (child of p) has been visited, n_p is number of times the current node has been visited and C is a constant used to balance exploration and exploitation. This formula is intrinsically balanced between the

exploitation and exploration (represented respectively by the first and second term) as the number of visits of a node n_i increases (inevitably together with n_p), the second term decreases, while when another child of the same parent node is visited, only n_p in the numerator increases. The constant value can be adjusted to favor exploration or exploitation. The value of $C = 1$ was shown to ensure the asymptotic optimality of the solution with rewards the range [0,1] [29]. With rewards outside this range, appropriate values of C could be found by manual tuning or other automated method [30]. Algorithm 3 shows the UCT algorithm in pseudocode.

Each node v contains information about: the associated state $s(v)$, the outgoing action $a(v)$, the total simulation reward $Q(v)$ and the visit count $N(v)$. To have a more efficient usage of memory, the associated state $s(v)$ is recomputed as `TREEPOLICY` descends the tree, instead of storing it for each node. The term $\Delta(v, p)$ in function `BACKUP` indicates the component of the reward vector Δ associated with the current player p at node v. The return value of the overall search is given by $a($`BESTCHILD`$(v_0))$, that is the action a associated to the child which has the highest reward, due to the fact that the exploration parameter c is set to 0 for the final call of this function on the root node v_0.

3.4 Characteristics

MCTS is a popular choice of algorithm for a variety of domains due to three main characteristics:

1. *Analytic*: MCTS does not require any domain-specific knowledge, it is sufficient to know only its legal moves and end conditions. This makes it applicable to any domain that can be modeled using a tree. However, in its basic version, MCTS can have low performance and some domain-specific knowledge can be included in order to significantly improve the speed of the algorithm;

2. *Anytime*: MCTS backpropagates the outcome at the end of every iteration, so the whole tree is immediately updated with

Algorithm 3 UCT Algorithm

function UctSearch(s_0)
  create root node v_0 with state s_0
  while within computational budget **do**
    $v_l \leftarrow$ TreePolicy(v_0)
    $\Delta \leftarrow$ DefaultPolicy($s(v_l)$)
    Backup(v_l, Δ)
  return a(BestChild($v_0, 0$))
function TreePolicy(v)
  while v is nonterminal **do**
    if v not fully expanded **then**
      return Expand(v)
    else
      $v \leftarrow$ BestChild(v, C_p)
  return v
function Expand(v)
  choose $a \in$ untried actions from $A(s(v))$
  add a new child v' to v
   with $s(v') = f(s(v), a)$
   and $a(v') = a$
  return v'
function BestChild(v, c)
  return $\underset{v' \in children\,of\,v}{\text{argmax}} \frac{Q(v')}{N(v')} + c \times \sqrt{\frac{2 \ln N(v)}{N(v')}}$
function DefaulPolicy(s)
  while s is non-terminal **do**
    choose $a \in A(s)$ uniformly at random
    $s \leftarrow f(s, a)$
  return reward for state s
function Backup(v, Δ)
  while v is non null **do**
    $N(v) \leftarrow N(v) + 1$
    $Q(v) \leftarrow Q(v) + \Delta(v, p)$
    $v \leftarrow$ parent of v

the last calculated rewards and visits counts. This allows the algorithm to stop and return the current best root action at any moment in time. Allowing the algorithm to run for extra iterations often improves the result;

3. *Asymmetric*: The tree policy allows the algorithm to use more computational resources on the most promising nodes of the tree, leading to an asymmetric growth of it over time. This makes the tree adapt to the topology of the search space leading to a better understanding about the game itself and making MCTS suitable for games with high branching factor.

3.5 MCTS Enhancements

A huge number of enhancements have been proposed for the core MCTS algorithm in order to improve performance, including modifications of the tree policy, the default policy and other more general modifications related to the backpropagation step and parallelization as described in Cameron et al. work [31]. These enhancements can generally be divided into two categories: *Domain independent* enhancements that do not require any prior knowledge about the domain in order to be applied in it; *Domain dependent* enhancements that are specific to particular domains. A more specific categorization can be done considering the used approach.

Bandit-Based Enhancements

This approach modifies the bandit-based method used for node selection in the tree policy. For this kind of enhancements plenty of different upper confidence bounds have been proposed, often improving bounds or performances in particular circumstances.

Selection Enhancements

This approach modify the tree policy in order to change how MCTS explores the search tree. The basic idea of selection is to assign a numeric score to each action in order to balance exploration with exploitation, so these enhancements influence this score using some

domain knowledge in order to bias the search toward the most promising area and make use of different kind of reward estimation.

All Moves As First (AMAF)

This approach modifies how node statistics are updated. The basic idea is to update statistics for all action that are selected during a rollout as if they were the first action applied. The AMAF algorithm treats all moves played during selection and rollout steps as if they were played on a previous selection step, meaning that the reward estimate for an action a from a state s is updated whenever a occurs during a playout, even if a was not the actual move chosen from s.

Game-Theoretic Enhancements

This approach uses the known game-theoretic value of a state to improve reward estimates for other non-terminal nodes backpropagating it up the tree.

Move Pruning

This approach applies the pruning technique, already used with *minimax*, in order to eliminate obviously poor choices and allow the search to focus more resources on the better ones.

Simulation Enhancements

This approach modifies the default simulation policy for MCTS which select randomly among the available actions, trying to obtain more realistic simulations by incorporating domain knowledge into the playouts.

Backpropagation Enhancements

This approach modifies the backpropagation step involving special node updates.

Parallelization Enhancements

This approach exploit the independent nature of each simulation in MCTS to introduce parallization. More simulations can be performed in a given amount of time and the wide availability of multicore processors can be exploited. However, parallelization raises issues such as the combination of results from different sources in a single search tree, and the synchronization of threads of different speeds over a network.

3.6 MCTS for puzzles

UCT was initially proposed for two-player games, therefore — as in minimax and other adversarial search algorithms — some of its characteristics were based on the assumption that there was a second player with the goal of minimizing our reward. Schadd et al. [23] presented a variant called *Single-Player Monte-Carlo Tree Search* (SP-MCTS) in which they exploit some properties of single-player games to increase the effectiveness of MCTS. The core mechanism of SP-MCTS is the same as in UCT. One of the new ideas introduced is a variation the selection strategy, which aims to maximize the following formula:

$$SP-UCT = \frac{v_i}{n_i} + C \times \sqrt{\frac{2 \ln n_p}{n_i}} + \sqrt{\frac{\sum v_i^2 - n_i \times (\frac{v_i}{n_i})^2 + D}{n_i}} \quad (3.2)$$

where the first two terms represent the UCT formula 3.1, while the newly added term represents possible deviations in the child node scores. Using the same notation used in the UCT formula, $\sum v_i^2$ is the sum of squared rewards obtained by rollouts that have transited through node n_i, $n_i \times (\frac{v_i}{n_i})^2$ is the expected squared reward, D is a large constant used to ensure that rarely explored nodes are considered promising.

One of the aspects in which puzzles can differ from zero-sum two-player games is the range of values for the reward. While in two-player games the outcome is usually defined as {0, $\frac{1}{2}$, 1} or {-1, 0, 1} for loss, draw or win, in puzzles where the goal is to maximize

the score the reward can vary greatly in range. Keeping in mind that [23] chose Samegame as a case study (which has a score in the order of thousands), the third term of the formula was specifically designed to take advantage of the score variation. Another major difference from two-player games is the absence of an opponent. The consequence is that if the game is deterministic there is no uncertainty given by the opponent choices. This leads to the second variation introduced in SP-MCTS: the method to choose the root action at the end of the search is not one of the four mentioned in Section 3.2 which generally aim towards the maximization of the winning probability, but instead the selected action is the one that belongs to the path which maximizes the top score. In order to employ this policy and to evaluate equation 3.2 the backpropagation policy was slightly modified to store in the nodes additional information like the total squared rewards and the top score. The best solution, i.e. the one with the highest score, was also stored throughout the search to ensure that the final result is the best among all rollouts executed.

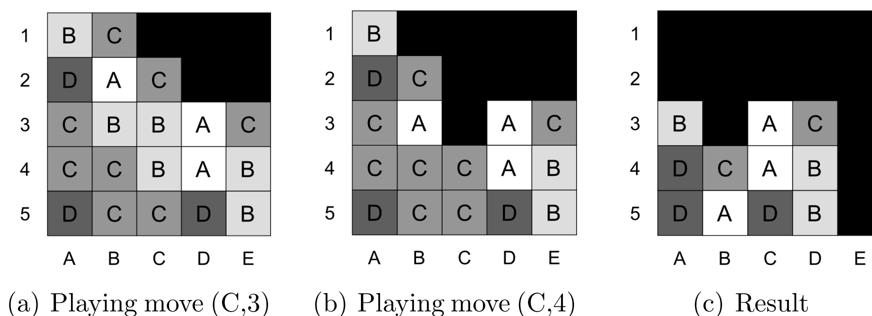
Schadd et al. [23] also applied a single-threaded version of root parallelization [32] called *Randomized Restarts* in which the search is repeated multiple times with different random seeds to avoid getting caught in local optima. They used the *Cross Entropy Method* [30] to tune the parameters and managed to obtain a score of 78012 in a standardized Samegame test set.

3.6.1 Samegame

Samegame is a fully observable deterministic single player game invented by Kuniaki Moribe under the name *ChainShot!* in 1985. In 1992 Eiji Fukumoto ported the game to Unix with a slight variation in settings calling it Samegame. Samegame is a tile-matching game and numerous variations of the game have been developed over the years.

Rules

Samegame is played on an $n \times m$ grid containing randomly arranged blocks of k different colors. The goal of the game is to obtain the highest score by removing groups of blocks. At every move, the player can remove a group of two or more orthogonally adjacent blocks of the same color. Once a block has been removed, all blocks above it in the same column will slide down until there are no holes left, as if they were subject to gravity, as shown in Figure 3.2(b). Once an entire column is removed, all columns on its right will slide to the left until there are no empty columns left, as shown in Figure 3.2(c).



(a) Playing move (C,3) (b) Playing move (C,4) (c) Result

Figure 3.2: Move sequence example [23]

Points are rewarded for each action according to the formula $(n-2)^2$, where n is the number of blocks removed with the current action. The game is over when either the player has cleared the board, leaving no blocks behind, or there are no more legal moves left, meaning that there are no two blocks with the same color are adjacent to each other.
Final points are awarded or subtracted based on the above end game condition:

- If the player was able to clear the board: A bonus of 1000 points is awarded.
- If the player was unable to clear the board: Points are subtracted using the formula $\sum (n_i - 2)^2$ where n_i is the number of blocks of the same color still on the board.

The standard game configuration is a 15×15 board with 5 colors. Given the scoring system, the maximum theoretical achievable score is $(n \times m - 2)^2 + 1000$, with n and m being rows and columns. With the standard configuration this results in a total of 49829. This is only obtainable if the initial configuration contains only a single color, requiring that no constraint on the minimum number of colors present are applied. If we consider enforcing the presence of all 5 colors, the maximum score becomes 48861. However, the actual score for a randomly generated board is rarely higher than 6000 points. On the 20 levels standard test set used in most researches [23] [25] [24], the average score of the best algorithm is 4392.1 [21].

The same mechanics can be found in other games, with the only difference often being the scoring formula. In *Clickomania!* for example the objective is minimizing the number of blocks remaining, thus clearing the board represent the top score. In *Jawbreaker*, a the Samegame porting for Microsoft Windows Mobile 2003, the formula used is $n(n-1)$, while in the Windows 3.1 porting the game is still called Samegame, but it computes the score as $n^2 - 3n + 4$.

Complexity

A good estimator for the complexity of a game is the game-tree complexity, that represents the number of leaf nodes in the search tree and can be approximated by b^d, where b is the average branching factor and d is the average game length. In Samegame the player can choose a block as a move and in a 15×15 board that would lead to an initial branching factor of 225. In reality, all moves in the same block group are equivalent so the actual branching factor is normally smaller. It also generally decreases as the game progresses. Schadd et al. [23] calculated the average game length and the average branching factor over 250 different configurations, and obtained $d = 62.2$ and $b = 21.1$, resulting in a game-tree complexity of 10^{82}.Another estimator for the complexity can be the state-space complexity, which is the total number of legal board configurations reachable from the initial state. In the case of Samegame, it can be computed as C^m where m is the number of columns and C is the number of possible configurations of a single column, obtained as

$C = \sum_{i=0}^{n} k^i$, where n is the number of rows and k is the number of different colors. With $n = 15$, $m = 15$ and $k = 5$ we can obtain a state-space complexity of 10^{159}. Furthermore, Samegame has been proven to be at least of complexity class NP-complete [23].

In addition to the sheer values of game-tree and state-space complexity - that render uninformed search methods unfeasible - it's very hard to find a reliable admissible heuristic, given that with every move, new block groups can be created and existing groups can be dismantled.

Chapter 4

Our Approach

In this chapter we present our solution to the puzzle solving problem. The proposed solution takes into account IDA*, MCTS and some enhancements of standard versions of these algorithms. We describe the enhancements implemented and added to the basic algorithms and to Sokoban. First, we describe the implemented IDA* optimizations; next we describe optimizations for the MCTS algorithm. The enhancements are either domain independent, which can be applied to each algorithm and to both the considered domains, or domain dependent, specific for the different domains.

4.1 IDA* optimizations

The performance of IDA* is strongly tied to the quality of the heuristic evaluation, however there are techniques that can improve the results under certain conditions by reducing the size of the search tree or the order in which nodes are explored.

4.1.1 Transposition Tables

Transposition tables are a data structure that keeps track of the visited states in order to reduce the branching factor of the game tree. The assumption at the base of the use of transposition tables is that the same state can be reached through different paths. When this assumption holds, transposition tables can be used to avoid cycles during the search by expanding only nodes that represent a

state that has not yet been visited. In addition to cycles avoidance, we can use transposition tables to prune sub-trees according to the outcome of previous searches.

A transposition table requires: an entry, a record that represent the state and data relevant to the search; an hash table, the actual table that holds an entry in each cell.

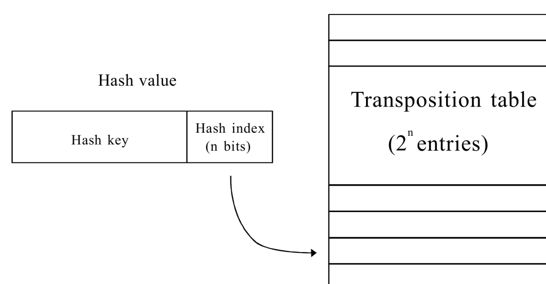


Figure 4.1: Hash Table [33]

Figure 4.1 shows the typical usage of transposition tables. The hash value is computed through an hash function specifically built for the domain we are considering. The perfect hash function should be built in such a way that two states that should be considered different for the search purpose have different hash values, while two state that should be considered equivalent have the same hash value. Once computed, the *hash index*, a portion of the hash value of a state, is used as an index to access the table, while either the remaining portion of the hash called *hash key*, or the full hash value is stored inside the table to handle collisions that might happen between different states that map onto the same index. Therefore, the table entry should contain the either the hash key or the hash value, along with all data that is relevant to the search. In the case of IDA*, this data include

- *Score*: the heuristic value obtained in previous visit through the state. When the state is first added to the transposition table, the score should be its heuristic value. Once the sub-tree that starts from the state has been completely explored up to the threshold, the score oh the state should be updated with the score obtained further along the search, since

it's supposed to be more accurate. In the case of IDA*, it's computed as the minimum value among the successors of the current node.

- *Depth*: the relative depth of the explored sub-tree. This represents how deep the tree has been explored, starting from the current state. In the context of IDA*, the depth can be computed as $threshold - g$, where $threshold$ is the threshold of the current iteration and g is the cost of the path from the root state to the current state.

- *Visited*: a flag that represent whether the entry is part of the current path or it has been added in a previous search. This allows us to discern between cycle avoidance and tree pruning.

In an ideal setting, the transposition table size would be enough to hold every state visited during the search. In real applications, such as Samegame and Sokoban, with large branching factors, we limit the size of the table as shown in Figure 4.1. A replacement scheme represent the policy used to handle conflicts. The replacement scheme defines whether upon collision, the entry contained in the table should be kept or should be replaced with the new entry. Breuker [33] examined several different replacement schemes based on different concepts:

- *Deep*: The entry with the largest depth is kept in the table. The rationale behind this scheme is that the sub-tree associated to an entry with a great depth contains more nodes than the lower depth one, thus containing more accurate information and saving more work in case of pruning.

- *New*: The newest entry is kept in the tree. This scheme is based on the observation that most transposition occur in local sub-trees, implying that keeping most recent nodes in the transposition table allows us to prune more often.

- *Old*: The oldest entry is kept in the tree. No replacement occurs. The author included this scheme only for the sake of completeness.

- *Big*: The entry with the largest amount of nodes in the sub-tree is kept in memory. This is similar to the Deep scheme and might perform better when the depth is not a good estimator of the size of the sub-tree, but requires to keep in the entry also the number of nodes, effectively reducing the number of entry that can be stored in the table.

- *Two-Level*: The transposition table is organized in two levels and the replacement scheme is combined with one of the above. The the main replacement scheme is applied with the selected entry stored in the first level of the table, while the entry that would normally be discarded is stored on the second level. When retrieving the entry, the second table is used only if there's no match in the first table. This scheme allows us to store both more recent (to increase the number of hits) and more informative entries (to prune large sub-trees). Note however that keeping two tables imply that the number of entry stored for each table is halved.

Breuker [33] results showed that the two-level replacement scheme obtains the highest reduction in number of explored nodes, with its Big combination having a slight edge over the Deep one. We decided however to follow Junghanns et al. [3] example and implemented a two-level transposition table based on depth, since it lends itself well for use with IDA*, as depth can be computed before actually exploring the sub-tree, as mentioned earlier.

As we already mentioned, the effect of transposition tables on IDA* is two-fold: cycles avoidance and tree pruning. In practice, the search explores a new node, it first checks the transposition table for an entry with the same state. If one is found and its depth is greater than the current estimated depth, the node is not explored further and its value is returned as the score of the entry. Otherwise, the a new entry is stored and the search continues among the successors (Algorithm 1), with the exception that only child nodes that do not appear as visited are explored.

4.1.2 Move ordering

IDA* explores in a depth first manner, so if the algorithm explores promising nodes first, it will find a solution faster. Following this idea, move ordering sorts the available moves before expansion according to a certain criterion. This criterion can be domain dependent, as in [3], or domain independent, as in [10]. We decided to implement move ordering with a domain independent ordering criterion by sorting the moves according to the heuristic evaluation of the resulting state. Since in IDA* all iterations except for the last one perform a complete search, the only contribution of move ordering is visible on the last iteration, but since that is the largest one by number of nodes, if the solution is found in an earlier sub-tree it may reduce execution time by a significant amount.

4.2 MCTS configuration

To identify a specific MCTS configuration we need to define simulation policies and the rewards used. We defined four different reward types for Sokoban:

1. *R0*: 1 if the state represents a solved level, 0 otherwise.
2. *Boxes*: the reward is the same as the one used by [34] and is computed as

   $$r = 0.1 \times steps + b_{ON} - b_{OFF} + solved \tag{4.1}$$

   where *steps* is the number of pushes executed, b_{ON} is the number of times a box has been pushed on a goal, b_{OFF} is the number of times a box has been pushed off a goal and *solved* is 10 if the level is solved, 0 otherwise.
3. *InverseBM*: $1/\sqrt{BM}$, where BM is the *minimum cost perfect matching on a complete bipartite graph*, i.e. the minimum sum of the distances from each box to its designated goal. BM is the heuristic evaluation used by [3]. This reward has the advantage of having values in the range {0,1}.

4. *NegativeBM*: $-BM$. This reward has values in the range $\{-\infty,0\}$, but has the advantage of the linearity of the reward.

We tested different simulation policies:

- *random*: selects a random action among those available in the current state.
- *ϵ-greedy*: selects a random action with probability ϵ or the action that maximizes the reward of the resulting state with probability $1-\epsilon$.
- *ϵ-IDA**: selects a random action with probability ϵ or – with probability $(1-\epsilon)$ – perform an IDA* search with a limited number of nodes and return the first action of the path that leads to the state with the lowest heuristic value. For this policy we tested different configurations in terms of number of IDA* nodes and MCTS iterations.

For the backpropagation policy, in addition to the usual sum of rewards and number of visits we stored the maximum score, the sum of the squared rewards and the sum of rewards and visits for the RAVE optimization (Section 4.3.5). The winning action is the one with the highest maximum score.

4.3 MCTS optimizations

Standard version of the core MCTS algorithm can be applied to various domains due to its main characteristic of not requiring domain-specific knowledge, but when it is needed to compare it - in a specific domain - with another artificial intelligence modified in order to have good performance in such domain, it can work not so well. In order to achieve better results in this kind of comparison, we need to modify the standard algorithm introducing some enhancements in order to improve performance.

In the proposed solution all enhancements are domain independent, so can be applied to any domain without prior knowledge about it, this choice is done in order to have a unique optimized MCTS

algorithm that can be used in each game considered in this work without the need of any change due to some specific knowledge.

4.3.1 Object Pooling

The Object Pool pattern is a software creational design pattern used to improve memory usage and performance. This pattern uses a set of initialized objects kept ready to use — also called "pool" — rather than allocating and destroying them on demand. In order to use this pool, it is possible to require an object from it and then perform operations on the returned object. When the object is not used anymore, it can be returned to the pool, in order to have the possibility to use it again later, rather than destroying it; this can be done manually or automatically.

The MCTS algorithm can create a large number of objects that are particularly expensive to instantiate and each object is only needed for a short period of time; so this can have a huge impact on performance. Using Object Pool pattern it is possible to create a set of objects that may be reused. When a new object is needed, it is requested from the pool. If a previously prepared object is available it is returned immediately, avoiding the instantiation cost. If no objects are present in the pool, a new item is created and returned. When the object has been used and is no longer needed, it is returned to the pool, allowing it to be used again in the future without repeating the computationally expensive instantiation process. The same pool can also be used in different consecutive run of the algorithm leading to a huge benefit in terms of memory performance.

4.3.2 Node Recycling

This optimization is a memory enhancement that can bring significant performance benefits. Considering the structure of the core MCTS algorithm, performing more iterations causes a huge increase of memory used by the algorithm, due to the fact that MCTS usually add a new node on each iteration. So as the number of iterations — performed by the MCTS algorithm — increase, the

memory usage of the algorithm is bounded only by the size of the game tree.

Powley et al. [35] present a study of different memory bounding techniques for the MCTS algorithm. The technique called *Node Recycling* is used in the proposed solution. Node Recycling method aims to throw away the policy information, learned by MCTS, that is least relevant to the search preserving as much useful information as possible in the remaining tree.

The basic idea of this technique is to remove nodes coming from unpromising areas of the tree and recycle the freed memory to build more promising ones. This approach is due to the fact that the behaviour of MCTS algorithm is to repeatedly visit the most promising areas of the search tree and, at the same time, decrease the frequency of exploration of less promising areas. Thus it make sense to recycle these areas of the tree that have not recently been accessed, as they have a low priority for being exploited.

The overall idea of Node Recycling is to allocate a fixed pool of nodes of size determined by a memory budget, rather than creating a new node upon each expansion step of the algorithm. These nodes are used until the pool is exhausted, after that the recycling process begin. The node to be recycled is the leaf node whose statistics have least recently been accessed, in other words the node for which the UCB1 score has least recently been calculated.

The implementation of the Node Recycling technique is done trying to not significantly increase the amount of execution time taken by each iteration of the MCTS algorithm. In order to find the least recently accessed leaf node without scanning the whole tree, a queue structure is used. Nodes of the tree are managed using a least recently used (LRU) cache implemented as a first-in first-out (FIFO) queue. During a MCTS iteration, when a node is accessed it is removed from its current position in the queue and it is pushed to the back. When the memory budget is reached (the pool of nodes is empty), the node on top of the queue is recycled.

4.3.3 SP-MCTS UCB

This optimization is a bandit-based enhancement suggested by Schadd et al. [23] to improve the selection step of the core MCTS algorithm modifying the standard UCT with formula 3.2. A description of the method can be found in Section 3.6.

4.3.4 UCB1-Tuned

This optimization is a bandit-based enhancement suggested by Auer et al. [36] to tune more finely the bounds of UCB1. This approach uses formula 4.2 as upper confidence bound for the variance of the arm j in a multiarmed bandit problem.

$$V_{\mathrm{j}}(s) = (\frac{1}{2}\sum_{\tau=1}^{s} X_{j,\tau}^2) - \overline{X}^2{}_{\mathrm{j,s}} + \sqrt{\frac{2\ln t}{s}} \tag{4.2}$$

This means that arm j, that has been played s times during the first t plays, has a variance that is at most the sample variance plus $\sqrt{\frac{2\ln t}{s}}$ [36]. Then, the upper confidence bound $\sqrt{\frac{2\ln n}{n_{\mathrm{j}}}}$ of UCB1 is replaced with formula 4.3.

$$\sqrt{\frac{\ln n}{n_{\mathrm{j}}} \min\left\{\frac{1}{4}, V_{\mathrm{j}}(n_{\mathrm{j}})\right\}} \tag{4.3}$$

4.3.5 Rapid Action Value Estimation (RAVE)

This optimization is an All-Moves-As-First enhancement that combine the standard UCT score for each node with an AMAF score [37]. Figure 4.2 shows the AMAF heuristics in action on a simple situation. UCT is used to select actions during selection step of the core MCTS algorithm, then the simulation step plays some action leading to a terminal state. The basic idea of AMAF heuristic is that UCT could have also selected the simulated moves during the selection step as alternatives. Since these moves were used during the simulation, their corresponding nodes in the tree have their reward/visit count updated by the AMAF algorithm. Nodes that receive the extra AMAF update during backpropagation are marked

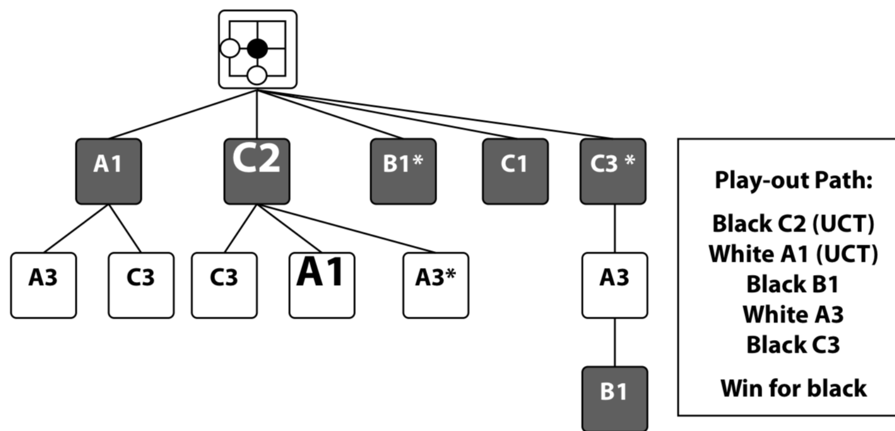


Figure 4.2: AMAF heuristic [37]

with an asterisk (*). In order to have both values, a separate count of rewards and visits for each type of update is maintained. So the total score of an action is expressed as

$$\alpha A + (1 - \alpha) U \tag{4.4}$$

where U represent the standard UCT score and A represent the AMAF score. The value of α used at each node decrease with each visit and it is computed, supplying a fixed positive integer $V > 0$, after n visit as [37]

$$\max \left\{ 0, \frac{V - v(n)}{V} \right\} \tag{4.5}$$

where parameter V represent the maximum value of visits a node can have in order to use the RAVE values to correct the UCT score; when a node is visited more times than the value expressed by V, RAVE values are not being used at all. With this approach exploited areas of the tree will use the accurate statistics more that unexplored areas of the tree.

4.3.6 Node Elimination

In addition to known enhancements we propose a new method to improve MCTS performance in domains with many early terminal states. One such domain is Sokoban, in which earlier deadlock detection can significantly reduce the search space. The idea behind *Node Elimination* is based on the observation that MCTS often repeatedly selected nodes that would lead to a deadlock inside the tree. This was caused by the constraints posed by the problem itself and we will therefore use Sokoban to illustrate the concept of Node Elimination.

In a Sokoban search, every rollout can end in either a solved state or a deadlock. With deadlocks being far more likely in non trivial levels, penalizing the reward of every rollout that terminated in a deadlock would not be beneficial to the search, since the hard part is not minimizing the cost to the solution but actually finding the solution. Furthermore, the length of the rollout does not give any insight on the quality of the rollout, since the move that originally caused a deadlock might have been executed far before the deadlock was finally detected. This lead to nodes close to the root being selected many times despite representing a deadlock state, wasting search resources. The solution we propose to this kind of situation is a recursive Node Elimination starting from terminal states.

During the backpropagation phase we remove from the tree all nodes in the current path that have no children and no untried moves. Terminal nodes are therefore directly eliminated (as they never have untried moves), effectively removing all deadlocks from the tree. When a node is eliminated it's also removed from its parent's list of children, therefore if every child of a node is a deadlock, that node is removed too. This goes on recursively, removing dead sub-trees that have already been fully explored. The elimination process is illustrated in figure 4.3. Node Elimination was also combined with Node Recycling by removing eliminated nodes from the node pool.

While in most two-player games, revisiting the same terminal nodes can be beneficial to the accuracy of the evaluation of the best move,

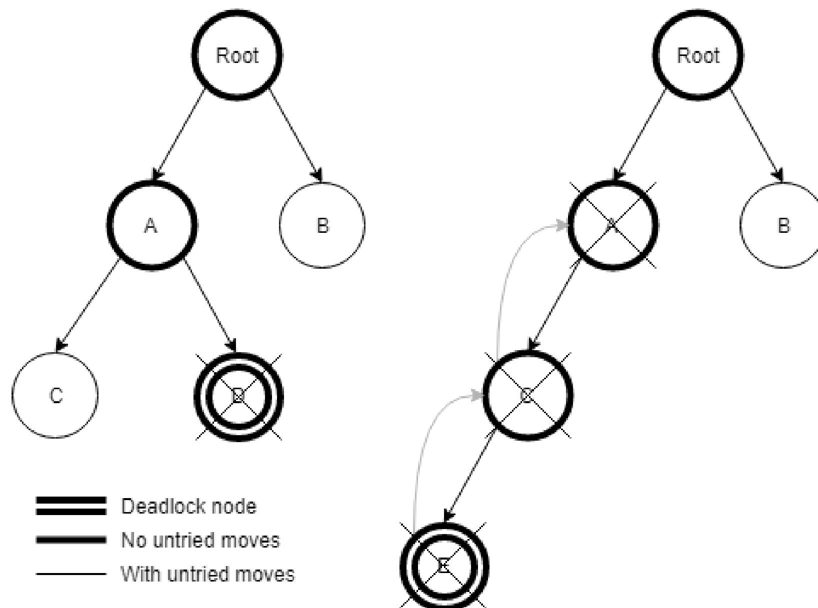


Figure 4.3: Node elimination:
Simple Node Elimination (left): only node D can be removed because node A still has children. Recursive Node Elimination (right): node C is expanded into terminal node E and — after the 0-length rollout — nodes E, C and A can be removed because once one is removed, its parent is left with no children and no untried moves.

in single player games the only required information is the top score achieved among all rollouts. Therefore, once a path has been fully explored it's no longer needed and can be ignored for the rest of the search. In the context of puzzle games, Node Elimination should provide the largest improvements in domains where terminal states can appear early in the search.

4.3.7 Cycles Avoidance

When working in domains where the game tree contains many cycles it's important to keep track of the visited states in the current path to ensure that no state is visited more than once, otherwise the search can enter loops and waste computational resources. In our SP-MCTS implementation we keep track of all visited states in the iteration, from the root to the end of the simulation. When

selecting a move for expanding a node or for performing the roll-out, we check if the resulting state has already been visited in the current path. If it is, we considered two options:

- *Stop On Cycle*: whether the cycle is encountered during the expansion phase or the simulation phase, the search treats the last state as terminal and immediately performs the pack-propagation phase.

- *Avoid Cycles*: the algorithm tries different moves among those available in the current state until either a valid non-visited state is obtained, or no more moves are available. If a valid state is obtained, the corresponding move is selected, otherwise the iteration is stopped.

4.4 Sokoban optimizations

In order to test the performance of the algorithm on Sokoban we had to implement some domain specific optimizations to reduce the search tree. These are some of the optimizations that were briefly described in Section 2.2.3.

4.4.1 Push Level Search

In Sokoban the pusher can move freely on the board but in reality the only relevant moves are those that change the position of a box. This means that instead of performing the search on states resulting from a single move, we perform it on an abstract layer in which each high-level move is actually a sequence of basic moves that ends with a box being pushed. In order to generate these high level moves, we need to perform local searches to identify the squares that the player can reach and from which he can push a box. These local searches are performed as breadth first searches on the low-level representation of the state. This solution can increase the branching factor if the player can reach many boxes from different directions, but it greatly reduces depth and minimizes the occurrence of cycles, reducing the overhead introduced by cycle avoidance.

4.4.2 Deadlock Detection

In addition to the abstract representation of the board, the first and fundamental optimization required is deadlock detection. We employed two different techniques to identify two types of deadlock: simple deadlocks and freeze deadlocks.

Simple deadlocks consist in a set of squares from which a stone can't reach any goal. They are computed only once, upon creation of the state object. To find them, a series of breadth first searches are performed: starting from each goal, the pusher is moved as if it was pulling a box instead of pushing it. It tries to pull in every direction into adjacent squares and recursively repeats the process for every new square encountered. During the search, all squares from which the pusher can pull a box are marked as visited, and represent all positions from which the box can reach that goal. After the search has been executed for each goal, the squares that are not marked as visited are those from which no box can reach a goal and will be kept in a hash table for the duration of the search. Whenever a box is pushed to one of those squares the state is flagged as a deadlock.

Freeze deadlock are identified during the search since they depend on the configuration of multiple boxes. A box is considered frozen if it can't move neither horizontally nor vertically. A box can't move horizontally if it has a wall on either side, or a frozen box. When a box is pushed to be adjacent to another box, we check if the box we just pushed is frozen, and in order to establish that we need to check if adjacent boxes are frozen. If any one of the chain of boxes is frozen, the state is a deadlock.

4.4.3 Tunnel Macros

Tunnel Macros are used to merge multiple pushes into a single move. They reduce tree depth and can help avoid deadlocks that would occur by pushing two boxes into the same tunnel. A tunnel can be defined as a portion of the level in which a line of squares is surrounded by walls. Every time available moves are generated, if the pattern of box and walls matches the one shown in Figure 4.4,

instead of a single push, a series of pushes is generated, equal to the number of squares required to place the box out of the tunnel. The only exception to this rule is if the tunnel contains a goal; in this case the box can be stopped on the goal (shown in figure 4.5).

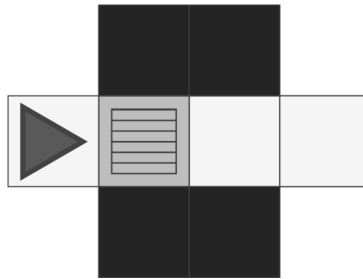


Figure 4.4: Tunnel Macro
Instead of R, the move RR is generated

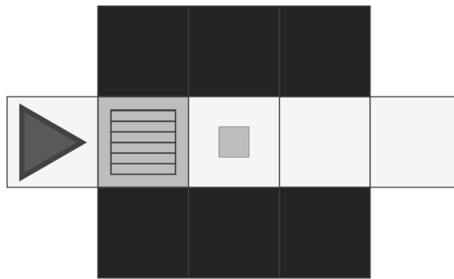


Figure 4.5: Tunnel Macro Exception
The move R is generated to push the box on the goal. The next time available moves are computed, move RR will be generated

4.4.4 Goal Macros

Goal Macros are used to identify the order with which goals are covered by boxes inside a goal room. A goal room is an area of the board with at least three adjacent boxes, which tries to maximize the number of squares and minimize the number of entrances and the number of boxes, according to the following formula

$$GRoomScore = 1000 * (20 - en) + 5 * sq - 100 * man_io - 500 * boxes \tag{4.6}$$

where *en* is the number of entrances, *sq* is the number of squares, *man_io* are entrances from which the pusher can enter the goal room but he cannot push a box through and *boxes* is the number of boxes inside the goal room. If a suitable goal room is found, the next step is to build the Goal Macro Tree. The Goal Macro Tree contains for each node a hash that represents the boxes on the goal room's goals, a sequence of pairs entrance-goal that represent the goal that the box should be pushed to if it was on that entrance and a sequence of moves that can be used to execute the macro. During moves generation, if a box is about to be pushed to an entrance square, the tree is accessed to retrieve the move sequence. If the move sequence is valid in the current state, it's substituted to the single push. If the sequence is not valid, an online search is executed by considering a sub-problem in which every other box is turned into a wall and the only valid goal is the one that should be filled by the macro. If the search is successful, the sequence is used as a macro, otherwise the single push is used.

4.4.5 Goal Cuts

Goal Cuts are a simple improvement over Goal Macros for which instead of adding the Goal Macro to the list of available moves in the current states, it is returned as the only move available.

Chapter 5

Experimental results

In this chapter we compare the performance of IDA* and various versions of MCTS on Samegame and Sokoban.

5.1 Samegame

As a preliminary part of our study, we replicated the experiment performed by Schadd et al. [23] and compared our results. We ran an experiment using our implementation of SP-MCTS. Due to differences of the Samegame implementation, our program was considerably slower than the one used in [23]. Our implementation performed an average of 2630 iterations per second against the 13888 iterations per second obtained by [23]. The program used in [23] performed 100000 iterations and 1000 restarts with search time distributed per move, which — considering an average solution length of 64.4 — resulted in an average of 1553 iterations per move. We decided to keep execution time as a constraint instead of number of iterations, and to achieve that, we reduced the number of randomized restarts from 1000 to 180. The resulting MCTS configuration used a UCT constant of 4.31, a SP-MCTS constant of 96.67, 1500 iterations per move and 180 restarts. As a result, we obtained a score of 73586 in the standard test set [21]. Table 5.1 shows the results for each level in the test set in comparison with the results obtained by [23] and HGSTS [22], the top scoring documented algorithm for Samegame.

#	Our SP-MCTS	SP-MCTS(3)	HGSTS
1	2671	2919	2561
2	3723	3797	4995
3	3051	3243	2858
4	3781	3687	4051
5	4001	4067	4633
6	4189	4269	5003
7	2359	2949	2717
8	3881	4043	4622
9	4723	4769	6086
10	2623	3245	3628
11	2689	3259	2796
12	3083	3245	3710
13	2881	3211	3271
14	2687	2937	2432
15	3021	3343	3877
16	4915	5117	6074
17	4717	4959	5166
18	5109	5151	6044
19	4843	4803	5019
20	4639	4999	5175
Total	73586	78012	84718

Table 5.1: Results of our implementation of SP-MCTS

Our results are overall worse than the ones obtained by the original authors, but considering the difference in the number of restarts, such outcome is in line with what we expected. This experiment allowed us to ensure that our implementation is coherent to [23] and to proceed with further experiments with a baseline for performance comparisons.

5.1.1 Node Elimination & Cycles Avoidance

We started our optimizations analysis with Node Elimination. To reduce computation time we performed the tests with 1500 iterations without randomized restarts. The baseline was obtained

with SP-MCTS and no other optimization. We then enabled Node Elimination and compared the results, shown in Table 5.2. Node

Configuration	Score
Baseline	57965
Node Elimination	56940

Table 5.2: Node Elimination outcome

Elimination obtained a score lower than the baseline. We can assume this is due to the fact that in Samegame revisiting terminal nodes can help in the estimation of the action value function.

Since in Samegame cycles can't occur, there was no need to test Cycles Avoidance.

5.1.2 Parameters tuning

In order to take full advantage of the optimizations we described in Section 4, we needed to tune their parameters to obtain the best configuration. In particular, we executed repeated searches with different values for the RAVE threshold and the memory budget of Node Recycling. We also tuned again the UCT constant with UCB1-Tuned, since [23] performed the tuning with the SP-MCTS formula. Node Elimination has no parameters so it didn't require tuning. As before we performed the tests with 1500 iteration and no randomized restarts. The results of the tuning process are shown in Tables 5.3 and 5.4.

If we only consider the results with RAVE enabled, it reached the highest score with $V = 100$, but contrary to our expectations, it had an overall negative impact on the score. We concluded that the assumption upon which RAVE is based, does not hold for Samegame, i.e. moves performed later on in a simulation can't be considered as if they were performed at the beginning.

Node Recycling reduced the score too, although less than RAVE, which means that it may prune good sub-trees in Samegame. With a memory budget of 600 nodes it produced a score close to the baseline.

V	Score
1	57965
5	46663
10	47722
15	49519
25	49239
50	52028
100	54738

Table 5.3: Scores obtained with different RAVE thresholds. The first row is equivalent to disabling RAVE and represents our baseline.

Nodes	Score
300	55302
600	57798
900	54601
1200	56532
1500	57965

Table 5.4: Scores obtained with Node Recycling and different memory budgets. The last row is equivalent to disabling Node Recycling and represents our baseline.

As shown in Figure 5.1, UCB1-Tuned performed better than SP-MCTS with different values but reached the highest score with a constant value of 10.

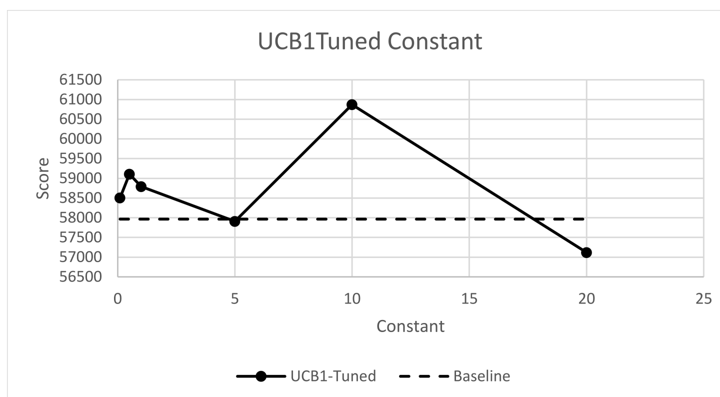


Figure 5.1: Scores obtained with different constant values

5.1.3 Results

Considering the results obtained in previous experiments, we performed a comparison between our basic version of SP-MCTS and SP-MCTS with UCB1-Tuned. For this evaluation we used 1500 iterations and 180 restarts. The results of this comparison can be seen in Table 5.5. UCB1-Tuned obtained a score of 74136, surpassing our previous score by 550 points.

#	Our SP-MCTS	UCB1-Tuned
1	2671	2615
2	3723	3763
3	3051	3251
4	3781	3607
5	4001	3891
6	4189	4147
7	2359	2675
8	3881	3875
9	4723	4689
10	2623	2601
11	2689	2683
12	3083	3125
13	2881	2805
14	2687	2687
15	3021	3329
16	4915	4891
17	4717	4639
18	5109	5117
19	4843	4869
20	4639	4877
Total	73586	74136

Table 5.5: Comparison between results obtained by our SP-MCTS implementation and UCB1-Tuned enhancement.

Finally we performed an experiment to compare our optimized SP-MCTS against our version of the IDA* algorithm. [23] proposed an upper bound on the score for Samegame by considering all blocks

of the same color as adjacent. The bonus of 1000 points for clearing the board is also awarded unless there exists a color of which a single block is remaining. We ran an experiment with IDA* using the same heuristic and a maximum of 7500000 nodes, giving approximately two hours of search time for each level. The results of this experiment can be seen in Table 5.6 compared with the best results obtained by MCTS. MCTS strongly outperformed IDA*, which with 9882 points performed at the level of a human beginner.

#	Optimized MCTS	IDA*
1	2615	415
2	3763	387
3	3251	378
4	3607	670
5	3891	592
6	4147	655
7	2675	435
8	3875	505
9	4689	518
10	2601	572
11	2683	295
12	3125	543
13	2805	643
14	2687	177
15	3329	545
16	4891	321
17	4639	457
18	5117	573
19	4869	541
20	4877	660
Total	73586	9882

*Table 5.6: Results obtained by comparing Optimized MCTS algorithm with IDA**

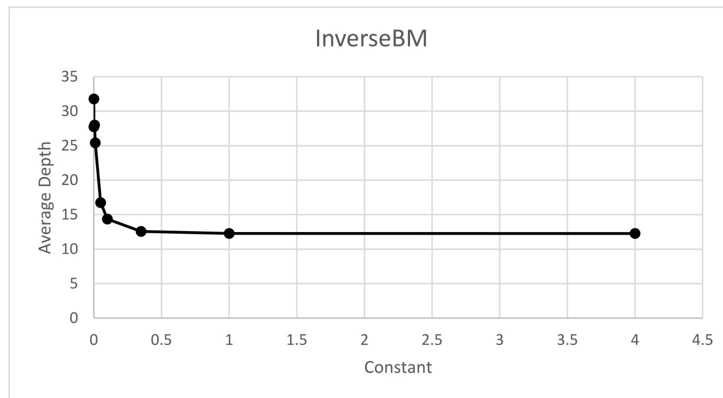
5.2 Sokoban

Contrary to Samegame, where the reward function was defined as a part of the game rules, in Sokoban we had to define a reward function. In order to determine which among those defined in Section4.2 performed better, we compared them using different UCT constant values. We used the 155 levels of the Microban [5] suite as a test set throughout all experiments.

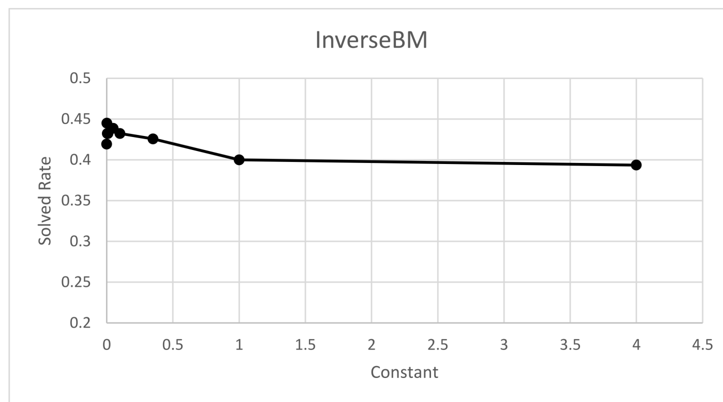
5.2.1 Reward types

All tests were performed with random rollouts, Avoid Cycles and Node Elimination enabled. Furthermore, the search was stopped as soon as a solution was found. Figures 5.2 through 5.5 show the impact of the constant on the depth of the generated tree and on the portion of solved levels for InverseBM, NegativeBM, R0 and Boxes respectively.

The optimal value for rewards has been proven to be $C = 1$ for rewards in the range $[0, 1]$ [2]. Despite having a range between 0 and 1, the best scores for InverseBM were obtained with constants in the range $0.001 - 0.05$. The tree depth graph shows that with values higher than 0.35 the average depth among all levels remains almost constant. This can be interpreted as the algorithm leaning towards exploration for those values. This unusual behavior may be due to the distribution of the reward function values: while in constant-sum games in the range $[0, 1]$ the reward distribution is typically centered on $\frac{1}{2}$ with 0 and 1 having the same probability distribution, the InverseBM reward distribution varies according to the level and generally leans towards 0. This can cause the 0.35 constant value to assign an higher weight to the exploration part of the UCT formula.

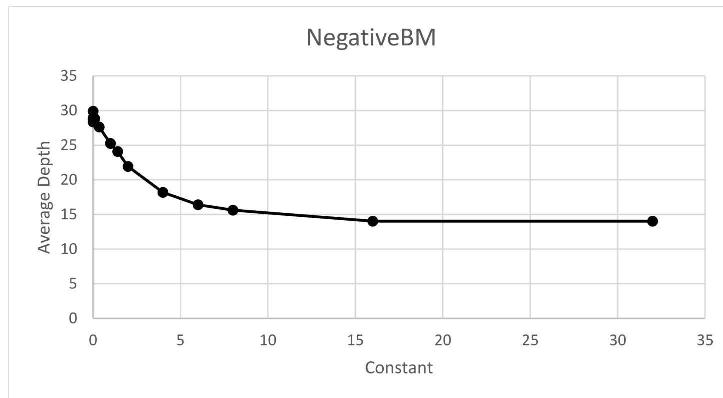


(a) Trend of generated tree depth using different constant values for reward InverseBM

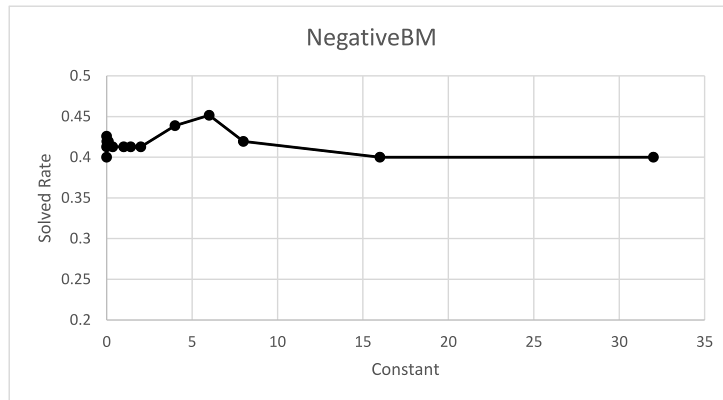


(b) Trend of solved levels rate using different constant values for reward InverseBM

Figure 5.2



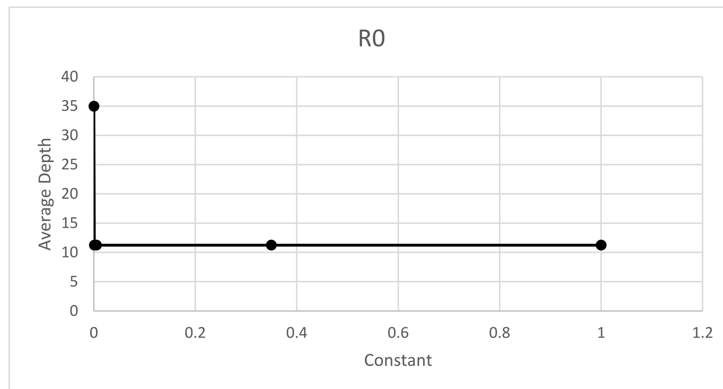
(a) Trend of generated tree depth using different constant values for reward NegativeBM



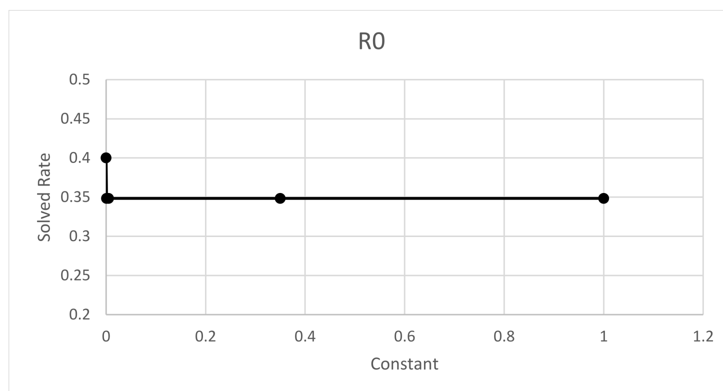
(b) Trend of solved levels rate using different constant values for reward NegativeBM

Figure 5.3

NegativeBM reaches its maximum score with a constant value of 6 and the generated tree depth decreases more gradually due to the larger reward range.



(a) Trend of generated tree depth using different constant values for reward R0

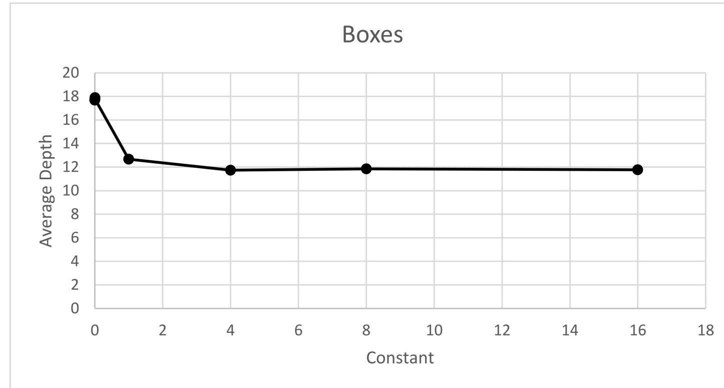


(b) Trend of solved levels rate using different constant values for reward R0

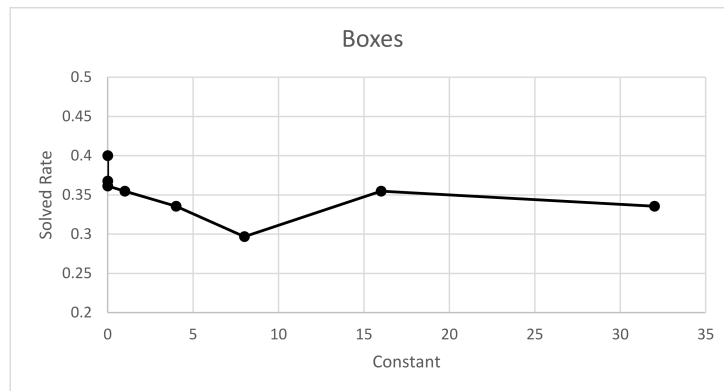
Figure 5.4

R0 charts clearly show how this reward type is not suitable for solving Sokoban using MCTS. The depth of the trees confirms what we suspected: with the reward being 0 for every rollout except for the last one (since the search is stopped at the first solution), the only relevant part of the UCT formula is the exploration component, which means that for every value of the constant greater than 0, the algorithm will perform a pure exploration. In our implementation if two or more nodes have the same UCT value, the selection phase is deterministic and always chooses the node that was generated first. This implies that with the constant equal to 0, all nodes have a UCT value of 0 and the algorithm will perform a pure

exploitation.



(a) Trend of generated tree depth using different constant values for reward Boxes



(b) Trend of solved levels rate using different constant values for reward Boxes

Figure 5.5

The results using Boxes reward do not seem to follow a specific trend, and with the top score obtained with the constant equal to 0, this too does not appear to be suitable for solving Sokoban with MCTS.

Considering these results, we selected NegativeBM as a reward for the following experiments.

We also performed the same experiments using SP-MCTS and UCB1-Tuned to determine the best reward and constant configuration for each. The results are shown in Figures 5.6 through 5.9. SP-MCTS

reached its peak performance with NegativeBM and a constant of 2, while UCB1-Tuned obtained the best result with InverseBM and a constant of 0.05.

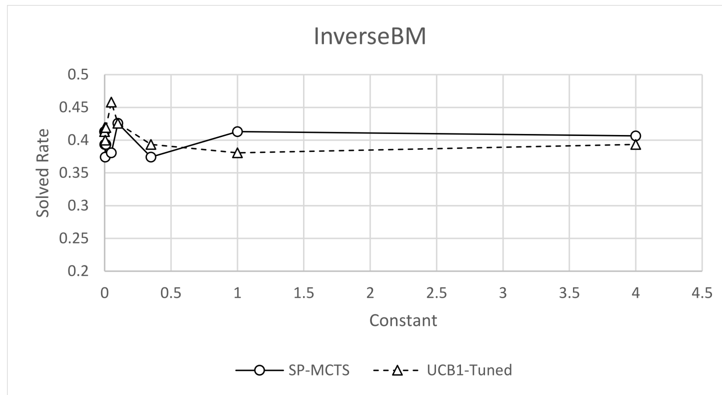


Figure 5.6: SP-MCTS and UCB1-Tuned results with InverseBM

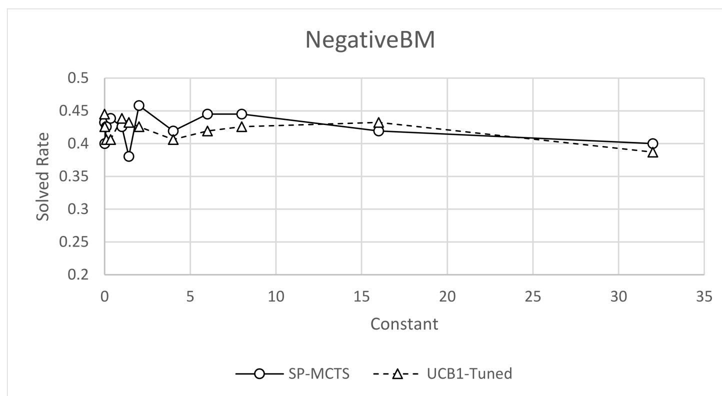


Figure 5.7: SP-MCTS and UCB1-Tuned results with NegativeBM

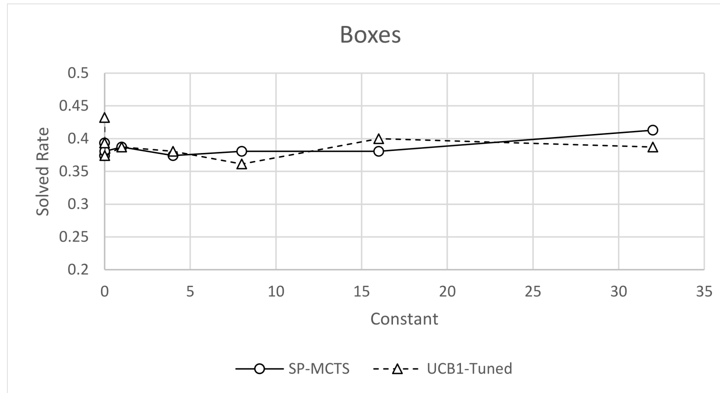


Figure 5.8: SP-MCTS and UCB1-Tuned results with Boxes

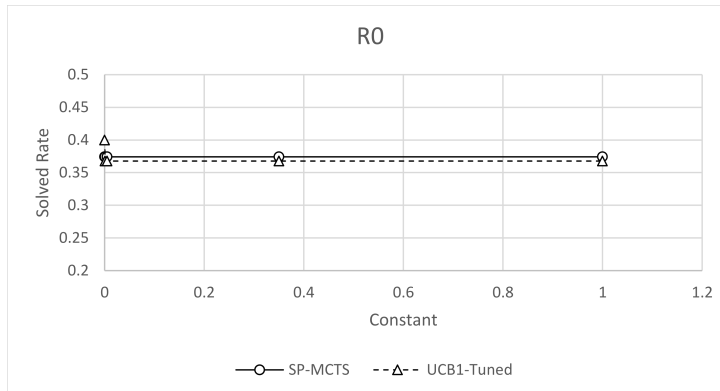


Figure 5.9: SP-MCTS and UCB1-Tuned results with R0

5.2.2 Node Elimination & Cycles Avoidance

In the following experiment we evaluated the effects of the Node Elimination and Cycles Avoidance optimizations. The baseline was obtained with standard UCT, random simulations, Node Elimination disabled and Cycles Avoidance in Stop On Cycle mode (i.e. the iteration is stopped if a cycle is detected).

Figure 5.10 shows the number of nodes added to the search tree, the number of nodes deleted by Node Elimination and the number of iterations in which the expansion phase was skipped because the last selected node was terminal (nodes skipped). *BASE* represents

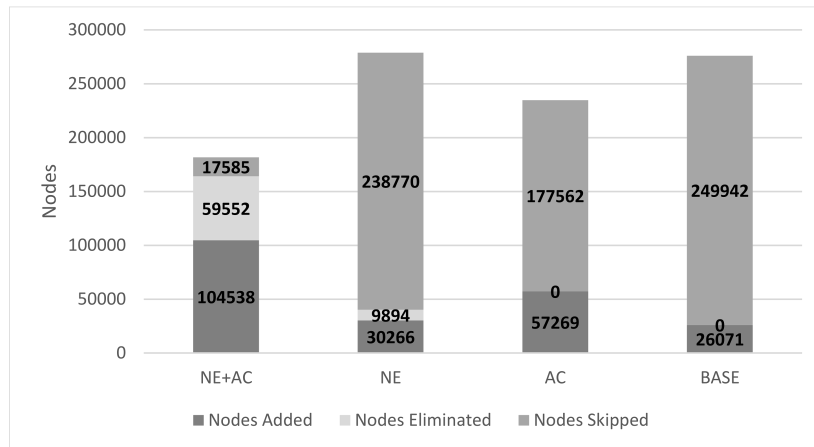


Figure 5.10: Nodes added, removed or skipped with combinations of Node Elimination (NE) and Avoid Cycles (AC)

the baseline. We can see in Table 5.7 how in the baseline the great majority of the iterations were executed without expanding the tree, thus terminating either on a cycle or on terminal nodes. Node Elimination prevented the algorithm from revisiting terminal nodes, causing it to slightly increase the number of nodes added to the tree, but its impact on the number of solved levels was not positive. The Avoid Cycles mode of Cycle Avoidance lead to a larger increase in the number of expanded nodes, when compared to Node Elimination. The increase in number of solved levels also lead to a decrease in the total number of iterations (as the search is stopped as soon as a solution is found). The combination of Node Elimination and Avoid Cycles obtained the best results, with far more nodes added to the tree and the highest number of solved levels.

Figure 5.11 illustrates the performance of the different configurations with various number of iterations.

Configuration	Added	Eliminated	Skipped	Solved
Baseline	9%	0%	91%	20
Node Elimination	11%	3%	86%	18
Avoid Cycles	24%	0%	76%	44
NE + AC	57%	33%	10%	76

Table 5.7: Percentages of nodes added, eliminated and skipped, along with the number of solved levels for each configuration. The last row represents the combination of Node Elimination and Avoid Cycles

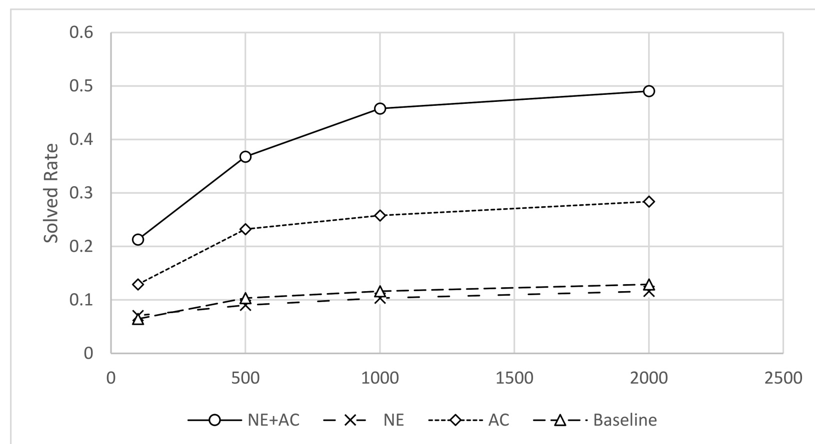


Figure 5.11: Solved levels with combinations of Node Elimination (NE) and Avoid Cycles (AC) and different number of iterations

5.2.3 Parameters tuning

Then we tested all MCTS optimizations separately, as we done for Samegame, in order to determine the effect of each one on Sokoban and, after that, we combined all the enhancement together, trying to obtain the best result. Again we proceeded with the tuning of the parameters. In particular, we executed repeated searches with different values for the RAVE threshold and the memory budget of Node Recycling. Other enhancements such as UCB1Tuned and Node Elimination have no parameters, hence they do not require tuning.

Table 5.8 shows variation in term of performance on the test set as the parameter V change; this parameter represents the maximum number of visits a node needs to have before the RAVE values are not used at all.

V	Solved
1	45.8 %
5	41.9 %
10	42.6 %
15	41.9 %
25	42.6 %
50	43.8 %
100	42.6 %

Table 5.8: Levels solved with different RAVE thresholds. The first row is equivalent to disabling RAVE and represents our baseline.

With the exception of $V = 1$, which is equivalent to disabling RAVE, we can see that RAVE solved the highest number of levels with $V = 50$. However the performance was still lower than the baseline. As we did for Samegame, we concluded that the assumption necessary for RAVE to perform well does not hold for Sokoban.

Table 5.9 shows instead performance trend on the test set changing the parameter responsible for the maximum number of nodes kept in memory by the Node Recycling optimization; this parameter is

used as memory budget parameter in order to keep in memory only the most promising nodes and recycling the less promising ones. We

Nodes	Solved
200	43.2 %
400	43.2 %
600	43.2 %
800	43.2 %
1000	45.8 %

Table 5.9: Levels solved with Node Recycling and different memory budgets. The last row is equivalent to disabling Node Recycling and represents our baseline.

can see how independently of the memory budget, Node Recycling performed uniformly worse than the baseline in Sokoban.

5.2.4 Simulation policy

Next we tested different options for the simulation policy: we performed searches using random simulations, ϵ-greedy and ϵ-IDA* with different combinations of IDA* nodes and MCTS iterations. Since the IDA* simulations were significantly heavier than random and ϵ-greedy ones, we compared the policies on similar execution times. The value of ϵ was set to 0.2 for both ϵ-based policies. Table 5.10 shows the results of the various ϵ-IDA* configurations. Table 5.11 shows the comparison between ϵ-IDA*, ϵ-greedy and random simulation policies. ϵ-greedy solved 95 levels

IDA* Nodes	MCTS Iterations	Time	Solved
100	100	1h 13m 21s	55.5%
20	500	1h 28m 30s	54.2%
5	1000	1h 0m 50s	54.2%

Table 5.10: Results of ϵ-IDA. Each row represents one configuration.*

Policy	Iterations	Time	Solved
ϵ-IDA*(100)	100	1h 13m 21s	55.5%
ϵ-greedy	5000	1h 9m 47s	61.3%
Random	5000	1h 11m 55s	52.3%

Table 5.11: Results of the different simulation policies.

5.2.5 Sokoban complexity

During our experiments we noticed that contrary to what we expected, the performance of MCTS didn't improve much when increasing the number of iterations. When comparing the number of levels solved relative to the number of nodes used in IDA* and the number of iterations performed in MCTS, we noticed a similar trend (Figures 5.12 and 5.13). After reaching a certain threshold, the resources needed to solve additional levels increase dramatically. We consider this to be a consequence of the quality of the heuristic evaluation which is not suitable for levels in which boxes must be pushed away from goals in order to find a solution. It's still not clear why the threshold in MCTS appears earlier than in IDA*.

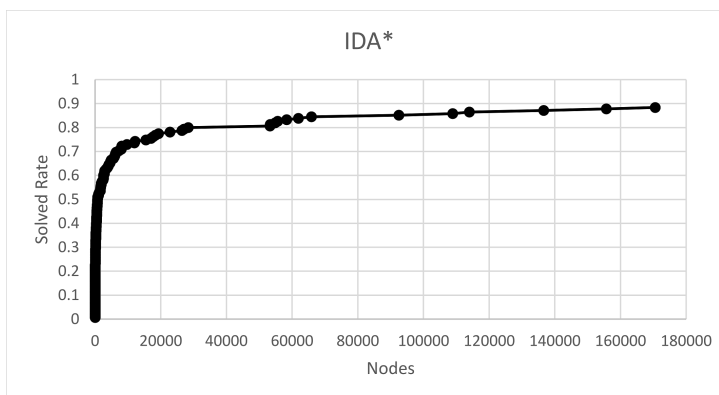


*Figure 5.12: Levels solved according to number of nodes used in IDA**

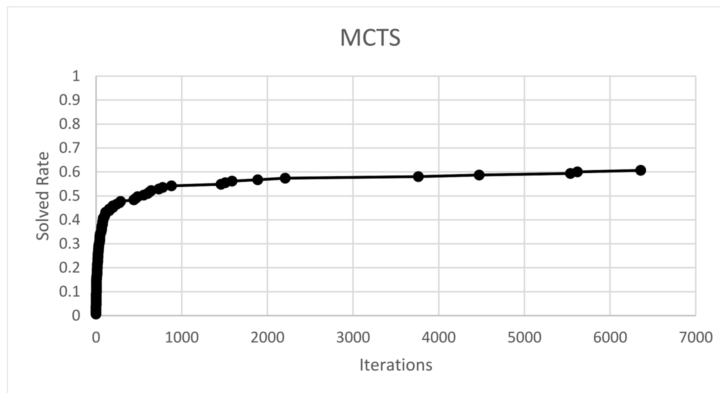


Figure 5.13: Levels solved according to number of iterations performed in MCTS

5.2.6 Results

The last experiment we performed consisted in the comparison between IDA* and different versions of MCTS for Sokoban. For this final comparison we used three versions of MCTS based on the formula used for the estimation of the action-value function: standard UCT, UCB1-Tuned and SP-MCTS. Standard MCTS used NegativeBM as a reward with a constant of 6. UCB1-Tuned used InverseBM as a reward with a constant of 0.05. SP-MCTS used NegativeBM as a reward, with a constant of 2. Node Elimination and Cycles Avoidance were enabled in all three configurations. To keep a comparable search time between all methods, IDA* used a maximum of 200000 nodes and all MCTS methods used 10000 iterations with ϵ-greedy as a simulation policy, with $\epsilon = 0.2$. Table 5.12 shows the final results of this comparison.

Method	Solved
IDA*	88.4%
UCT	64.5%
UCB1-Tuned	54.8%
SP-MCTS	60.6%

Table 5.12: Methods comparison

IDA* solved the highest number of levels with respect to the other methods. Among MCTS methods, UCT achieved the best performance with 64.5% of solved levels.

Chapter 6

Conclusions

The purpose of this thesis was to measure the performance of MCTS in Sokoban. We developed various optimizations for MCTS and verified their effectiveness on Sokoban and Samegame. RAVE obtained poor results in both domains and we concluded that neither Sokoban nor Samegame fit the AMAF criteria, according to which all moves performed during a simulation can be considered as if they were played at the beginning. UCB1-Tuned obtained good results on Samegame, improving the previous score obtained with SP-MCTS. Node Recycling obtained poor results in both games in terms of score and number of solved levels, although its major contribution concerns memory usage. SP-MCTS performed better than UCB1-Tuned but worse than UCT on Sokoban. Node Elimination and Cycles Avoidance provided a considerable improvement in Sokoban, especially when used together, since they were able to greatly increase the number of nodes added to the tree and, as a consequence, the number of solved levels. In Samegame, Node Elimination obtained poor performance, mainly because the aim was to maximize the score instead of finding a single solution, meaning that revisiting terminal nodes could be beneficial to the accuracy of the action-value function estimate. As a comparison, IDA* performed very poorly in Samegame, while in Sokoban it obtained the best results. A brief analysis of Sokoban and the chosen heuristic rewards led us to believe that the poor performance obtained by MCTS may be caused by the fact that, while typically MCTS is used on games where at the end of the simulation the outcome is

known, in Sokoban at the end of a simulation we needed to provide an evaluation of the distance from the solution. Since computing this evaluation would be as difficult as solving the level, we could only provide a heuristic estimate. IDA*, on the other hand, is designed to use heuristics to guide the search and prune the tree. According to the results obtained with the tested configurations, MCTS doesn't appear to be successful in Sokoban.

6.1 Future research

During our experiments on Sokoban we noticed that the Microban suite was split between very easy levels, which could be solved with very few iterations and very hard levels, which couldn't be solved within reasonable time. As shown by Figures 5.12 and 5.13 this was true for both MCTS and IDA*, which means that it was a characteristic of the levels and not of the specific algorithm. A more precise analysis on the effects of the optimizations could be performed by building a test set with levels of gradually increasing difficulty, in order to obtain more accurate indications on the effect of each parameter and optimization.

On the subject of improving MCTS performance on Sokoban, better results could be obtained with more complex reward functions, which make use of domain knowledge to improve the accuracy of the estimated action values.

One final interesting research topic can be the application of the newly proposed Node Elimination and Cycles Avoidance in other domains. Cycles Avoidance in the Avoid Cycles variant should be effective in all games where cycles frequently occur, while our results suggest that Node Elimination could provide benefits in those domains where terminal nodes appear early in the game tree and the goal is not to maximize a score, but to find a single solution.

Bibliography

[1] R. Coulom, "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search," in *5th International Conference on Computer and Games* (P. Ciancarini and H. J. van den Herik, eds.), (Turin, Italy), May 2006.

[2] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *Proceedings of the 17th European Conference on Machine Learning*, ECML'06, (Berlin, Heidelberg), pp. 282–293, Springer-Verlag, 2006.

[3] A. Junghanns and M. Eltern, "Pushing the limits: New developments in single-agent search," tech. rep., 1999.

[4] G. Kendall, A. J. Parkes, and K. Spoerer, "A survey of np-complete puzzles.," *ICGA Journal*, vol. 31, no. 1, pp. 13–34, 2008.

[5] D. W. Skinner, "Microban." `http://www.abelmartin.com/rj/sokobanJS/Skinner/David\%20W.\%20Skinner\%20-\%20Sokoban_files/Microban.txt`.

[6] J. C. Culberson, "Sokoban is pspace-complete," 1998.

[7] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," *Artif. Intell.*, vol. 27, pp. 97–109, Sept. 1985.

[8] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, pp. 100–107, July 1968.

[9] R. E. Korf and L. A. Taylor, "Finding optimal solutions to the twenty-four puzzle," in *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, AAAI 96, IAAI 96, Portland, Oregon, August 4-8, 1996, Volume 2.* (W. J. Clancey and D. S. Weld, eds.), pp. 1202–1207, AAAI Press / The MIT Press, 1996.

[10] A. Reinefeld and T. A. Marsland, "Enhanced iterative-deepening search," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 16, no. 7, pp. 701–710, 1994.

[11] J. C. Culberson and J. Schaeffer, "Searching with pattern databases," in *Advances in Artifical Intelligence* (G. McCalla, ed.), (Berlin, Heidelberg), pp. 402–416, Springer Berlin Heidelberg, 1996.

[12] "Pi-corral pruning - sokoban wiki." `http://sokobano.de/wiki/index.php?title=Sokoban_solver_"scribbles"_by_Brian_Damgaard_about_the_YASS_solver\#PI-Corral_pruning`.

[13] B. Bouzy, "Associating shallow and selective global tree search with monte carlo for 9 × 9 go," in *Computers and Games* (H. J. van den Herik, Y. Björnsson, and N. S. Netanyahu, eds.), (Berlin, Heidelberg), pp. 67–80, Springer Berlin Heidelberg, 2006.

[14] H. Juille, P. Kenneth, and A. D. Jong, "Methods for statistical inference: Extending the evolutionary computation paradigm," 1999.

[15] H. S. Chang, M. C. Fu, J. Hu, and S. I. Marcus, "An adaptive sampling algorithm for solving markov decision processes," *Operations Research*, vol. 53, no. 1, pp. 126–139, 2005.

[16] A. Bai, F. Wu, and X. Chen, "Online planning for large markov decision processes with hierarchical decomposition," *ACM Trans. Intell. Syst. Technol.*, vol. 6, pp. 45:1–45:28, July 2015.

[17] C.-S. Lee, M. Müller, and O. Teytaud, "Special issue on monte carlo techniques and computer go," vol. 2, pp. 225 – 228, 01 2011.

[18] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, Jan. 2016.

[19] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of go without human knowledge," *Nature*, vol. 550, pp. 354–, Oct. 2017.

[20] B. Arneson, R. B. Hayward, and P. Henderson, "Monte carlo tree search in hex.," *IEEE Trans. Comput. Intellig. and AI in Games*, vol. 2, no. 4, pp. 251–258, 2010.

[21] "Samegame highscores - js-games.de." `http://www.js-games.de/eng/highscores/samegame/lx`.

[22] S. Edelkamp, P. Kissmann, D. Sulewski, and H. Messerschmidt, "Finding the needle in the haystack with heuristically guided swarm tree search," (Göttingen), pp. 235–255, February23–25 2010.

[23] 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," *Knowl.-Based Syst.*, vol. 34, pp. 3–11, 2012.

[24] C. Rosin, "Nested rollout policy adaptation for monte carlo tree search," 2011.

[25] T. Cazenave, "Nested Monte-Carlo Search," in *Proc. 21st Int. Joint Conf. Artif. Intell.*, (Pasadena, California), pp. 456–461, 2009.

[26] A. Guez, T. Weber, I. Antonoglou, K. Simonyan, O. Vinyals, D. Wierstra, R. Munos, and D. Silver, "Learning to search with mctsnets," *CoRR*, vol. abs/1802.04697, 2018.

[27] G. Chaslot, M. Winands, H. Herik, J. Uiterwijk, and B. Bouzy, "Progressive strategies for monte-carlo tree search," vol. 04, pp. 343–357, 11 2008.

[28] F. C. Schadd, "Monte-carlo search techniques in the modern board game thurn and taxis," master thesis, Maastricht University, Maastricht, Netherlands, 12/2009 2009.

[29] L. Kocsis, C. Szepesvári, and J. Willemson, "Improved monte-carlo search," 2006.

[30] R. Rubinstein, "The cross-entropy method for combinatorial and continuous optimization," *Methodology And Computing In Applied Probability*, vol. 1, pp. 127–190, Sep 1999.

[31] C. Browne, E. J. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. P. Liebana, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods.," *IEEE Trans. Comput. Intellig. and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012.

[32] G. M. J. B. Chaslot, M. H. M. Winands, and H. J. van den Herik, "Parallel monte-carlo tree search," in *Computers and Games* (H. J. van den Herik, X. Xu, Z. Ma, and M. H. M. Winands, eds.), (Berlin, Heidelberg), pp. 60–71, Springer Berlin Heidelberg, 2008.

[33] D. M. Breuker, "Memory versus search in games," phd thesis, Maastricht University, Maastricht, Netherlands, 1998.

[34] T. Weber, S. Racanière, D. P. Reichert, L. Buesing, A. Guez, D. J. Rezende, A. P. Badia, O. Vinyals, N. Heess, Y. Li, R. Pascanu, P. Battaglia, D. Silver, and D. Wierstra, "Imagination-augmented agents for deep reinforcement learning," *CoRR*, vol. abs/1707.06203, 2017.

[35] E. Powley, P. Cowling, and D. Whitehouse, "Memory bounded monte carlo tree search," 2017.

[36] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine Learning*, vol. 47, pp. 235–256, May 2002.

[37] D. P. Helmbold and A. Parker-Wood, "All-moves-as-first heuristics in monte-carlo go.," in *IC-AI* (H. R. Arabnia, D. de la Fuente, and J. A. Olivas, eds.), pp. 605–610, CSREA Press, 2009.