Norwegian University of
Science and Technology

Improving sliding-block puzzle solving using meta-level reasoning

Ruben Grønning Spaans

Master of Science in Computer Science

Submission date:	June 2010
Supervisor:	Agnar Aamodt, IDI
Co-supervisor:	Tor Gunnar Høst Houeland, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

A sliding-block puzzle is a kind of single-player puzzle. Instances of this domain can be very different, and may require different algorithms from different fields. No efficient method of solving arbitrary instances is known. In the specialization project, the domain of sliding-block puzzles was investigated, and a program was developed that can solve many instances of such puzzles. This program offers a multitude of parameters that can be set by the user. Some puzzles require specific settings in order to be solved by the program.

This master thesis project will study how the domain of sliding-block puzzles can benefit from using meta-level reasoning, and how these techniques can be of benefit in solving such puzzles. Based on the study, one or more methods within meta-level reasoning will be chosen, and the existing program from the specialization project will be enhanced using these methods. This includes enhancing the underlying solver program from the specialization project. Finally, the enhanced program will be tested. An evaluation will be performed based on a comparison of the results from the enhanced and the original program.

Assignment given: 15. January 2010
Supervisor: Agnar Aamodt, IDI

Acknowledgements

This master thesis was carried out within the Division of Intelligent Systems (DIS) at the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU) during the Spring semester of 2010.

I would like to thank my supervisors Agnar Aamodt and Tor Gunnar Houeland for their valuable feedback and suggestions, and for constantly pushing me to work on this thesis.

Thanks to the domain of sliding-block puzzles for being an extremely interesting challenge to work on.

Trondheim, 17th June 2010

Ruben Spaans

Abstract

In this thesis, we develop a meta-reasoning system based on CBR which solves sliding-block puzzles. The meta-reasoning system is built on top of a search-based sliding-block puzzle solving program which was developed as part of the specialization project at NTNU. As part of the thesis work, we study existing literature on automatic puzzle solving methods and state space search, as well as the use of reasoning and meta-level reasoning applied to puzzles and games. The literature study forms the theoretical foundation for the development of the meta-reasoning system. The meta-reasoning system is further enhanced by adding a meta-control cycle which uses randomized search to generate new cases to apply to puzzles. In addition, we explore several ways of improving the underlying solver program by trying to solve hard puzzles by using the solution for easier variants, and by developing a more memory-efficient way of representing puzzle configurations.

We evaluate the results of our system, and shows that it offers a slight improvement compared to solving the puzzles with a set of general cases, as well as showing vast improvement for a few isolated test cases, but the performance is slightly behind the hand-tuned parameters we found in the specialization project.

We conclude our work by identifying parts of our system where improvement can be done, as well as suggesting other promising areas for further research.

Contents

1 Introduction **13**
1.1 Goal . . . 13
1.1.1 Our previous work . . . 13
1.2 Background and motivation . . . 14
1.2.1 Specialization project . . . 14
1.3 Literature study . . . 15
1.3.1 Goal of the literature study . . . 16
1.3.2 Method . . . 16
1.4 Structure of this thesis . . . 16
1.5 Credits . . . 17

2 Classical algorithms **19**
2.1 Uninformed search algorithms . . . 19
2.1.1 Random walk . . . 19
2.1.2 Depth-first search . . . 20
2.1.3 Breadth-first search . . . 20
2.1.4 Bidirectional search . . . 20
2.1.5 Iterative deepening . . . 21
2.2 Informed search algorithms . . . 21
2.2.1 A* . . . 21
2.2.2 IDA* . . . 22
2.3 Domain-independent enhancements . . . 22
2.3.1 Pattern databases . . . 22
2.3.2 Macro moves . . . 23
2.3.3 Transposition tables . . . 23
2.3.4 Endgame databases . . . 23
2.4 Methods based on expert knowledge . . . 23
2.4.1 Rule-based reasoning . . . 24
2.4.2 Case-based reasoning . . . 24
2.4.3 Model-based reasoning . . . 25
2.4.4 Hybrid design . . . 25
2.5 Meta-level reasoning . . . 25

3 Sliding-block puzzles defined **27**
3.1 Definition of the problem domain 27
3.1.1 15-puzzle, $(n \times m - 1)$-puzzle 29
3.1.2 Rush Hour 30
3.2 Definition of our meta-level domain 31
3.2.1 Properties of the meta-level domain (parameter space) 32
3.2.2 Puzzle features 36
3.3 Our test suite 37

4 How puzzles have been solved **39**
4.1 Sliding-block puzzles 39
4.1.1 $(n \times m - 1)$-puzzle 40
4.1.2 Rush Hour 40
4.2 Sokoban 40
4.2.1 Takakashi's unpublished effort 41
4.2.2 Lishout, Gribomont: Multi-agent approach 41
4.2.3 Takes: reversed solving 42
4.2.4 Binary decision diagrams 43
4.3 Atomix 45
4.4 Attempts at solving games with reasoning 46
4.4.1 Go 47
4.4.2 Othello 47
4.4.3 Connect-Four 47

5 Considering algorithms **49**
5.1 Search 49
5.1.1 Search algorithms 49
5.1.2 Partial IDA* 50
5.1.3 Enhancing A* with bitstate hashing 50
5.1.4 Solving an easier version 51
5.1.5 MGSS* 51
5.2 Data structures 51
5.2.1 Binary decision diagram 51
5.3 Reasoning 55
5.3.1 Rule-based systems 57
5.3.2 Model-based reasoning 57
5.3.3 CBR 58
5.3.4 Other methods 58
5.4 Other issues regarding metareasoning 59
5.5 What we will implement 61
5.5.1 CBR features 62

6 System design **63**
6.1 System overview 63
6.2 Solver 63
6.3 Meta-reasoner 65

6.3.1 CBR . . . 66

7 Implementation 69
7.1 Solver . . . 69
7.1.1 Communication between the meta-reasoner and the solver . . 69
7.1.2 Display of solution sequence . . . 70
7.1.3 New block types . . . 71
7.1.4 Storing positions more efficiently with permutation rank . . . 74
7.1.5 Feature extraction . . . 75
7.2 Meta-reasoner . . . 76
7.2.1 Case base . . . 77
7.2.2 Measuring similarity . . . 78

8 Experimental results 79
8.1 How the tests were run . . . 79
8.2 Test suite and case base . . . 80
8.3 Raw results . . . 81
8.3.1 Results from running the meta-reasoner . . . 81
8.3.2 Comparing the meta-reasoner against a few preselected parameter settings . . . 85
8.3.3 Memory usage of permutation rank . . . 85
8.3.4 Solving an easier version of the puzzle . . . 91
8.4 Problems encountered during testing . . . 92

9 Discussion 97
9.1 Solver . . . 97
9.1.1 Use the solution of an easier version to solve the full puzzle . 97
9.1.2 Permutation rank . . . 100
9.1.3 New block types . . . 100
9.2 Meta-reasoner . . . 101
9.2.1 Meta-control cycle . . . 101
9.2.2 Matching of cases . . . 102
9.2.3 Finding similarity between cases . . . 104
9.2.4 Randomized proximity search . . . 105
9.3 Other issues . . . 105

10 Summary and conclusion 107

11 Future work 109
11.1 System architecture . . . 109
11.2 Meta-reasoning program . . . 109
11.2.1 Case matching and similarity . . . 110
11.2.2 Other AI methods . . . 110
11.3 Solver program . . . 110
11.3.1 Pattern databases . . . 110
11.3.2 Macro moves . . . 110

11.3.3 Moves 110
11.3.4 BDD 111
11.3.5 Optimizations for speed 111
11.3.6 Optimizations for reduced memory usage 111
11.4 Other areas of improvement 111
11.4.1 Parallelism 111
11.4.2 External storage 112
11.4.3 GPU 112
11.5 Open problems 112

List of Figures

2.1 Meta-level reasoning 26

3.1 The sliding-block puzzle *Forget-me-not* 27
3.2 Different ways to define an action 28
3.3 Two new block types. 29
3.4 The 15-puzzle in a solved state. 30
3.5 A Rush Hour puzzle. 30
3.6 Meta-level reasoning 31
3.7 Space value example 34
3.8 Resting block pruning 35
3.9 Some puzzles requiring different solving strategies 37

4.1 A screenshot of the original DOS version of Sokoban. 41
4.2 Binary decision diagram for the majority function. 43
4.3 A computer implementation of Atomix. 45

5.1 The sliding-block puzzle *Forget-me-not* 52
5.2 Two similar-looking puzzles, only one is possible to solve. 59

6.1 Screenshot of the meta-reasoning program 64

7.1 File format 71
7.2 Puzzles with disconnected blocks and sliders 72
7.3 File format example for new block types 73

8.1 Positions examined for solved puzzles, all three methods 87
8.2 Positions examined for solved puzzles, two methods and all puzzles . 87
8.3 Progress for unsolved puzzles, all three methods 88
8.4 Progress for unsolved puzzles, two methods and all puzzles 88
8.5 Puzzle 23 and two easier variants. 91
8.6 Puzzle 23 variant 1, subgoals ordered by number of steps needed . . 93
8.7 Puzzle 23 variant 2, subgoals ordered by number of steps needed . . 93
8.8 Puzzle 23 variant 1, subgoals ordered by search effort 94
8.9 Puzzle 23 variant 2, subgoals ordered by search effort 94

9.1 A subgoal which is impossible to reach. 98
9.2 A hard transition between two positions 99
9.3 Puzzle 02/44 (a)) and puzzle 13 (b)) 104

List of Tables

3.1 Pairwise distances for space value calculation. 34

4.1 Truth table for the majority (median) function. 43

8.1 Overview of which puzzles we solved. 82
8.2 Results for puzzles in the case base 83
8.3 Results for puzzles not in the case base 84
8.4 Detailed results of running all methods on all puzzles. 86
8.5 Runtimes of permutation rank and Huffman coding. 89
8.6 Number of bytes needed for permutation rank and Huffman coding. 90
8.7 Number of bytes actually needed by STL 91
8.8 Results from solving the easier variants of puzzle 23 92
8.9 Results from solving the full version of puzzle 23 92

9.1 Results for puzzles in the case base 103

Chapter 1

Introduction

1.1 Goal

The goal of this thesis is to develop a method for solving sliding-block puzzles using meta-reasoning. This will be achieved by studying existing literature in order to find out how to apply meta-reasoning to our domain.

We will analyze the methods, and find one or two variants which we will implement on top of an existing sliding-block puzzle solver. Then we will run experiments and evaluate the performance of the methods.

In addition, we will also attempt to enhance the methods used in the existing solver program. We are interested in algorithmic enhancements which enables us to search more efficiently, as well as optimizations and prunings which leads to a smaller search space size. Both kinds of improvements will be beneficial at the meta-level, because a smaller portion of the resources is spent doing search.

1.1.1 Our previous work

This thesis is a continuation of the work done in the specialization project. The result from this project was a sliding-block puzzle solving program using classical search algorithms, as well as several enhancements. The user decides which algorithm and options to select for a given puzzle. In this thesis, the meta-reasoning program should automatically find good options for a given puzzle.

1.2 Background and motivation

Sliding-block puzzles is a challenging domain where little research has been done. The domain of sliding-block puzzles is PSPACE-complete, and hence no efficient algorithm is known which can solve arbitrary instances. Research from similar domains has shown that it is possible to create programs able to solve most instances. The current best results in this domain are still far behind results from similar domains.

Previous authors [14] have claimed that breadth-first search is the best suited algorithm for sliding-block puzzles. Hobbyists have used various heuristics and domain-specific algorithms to solve harder instances. In our specialization project (see section 1.2.1), we used several algorithms (breadth-first search, A* and IDA* (which are described in detail in chapter 2), as well as adding several ways of pruning the search space.

Since the domain of sliding-block puzzles is PSPACE-complete, there is reason to believe that other problem domains within the fields of AI and computer science could benefit from significant progress in solving sliding-block puzzles.

It must be mentioned that the author of this thesis has a strong background in algorithms, which can be reflected in some parts of the thesis.

Also, we (I) think sliding-block puzzles are hard and fun, both as a recreational game and as a scientific problem.

1.2.1 Specialization project

This thesis is a continuation of the author's specialization project at NTNU, which was conducted during Autumn 2009.

The goal of the specialization project was to study existing techniques for solving sliding-block puzzles, develop new ideas for solving such puzzles and incorporate these ideas into a program that solves instances of such puzzles.

A literature study was conducted where we investigated how others have tried to solve sliding-block puzzles, as well as studying problems exhibiting similar properties, like Sokoban, the 15-puzzle and Rubik's cube.

Then, we analyzed our problem domain, in order to find its search space properties and similarities to the other problem domain we studied. Based on the results of the analysis, we came up with a set of suggested algorithms, algorithmic improvements and ways to prune the search space.

The program we developed used the following search algorithms, improvements and options:

- Choose between three algorithms: BFS, A*, IDA* (described in more detail in chapter 2)

- For A* and IDA*, choose between three heuristic functions.
- For IDA*, manually set the initial pathlimit (the maximal permitted search distance during the first iteration of the search).
- Deleting inferior states and resuming search when memory is exhausted.
- Prune positions where the spaces are far apart (*space value* pruning).
- Prune positions containing blocks not packed in a corner.
- Disallow movement of certain blocks in a puzzle.
- Only allow master block moves that decrease its distance to the goal.
- Solve a puzzle with the aid of user-supplied subgoals.

In addition, the program supports two ways of calculating an upper bound of the search space of a puzzle:

- **Combinatorial bound** (or permutation bound). Given a puzzle, each block (including spaces) can be seen as elements in a permutation, where similar elements are allowed. The multinomial coefficient of this multiset is an upper bound of the search space for this puzzle. This bound is easily calculated for every puzzle.
- **Packing bound**. Given a puzzle, count the number of ways it is possible to pack the blocks in the puzzle area. This bound is not feasible to calculate for larger puzzles having search spaces of 10^{20} and higher.

These enhancements are described in further detail in section 3.2.1.

A test suite of 26 puzzles was constructed. We ran our program on each puzzle in the test suite with different settings, in order to measure the efficiency of each algorithm and improvement.

We managed to solve 15 of the 26 puzzles. Our results showed that A* performed slightly better than BFS on our domain. In addition, finding the correct parameters for the A* heuristics was important for solving some puzzles. The search work needed could differ by several orders of magnitude for different parameters on the same puzzle.

1.3 Literature study

In this section, we describe how the literature study was conducted in this thesis. More specifically, we will mention what goals we had for the literature study and we will describe the process we used for finding relevant literature, and outline the plan we used for analyzing each paper.

The literature was grouped in the following categories:

- Meta-level reasoning - everything which has to do with reasoning about the reasoning process
- Reasoning - classical reasoning methods and their use in games
- Search - the algorithms that search the state space
- Optimization - optimizing and tweaking of performance

1.3.1 Goal of the literature study

The goal of the literature study was to find literature which would help us in reaching the goals for this thesis.

1.3.2 Method

The following methods were used to find relevant literature.

As the main tool, `scholar.google.com` was used to find papers, where we searched for relevant terms like meta-level reasoning, search, planning, sliding-block puzzles and similar puzzles like Sokoban, 15-puzzle and Atomix. In addition, our specialization project report [32] was included as relevant literature.

We also checked references for the papers we found, and we searched for papers having our newly found papers as references (citations).

In addition to using `scholar.google.com`, we searched for relevant conferences. We also checked the homepages of some authors, if we discovered that they have written several relevant papers or if their area of research was relevant to us. In one case, we emailed the author in order to get a paper that wasn't available online.

When a paper was found, it was placed in one of the categories mentioned above. At the very least we read the abstract in order to find out the topic of the paper, as well as maintaining a separate file containing comments about each paper.

For each of the categories, we then collectively analyzed all the useful papers together, discussed the various approaches and compared them to each other (see chapter 5).

1.4 Structure of this thesis

Chapter 2 is an introduction to common search algorithms (both uninformed and informed) and reasoning methods. Knowledge of these is a prerequisite for understanding the rest of the thesis.

In chapter 3 we define our problem domain, both at the search level and at the meta-level. We also look at some common puzzles that appear in our domain as subdomains.

In chapter 4, we look at how sliding-block puzzles have been solved before. We also look at how similar puzzles like Sokoban and 15-puzzle have been solved, both with search algorithms and methods using knowledge.

Chapters 2-4 can be considered a part of the literature study, while chapter 5 contains discussion and decisions we make based on the literature study.

Chapters 6-7 describes our implementation.

Chapters 8 and 9 contain results and evaluation.

Chapter 10 contains a conclusion and a summary.

Chapter 11 identifies areas where further research can be done.

1.5 Credits

All the pictures of sliding-block puzzles and configurations are screen-captured from the Bricks computer game [5]. These pictures mainly appear in chapter 3, but also they also appear in chapters 5, 7, 8 and 9.

Figure 3.4 is taken from the Wikipedia article about the 15-puzzle [10].

Figure 3.5 is taken from a website where Rush Hour can be played online [28].

Figure 4.1 is taken from the Wikipedia article about Sokoban [31].

Figure 4.2 is taken from Knuth [20].

Figure 4.3 is taken from the website of Atomix 2.13.4 for GNOME [3].

Chapter 2

Classical algorithms

In this chapter we will briefly introduce the classical algorithms used for single-agent problems, as well as reasoning methods. This chapter is mainly here to familiarize the reader with these algorithms, since many of them will be referred to many times later in this thesis.

2.1 Uninformed search algorithms

This section will introduce classical uninformed algorithms for traversing a search space.

2.1.1 Random walk

The *Random walk* algorithm works as follows: For each move, move to a randomly chosen neighbouring state. States are chosen with uniform probability.

Even though this algorithm might sound trivial and silly, it can be useful in cases where there is a high number of solution states in the search space, we don't care about an optimal solution sequence, or if we have little domain knowledge.

Junghanns [17] suggests using the following improved algorithm: Each time a new state is visited, push all its unvisited neighbouring states into a set of open states. Then, select and visit a random state from the set of open states, removing the state from this set in the process.

2.1.2 Depth-first search

Depth-first search (DFS) is a graph traversal algorithm that expands all the nodes in an entire subtree before examining any of the sibling nodes.

The algorithm works as follows: As long as there exists a node with unvisited children, visit the first child node. When there are no such nodes, backtrack to the parent node.

If the graph has cycles, the basic algorithm will enter an infinite loop (recursion). Also, if the graph isn't a tree, some nodes can be visited multiple times. To avoid this, we modify the algorithm to not visit any node more than once.

Standard DFS only uses memory proportional to the current search depth. However, when the graph is not a tree it is favourable to also keep track of the visited nodes. We are not guaranteed to find a solution node in the shortest amount of steps using DFS.

2.1.3 Breadth-first search

Breadth-first search (BFS) is a graph traversal algorithm that visits all sibling nodes before visiting any child nodes. All nodes n steps away from the starting node are visited before any node that are $n + 1$ steps away.

BFS will find a solution node in the graph if one exists. Also, the shortest path to the solution will be found.

The main disadvantage with BFS is the memory usage - a queue containing all the nodes to be visited needs to be maintained.

2.1.4 Bidirectional search

In bidirectional search, search is performed simultaneously from the start node and the goal node(s). A solution is found when the two search trees meet in the middle.

Bidirectional search is often used in combination with BFS. Consider a graph with branching factor b and distance from start to goal d. Using BFS, a solution can be found using $O(b^d)$ work. Using bidirectional search, a solution can be found after doing two searches, each with $O(b^{d/2})$ work.

Some disadvantages with bidirectional search includes extra logic needed in order to determine when the two search trees meet. In some graphs, there can be several goal nodes, which can cause the second search tree to become much larger than the first one, reducing the advantage. In some cases it can be difficult to define the backward action needed to traverse the graph from the goal node.

2.1.5 Iterative deepening

This is not an algorithm in itself, but an enhancement which can be applied to non-breadth-first algorithms. It is typically used in combination with A* (see section 2.2.1) and DFS.

Iterative deepening works as follows: A series of searches are performed (using A*, DFS or other algorithms), with the constraint that the search is not allowed to search past a certain depth limit l. If a search is completed and no solution is found, the depth limit l is increased and a new search is performed. As long as the depth limit l is increased by the least edge cost in the graph, iterative deepening will find an optimal solution.

For a well-behaved search tree where the branching factor is equal everywhere, the last iteration will dominate all the others in search effort. If the solution is at depth d, the runtime is $O(b^d)$ which is the same as BFS, but with less memory usage.

2.2 Informed search algorithms

This section will introduce classical informed algorithms for traversing a search space. More specifically, we will introduce a class of search algorithms called *best-first search algorithms*. These algorithms are equipped with a heuristic function which estimates the distance from any state to the goal. Using this heuristic function, these algorithms attempt to pick states more likely to have a short distance to a solved state.

First, we will introduce some common terminology for the informed algorithms:

- **Heuristic function**: A function h which, given a state, returns the estimated distance to goal.
- **Admissible**: A heuristic function is *admissible* if it never overestimates the distance to the goal. That is, for every state x, $h(x) \leq h^*(x)$ where h^* is the function returning the optimal distance to the goal.
- **Monotone**: A *monotone* (or *consistent*) heuristic is one that approaches the goal in an incremental way, without taking a step back. More formally, for each state x, with parent state p and cost $c(p, x)$ for moving from state p to x, the following inequality always holds:

$$h(x) \leq h(p) + c(p, x)$$

2.2.1 A*

A* is a best-first search algorithm which use a *heuristic function* to guide the search. Whenever a node is chosen for expansion, the node x with the lowest f-

value is chosen, where $f(x) = g(x) + h(x)$. $g(x)$ is the actual cost of moving from the start position to node x, and $h(x)$ is the estimated distance from x to goal.

If the heuristic h is *admissible*, A* will find the optimal solution if a solution exists.

The main disadvantage of A* is its memory usage. A priority queue needs to be maintained, containing all the unexplored neighbours to nodes we have visited so far. Also, a set containing all the nodes already seen must also be maintained.

2.2.2 IDA*

IDA* is a variant of A*, where iterative deepening is applied in order to reduce the memory usage of the A* algorithm.

IDA* works as follows: A series of A* searches are performed with the additional constraint that no node x with $f(x) > p$ is expanded, where p is the pathlimit for the current search. If no solution is found during this search, the pathlimit p is raised and a new search is performed.

If IDA* is given an admissible heuristic h and a solution exists, IDA* will find the optimal solution. If the search tree has approximately the same branching factor everywhere, IDA* will use asymptotically the same time as A* to find a solution, as the time of the last iteration will dominate the sum of all previous iterations. In addition, the memory usage is less than A*, since IDA* doesn't need to push nodes with f-values exceeding the pathlimit to the open set.

2.3 Domain-independent enhancements

This section describes some common algorithmic enhancements to the search algorithms mentioned above.

2.3.1 Pattern databases

In a *pattern database*, a subdomain of the full problem domain is considered, and the exact distance from any node to a solution is stored. The subdomain is typically chosen so that it is feasible to store the distance from all nodes. The construction of a database is usually done in advance, before solving instances of the problem in question.

A typical application of a pattern database is to improve the lower bound of the heuristic function. For any given position, it is converted to the subdomain and is looked up in the pattern database. The distance from this position to goal is used as the estimated distance to goal.

2.3.2 Macro moves

A *macro move* is a kind of super-action - performing several actions in a row as one atomic action. This has the effect of reducing the size of the search space when the macro moves replace the individual actions they consist of.

Macro moves are mainly useful when there exist sequences of actions where the we are not interested in the intermediate positions resulting from this sequence.

Macro moves should be carefully defined if one is interested in preserving optimality.

2.3.3 Transposition tables

Transposition tables are used to avoid re-evaluating nodes in graph search algorithms, and they work by caching the result from the evaluation of the node, making it available for later retrieval. The need for transposition tables typically arise when the search graph contain cycles or multiple paths to a specific node.

Transposition tables are commonly used in search in two-player games, but can also be used in conjunction with algorithms like IDA*.

A disadvantage of using transposition tables is the memory usage. To help remedy this, one could implement a replacement scheme. When the transposition table becomes too large, entries can be dropped. It could be the entry that is least recently used, least frequently used, the entry corresponding to the node farthest away from a goal state (as measured by a heuristic function) or some other criteria.

2.3.4 Endgame databases

An *endgame database* is a pre-calculated database containing the evaluation of every node that are up to a certain distance from a goal node.

For single-agent problems, the endgame database typically contains all nodes up to a distance d from a goal position. For each such node, the exact distance to a goal node is stored.

For two-player games, the endgame database typically contains the result of the game for each node in the database, and might also hold the number of moves to either the end of the game, or to a position where it is trivial to determine the game result.

2.4 Methods based on expert knowledge

In this section will briefly describe problem solving methods based on expert knowledge. Luger [27] is used as a reference for these, except case-based reasoning where

we have used Aamodt, Plaza [1] as a reference.

2.4.1 Rule-based reasoning

Historically, *rule-based systems* have commonly been called *expert systems* in literature. The knowledge base in rule-based systems are encoded as a rule base containing several *if...then...* rules.

An *inference engine* is given the premises of a problem instance (the user input), from which it will try to deduce further facts, according to the rules. A *working memory* is used, which initially only holds the premises, but later on it will also contain additional information derived from the premises by the inference engine. The inference engine performs a recognise-act cycle, which involves going through the rules multiple times using the information in the working memory. When a rule is found such that the premises of the rule is matched by the information in the working memory, the information in the *then* portion of the rule is added to the working memory. When no more matches can be made, the inference engine is done and the working memory will, hopefully, contain some kind of conclusion.

Rule-base reasoning is one of the oldest techniques for representing domain knowledge in such a system, and is still widely used.

2.4.2 Case-based reasoning

Case-based reasoning (CBR) is a way of solving new problems by remembering a previously solved similar problem and reusing information and knowledge from the stored solution. CBR systems typically have a *case base* of previously solved problem instances along with their solutions. When a new problem needs to be solved, a similar past case is found, and reused to solve the new problem.

The general CBR cycle may be described by the following four processes:

- **Retrieve**: Find the case or cases most similar to the problem we want to solve. In this stage, our new problem is analyzed and a set of problem descriptors, *features* are found. Then, a search is performed through the database of previously solved problems to find the best match(es).

- **Reuse**: Use a retrieved case to solve the new problem. This can be done by just copying the case as applying it to the solution as-is, or adapting it to better fit the problem.

- **Revise**: Apply the case to the new problem, and determine whether it solves the problem. If the problem is successfully solved, we can use the solution to solve future problems by retaining the case. If the problem is not solved, the solution can be repaired using domain-specific knowledge.

- **Retain**: Keep the solution to this problem, and make use of the knowledge for solving future problems. This is the learning phase of CBR. The case base is updated according to the result of applying the previously retrieved case to our new problem. If the problem was solved, the case base is updated with the information that the problem can be solved with this case. If the problem was not solved, the case base can be updated with information about the failure, in order to prevent similar failures in the future.

2.4.3 Model-based reasoning

Model-based reasoning is based on a deeper understanding of the problem domain. By seeking deeper understanding, one hopes to improve upon systems based on making decisions based on observations. For example, when attempting to find a medical diagnosis, one can make a diagnosis on observable symptoms like headaches, nausea and fever. This is the approach used by traditional reasoning system based on heuristics. Such heuristics do not give us any deeper and causal understanding of the human body. A model-based system would in this case use a model of the human body, and would try to detect the presence of infecting agents, note the resulting inflammation of cell linings, the presence of inter-cranial pressures and infer the causal connection to the symptoms of headache, elevated temperatures and nausea.

Modelling of electronic circuits is an early application of model-based reasoning. For such domains, the physical model is simulated in software.

An advantage with model-based reasoning is that the resulting system is typically robust and thorough. However, such a system is still a model, and could be wrong or not detailed enough. Also, the knowledge acquisition for a model-based system can be demanding and the resulting system can be large.

2.4.4 Hybrid design

It is possible to combine two or more of the reasoning approaches mentioned above. The strengths of one system can compensate for the weaknesses of another. For example, by combining rule-based and model-based systems, one can use the model-based approach when given a problem that cannot be solved by the heuristic rules of the rule-based system.

2.5 Meta-level reasoning

In artificial intelligence, reasoning has traditionally been viewed as a decision cycle within an action-perception loop. The loop consists of an intelligent agent at the *object level* performing actions in an environment (the *ground level*). The cycle

consists of the agent performing actions and receiving stimuli from the environment based on the actions. New actions are then performed, and the cycle goes on.

Cox and Raja [6] define *Meta-level reasoning* (or *metareasoning*, as they call it) as the process of reasoning about this decision cycle. The action-perception cycle along with an additional meta-level is shown in figure 2.1.

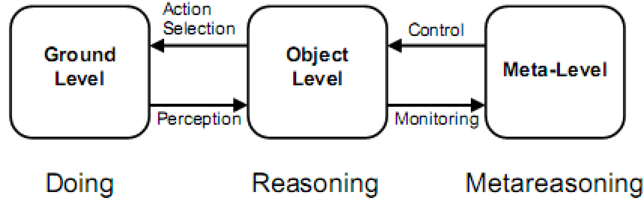

Figure 2.1: Meta-level reasoning

Using *meta-level control*, we can spend extra computational effort to decide how much and what kind of reasoning to do. Some kind of balance must be struck between the amount of computations at the meta-level and the object level.

In order to gather sufficient information to make efficient meta-level decisions, *introspective monitoring* is necessary. This can involve gathering of performance data for the different kinds of decision algorithms at the object level, building up a performance profile for each option.

Chapter 3

Sliding-block puzzles defined

3.1 Definition of the problem domain

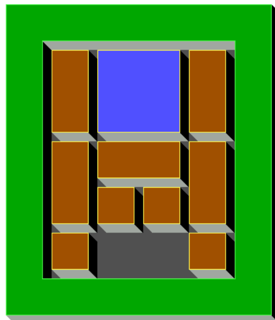

Figure 3.1: The sliding-block puzzle *Forget-me-not*

A sliding-block puzzle consists of several blocks within a closed frame where each block is a collection of unit squares which move together. The goal of such a puzzle is to reach a certain configuration. Typically, the goal specifies the destination location for a special block called the master block.

Figure 3.1 shows a typical sliding-block puzzle. The objective of this puzzle is to move the blue block to the center of the bottom row.

We will support the following kinds of blocks in this thesis:

- **Normal block**: The default block type. Two blocks of the same size and shape are considered to be indistinguishable.
- **Unique block**: Same as the normal block, but it is distinguishable from

normal blocks of the same size and shape. This block type is commonly used for the master block.

- **Disjoint block**: It behaves like a normal block, but a block of this type has disjoint unit squares. This is a new addition to our program from our specialization project [32].
- **Slider**: A block with additional movement restrictions: A block can be defined to be able to move only horizontally or only vertically. These blocks are typically of size $1 \times n$ or $n \times 1$. This is another new addition to our program.

The domain consists of positions in a search space. We can define two different actions in our search space:

- **Step**: A step is defined as moving one block one unit step either up, down, left or right. See figure 3.2 a).
- **Move**: A move is one or more consecutive steps using the same block. See figure 3.2 b).

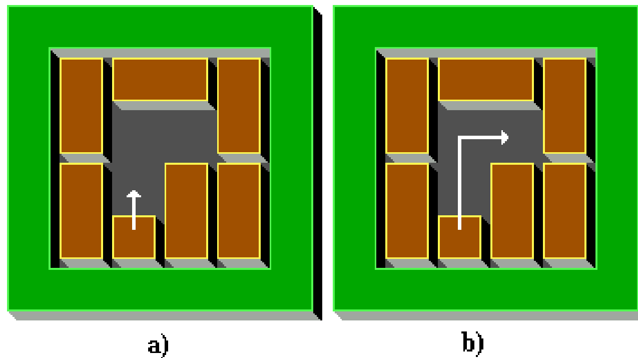

Figure 3.2: Different ways to define an action

Using moves and steps have their own set of advantages and disadvantages.

Advantages with moves:

- Shorter solution sequences and smaller diameter of search space
- Resting block pruning is expected to be more efficient
- Space value pruning is expected to be more efficient

Advantages with steps:

- Lower branching factor
- Easier to design a heuristic function

The solver program from the specialization project only supports steps.

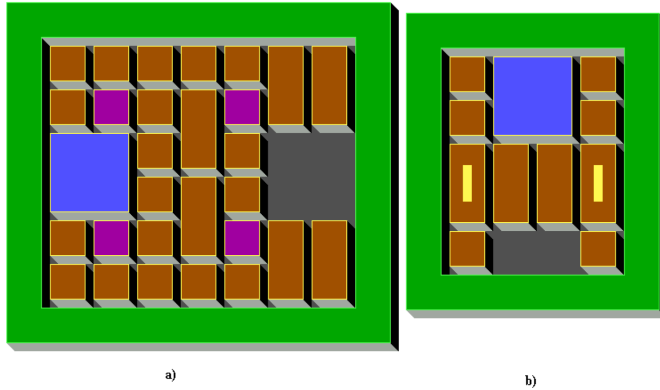

Figure 3.3: Puzzles containing the new block types. The left figure shows a disconnected block, which consists of four purple 1×1 cells. The right figure contains two slider blocks, indicated by a yellow vertical line in its interior.

As mentioned above, two block types are added to the program from the specialization project: disjoint blocks and sliders. The rationale for this addition is to be able to include more puzzles in our test suite, as well as increasing its diversity. The addition of sliders allow us to include puzzles from Rush Hour, as well (see section 3.1.2).

Figure 3.3 a) shows a puzzle containing a disjoint block. The four purple 1×1 cells belong to the same block and move together. Figure 3.3 b) shows a puzzle containing two vertical sliders. The sliders are the two blocks with a yellow vertical line.

3.1.1 15-puzzle, $(n \times m - 1)$-puzzle

This puzzle takes place in a grid of size $n \times m$. There are $n \times m - 1$ numbered tiles. The objective is to rearrange the numbered tiles so that they appear in increasing order. This puzzle is a special case of our domain, where all the blocks are of size 1×1 and distinguishable from each other.

The most well-known variant is the 15-puzzle, which was invented by Noyes Chapman, and popularized by Sam Loyd in 1880 [30]. In recent AI research [22, 23], the 24-puzzle (grid size 5×5) has been used as a problem domain because of the increased challenge compared to the 15-puzzle.

Figure 3.4 shows a solved configuration of the 15-puzzle.

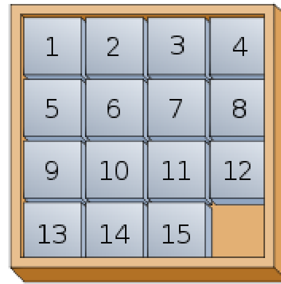

Figure 3.4: The 15-puzzle in a solved state.

3.1.2 Rush Hour

Rush Hour is a sliding-block puzzle manufactured by ThinkFun. The board size is 6×6, and all the blocks have sizes 2×1 or 3×1 (oriented horizontally or vertically). Each block can move either horizontally or vertically, in the direction they extend. The blocks represent cars and trucks, and the objective is to move a specific car out of the board via a gap in the wall.

Rush Hour is a special case of our problem domain, where all the blocks are sliders.

Figure 3.5: A Rush Hour puzzle.

Figure 3.5 shows a Rush Hour puzzle.

3.2 Definition of our meta-level domain

Cox, Raja [6] describe a model divided in a ground level, object level and meta-level (see figure 3.6). We described this model in section 2.5). This section describes one possible way to fit our problem into their model.

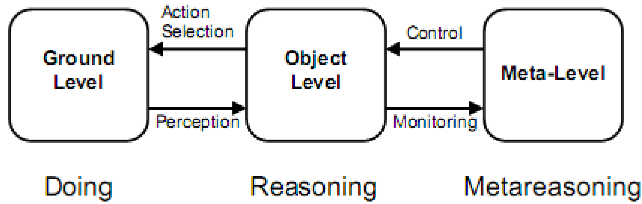

Figure 3.6: Meta-level reasoning

The ground level consists of the physical puzzle (or a human-playable computer implementation) and its rules. The object level consists of a representation of the puzzle in a computer, a set of actions (steps), a state space of all positions, and algorithms to examine the state space in order to try to find a solution to the puzzle.

The meta-level consists of controlling the object level. At this level, we control the strategies that the object level uses. By monitoring the object level, the agent at the meta-level gathers information such that it can make informed decisions about controlling the object level. The meta-level domain can be called the *parameter space*.

With *Introspective monitoring*, the meta-reasoner can gather information about the computational requirements and the performance of the various algorithms (BFS, A*, IDA*) with different parameters.

Using *meta-level control*, the meta-reasoner can decide whether to carry on with the current plan, or to cancel it and attempt to solve the problem with another strategy. A *strategy* in our domain is the set of *parameters* that the underlying solver is invoked with. (A list of the parameters is given in section 1.2.1. Parameters include, among other things, the choice of a search algorithm.) In our domain, this is analogous to determining whether the current search will be successful, cancelling it if it has exceeded its given allotment of resources (in time and memory) or determining if the puzzle is unsolvable with the current set of parameters. A change of strategy can involve either selecting another search algorithm, changing any of the other parameters, or even giving up (when we have exceeded our computing resources).

In our specialization project [32], we didn't have any automated reasoning at the meta-level. The user supplied the program's parameters. They were chosen by manually analyzing the starting position of a puzzle, using insights gained from playing the puzzle in a computer implementation, and by experimentation.

It is desirable that an automated meta-reasoner will have these possibilities:

- Starting a search with a given set of parameters.
- Monitoring the object level, both during the search and receiving a result after the search is finished. Store the information gathered from monitoring.
- Cancelling a search in progress.
- Based on the accumulated knowledge, come up with new strategies which will hopefully have a higher chance of success.

Our problem at this level is to select parameters such that a given instance will be solved, preferably with as little search work as possible. Experiences from the specialization project showed that this problem was very hard. There were a couple of puzzles which, according to our (not necessarily correct) judgment as experienced sliding-block puzzle solvers, doesn't look harder than a couple we did solve, but we weren't able to find parameters to enable these puzzles to be solvable within reasonable time by our program.

3.2.1 Properties of the meta-level domain (parameter space)

Our solver program has a multitude of options and parameters allowing the user or the meta-reasoner to change the parameters to fit any given puzzle. From the parameters we can create a multi-dimensional space, with one dimension for each option. Here follows a list of the parameters:

- Selection of search algorithm: BFS, A*, IDA*.
- Heuristic (A* and IDA* only): admissible or weighted non-admissible, the last heuristic has additional parameters for total weight, distance-to-goal weight and weight of average location of spaces (all weights are real values).
- Space value pruning threshold (integer value, or turned off (which is equivalent to a space value of ∞)).
- Resting blocks pruning: In the presence of hanging blocks, don't allow moves of resting blocks (three settings: off, relaxed, strict).

These parameters alter the search space of a puzzle. For instance, each puzzle has a threshold t such that it is solvable if and only if the chosen space value is $s \geq t$.

The subset of parameters making a puzzle solvable varies heavily from puzzle to puzzle. For trivial puzzles, pretty much any settings of the parameters will let us solve them, except when we use settings which render the puzzle unsolvable, for instance by setting a too low space value cutoff. Finding a set of parameters that

solves a given puzzle can be a very difficult task. A given option has an effect on the search space which can be hard to analyse. Also, experimenting with different parameters can be very time consuming, as a solver run can take several hours. In the specialization project, we managed to solve a puzzle with a set of parameters which was particularly tricky to find. An off-by-one change in the space value cutoff parameter would cause the puzzle to not be solved.

In-detail description of parameters

This section describes the parameters, prunings and heuristics in detail.

Heuristics

- Heuristic **h0**: Manhattan distance between master block and goal. This heuristic is admissible.
- Heuristic **h1**: A slight improvement over the Manhattan distance heuristic. It tries to estimate the number of steps needed in order to move the master block one step. This estimate is based on the size of the master block and the number of spaces. This heuristic is not admissible.
- Heuristic **h2**: Overestimating heuristic, taking into account distance between master block and goal, collective distance between spaces and master block, and space value. This heuristic is not admissible.

Space value pruning

Let $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ be the coordinates of the spaces in a given position. The space value of this position is then

$$\sum_{1 \leq i < j \leq n} (|x_i - x_j| + |y_i - y_j|),$$

that is, the sum of Manhattan distances of all pairs of spaces. The purpose of this optimization is the idea that positions where the spaces are scattered far apart are less useful and will not be part of a solution. In our program, we can set a cutoff t such that all positions with a space value higher than t will never be visited in the search. They are removed from the search space, and hence we reduce the search space size, making the problem slightly more tractable (hopefully). The hard part is finding a good cutoff t such that the search space size is reduced significantly (by several orders of magnitude) while still allowing the puzzle to be solvable. Setting t too low results in the puzzle being unsolvable. In addition to reducing the search space size, a lower cutoff value t can increase the solution length, and in some cases increase the search work needed to find a solution.

Consider the two positions shown in figure 3.7. Each positions comes from the same puzzle. Each space is numbered from 1 to 4. Table 3.1 show the pairwise Manhattan distances.

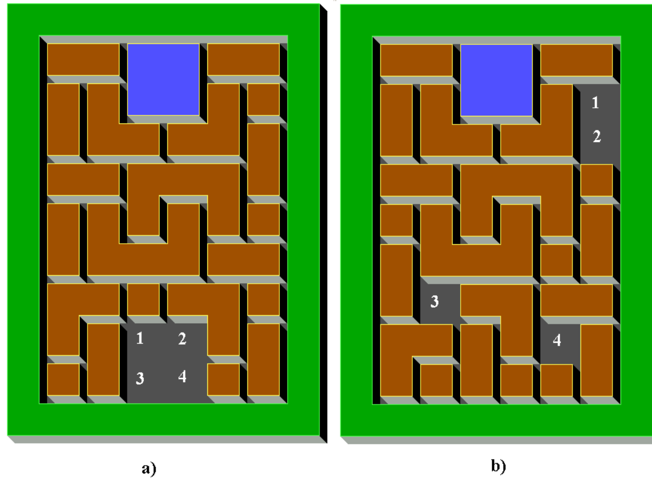

Figure 3.7: Space value example

Table 3.1: Pairwise distances for space value calculation.

Position a)					Position b)				
$i \setminus j$	1	2	3	4	$i \setminus j$	1	2	3	4
1	-	1	1	2	1	-	1	8	7
2	-	-	2	1	2	-	-	8	6
3	-	-	-	1	3	-	-	-	4
4	-	-	-	-	4	-	-	-	-

The space value is the sum of all pairwise distances, so the space values of the positions shown in figure 3.7 are 8 for position a) and 34 for position b).

The space value can be seen a numeric manifestation of the mobility of a position. A lower value means that the spaces are grouped more together (or that the blocks in the position are packed without gaps), which in turn means that larger blocks can move, and a block is more likely to be able to move several steps in a row. See figure 3.7, where position a) is superior to position b).

The ability to remove positions with a high space value from the search space means that a solver program can find a solution with less search work.

Resting block pruning

The idea is that when moving a block, it will be moved all the way to a sensible location, and will be left adjacent to other blocks (or the frame) such that at least two of its sides (one vertical and horizontal size) rests against other blocks (or the frame). By removing positions from the search space where blocks we move are left hanging in "mid-air", we hope to achieve significant savings in search space size. This optimization works best with moves, rather than steps. When using steps we naturally need to investigate all positions on a block's path to its desired location. When using steps, we simply need to avoid moving other blocks as long as there is a block which is "hanging". When using moves, we simply don't put positions into the queue where the current moving block hangs in mid-air.

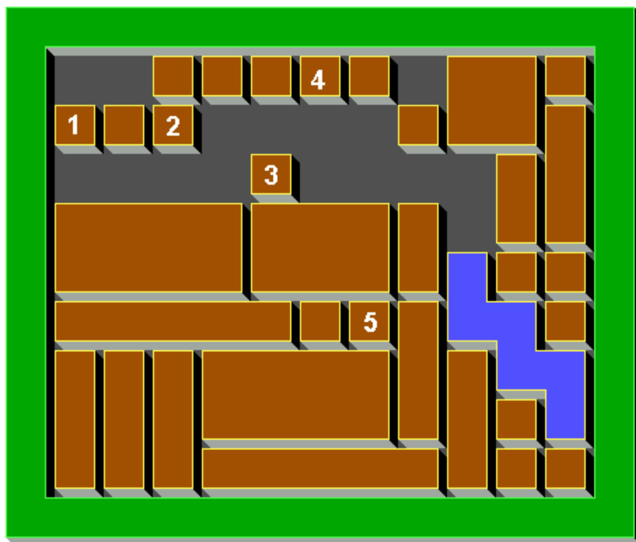

Figure 3.8: Resting block pruning

Figure 3.8 shows several examples of resting and hanging blocks. Blocks 2, 4 and 5 are resting. Block 2 can't move to the left and up, block 4 can only move down and block 5 cannot move at all. Hence, these blocks have two orthogonal directions where they rest against other blocks. Blocks 1 and 3 are hanging. Block 3 can move in all directions, except down. Block 1 rests against the frame to its left and a block to its right. However, these directions are not orthogonal, and hence the block hangs in mid-air.

3.2.2 Puzzle features

This section will mention some features that are possible to extract either by analyzing the initial position of a puzzle statically, or which can be obtained via search (which can help in subsequent searches).

We can divide the features into quantitative features and qualitative features. Quantitative features are purely numeric in nature, while qualitative features attempt to capture some higher-level concepts.

Quantitative features:

- Estimate of search space size. The easy (non-tight) estimate is the combinatorial bound, the harder one (not always applicable) is the packing bound. If this number is low enough, we can perform a full search (BFS) and always obtain the shortest solution. Low search space size implies easy to solve, but not necessarily the other way around, as the solution position can be close to the starting position in the search space graph.
- Branching factor. The exact number can't be determined without doing a full search, but it can be estimated by performing a partial search. A high branching factor is bad news for a search algorithm, as it limits the search depth we can explore completely.
- Number of spaces. Immediately determinable from the starting position. A high number of spaces makes it easier to move the blocks around. It also creates many possible moves, contributing to the combinatorial explosion for a search algorithm.
- Block shape niceness. The number of rectangular blocks divided by the total number of blocks. A low number of rectangular blocks can be indicative of a harder puzzle where it is more difficult to move the blocks around and to keep them packed at all times.
- Average block size. Large blocks are harder to move around, and can contribute to the difficulty of solving a puzzle.
- Puzzle size. The number of non-wall squares in the puzzle. A large puzzle has a higher potential of being difficult.

Qualitative features:

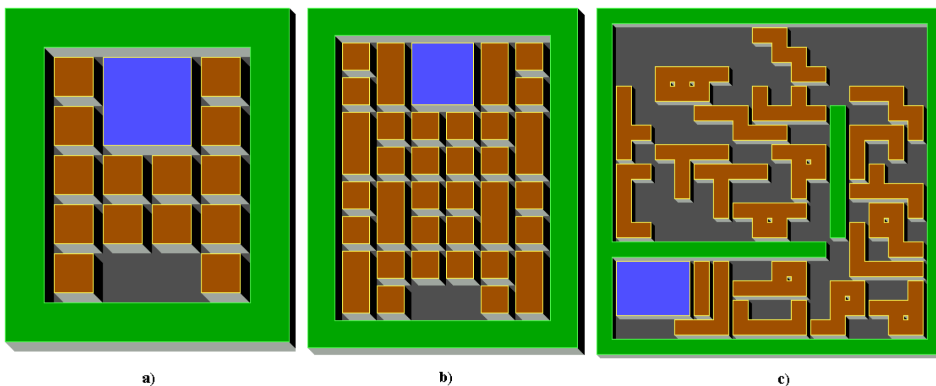

Figure 3.9: Some puzzles requiring different solving strategies. In puzzles a) and b), the objective is to move the master block to the center of the bottom row. In puzzle c), the objective is to move the master block to the upper right corner.

- Ease of movement: Takes the block sizes and shapes into account. Small, rectangular blocks make movement easy. Large, irregularly shaped blocks make movement hard. (A high branching factor doesn't necessarily imply easy movement. For instance, puzzles a) and b) in figure 3.9 have a low branching factor, but only 1×1 and 1×2 blocks, hence it's easy to move blocks around. Puzzle c) in figure 3.9 has a high branching factor, but the combination of blocks shaped like arbitrary polyominoes, the need to pack near-perfectly and the need to rearrange blocks multiple times make movement very hard. If it is easy to move blocks around, A* is a good candidate algorithm. Depending of the speed in which it is possible to move the master block (in moves needed per unit it is moved), adjust the heuristic weight.
- Type of instance: Is it a packing problem? Do we need to pack the rest of the blocks in the puzzle in some way before we can make a path for the master block? Then we first need to determine how the blocks should be packed and devise a plan to achieve this packing. It is a puzzle where we can do nothing but search? These puzzles are characterized by very little space, very small branching factor and long solution sequences. BFS is a good choice for those.

3.3 Our test suite

In order to facilitate testing and evaluation of our algorithm, we have chosen numerous puzzles that will be our test suite. These puzzles are taken from various computer implementation of sliding-block puzzles, most of them from Bricks.

The puzzles were mainly chosen based on the expected difficulty level. The difficulty level of the chosen puzzles were similar to or slightly harder than the hardest puzzles

solved in the specialization project.

In addition, some very easy puzzles containing the newly supported block types (disconnected blocks and sliders) were included, to confirm that our implementation was correct. We also constructed some puzzles for testing some of the extreme settings of our parameters.

In total, 46 puzzles were chosen, which includes the 26 puzzles used in the specialization project.

Chapter 4

How puzzles have been solved

In this chapter we will describe how sliding-block puzzles and similar single-player puzzles have been attempted solved. We will only briefly mention solving attempts that were already described in our specialization project [32], while describing in more detail the attempts found in the literature study of this thesis. At the end of this chapter, we will also mention some games-playing programs (not necessarily single-player puzzles) using reasoning.

4.1 Sliding-block puzzles

In this section we will all attempts at solving sliding-block puzzles that are known to us. None of the following approaches have used other methods than standard search algorithms like BFS, A* and IDA*.

In our specialization project [32], we developed a program supporting three algorithms (BFS, A*, IDA*) and a wide range of heuristics and options. A list of the options in the program can be found in section 1.2.1. Our program was able to solve 15 puzzles from a test suite of 26 puzzles.

The JimSlide program by Leonard [25] used BFS to solve sliding-block puzzles. Their program supports swapping positions to disk, hence it is capable of fully examining search spaces which cannot be held in main memory alone. No results of their program are given, but from the description of the program we believe it is able to solve the same puzzles as the program from our specialization project using BFS with none of the extra options turned on.

4.1.1 $(n \times m - 1)$-puzzle

Culberson and Schaeffer [8] used IDA* with pattern databases (to improve the lower bound heuristics), transposition tables (to avoid re-examining previously visited state) and endgame databases (store all states at most 25 moves away from the goal state). They achieve a 1707-fold reduction in search space size compared to IDA* with the Manhattan distance heuristic.

Korf, et al [22, 23, 24] have used multiple approaches for solving such puzzles:

- They managed to solve 10 random instances of the 24-puzzle (5×5), by using IDA* with 4 different heuristics combined into an admissible heuristic, as well as finite-state machine pruning to avoid duplicate states.
- They used parallel BFS to perform a complete search of the search space of the 15-puzzle, proving that no instance needs more than 80 moves.
- By using disjoint pattern databases, they achieve a 2000-fold reduction in solving time for 15-puzzles, as well as managing to solve 50 random instances of the 24-puzzle.

All of the solving attempts for $(nm - 1)$-puzzles mentioned here were found during our specialization project.

4.1.2 Rush Hour

The JimSlide program mentioned in section 4.1 can also solve Rush Hour puzzles, since they are essentially sliding-block puzzles where every block is a slider (that is, restricted to either horizontal or vertical movement). Since standard Rush Hour puzzles are always of size 6×6, all instances should be easily solvable using BFS.

When generalised to an arbitrarily large board, the problem of deciding if a Rush Hour puzzle has a solution is PSPACE-complete [11].

The program from our specialization project [32] doesn't support sliders, and cannot solve Rush Hour puzzles.

4.2 Sokoban

Sokoban is a puzzle game where the player pushes boxes in a maze. The purpose is to push all boxes to designated goal locations. Only one box can be pushed at the same time, and boxes cannot be pulled. Figure 4.1 shows an example of a Sokoban puzzle.

The program *Rolling Stone* by Junghanns and Schaeffer [18] was able to solve 57 problems from a standard test suite of 90 problems from the computer game

Figure 4.1: A screenshot of the original DOS version of Sokoban.

XSokoban[1]. They used IDA* with a multitude of enhancements (transposition tables, deadlock tables, macro moves, pattern search and relevance cuts.) Their effort was studied in our specialization project[32], and will not be mentioned in detail here.

The remainder of this section will list other attempts at solving Sokoban which we found during the literature study of this thesis.

4.2.1 Takakashi's unpublished effort

Takakashi [33] has released a Sokoban solving program on his website. According to the website, the program is able to solve 86 of the 90 problems from XSokoban, which is significantly better than any of the documented efforts at solving Sokoban. The program does not try to find optimal solutions. However, details about the algorithms are not given, except for a high level description which says that they use "Best-first search with Means-ends analysis as the basic search algorithm." The source code is not released as they mention that "the research is under continuation." A move is defined as multiple pushes of the same box in a row.

4.2.2 Lishout, Gribomont: Multi-agent approach

Lishout, Gribomont [26] used a multi-agent approach. They consider each box in the maze to be an agent whose aim is to reach one of the goal squares. The global goal us to find a solution where every agent reaches his objective. In this approach, the man is just a puppet which can be used by the agents when needed.

[1] http://www.cs.cornell.edu/andru/xsokoban.html

They define a special subclass of Sokoban positions which consists of the positions where it is possible to push the boxes directly to their goal positions one by one. This subclass is also sometimes referred to as the *Lishout subclass.* To be more precise, a position in this subclass must satisfy the following conditions:

- It must be possible to determine in advance the order in which the goal squares will be filled.
- It must be possible to move at least one box to the first selected goal square without having to move other boxes.
- For each box which satisfies the previous condition, the position obtained by removing that box and replacing the selected goal square by a wall must also be in the subclass.

Their program *Talking stone* used a rather naïve algorithm: Try all moves until a position is reached which belongs in the subclass defined above. They chose to use pure iterative deepening depth first search instead of IDA* because of the difficulty of designing a good heuristic function, and because it was expected that the moves needed to reach a position in the subclass were not expected to reduce the global box-goal distance. The program does not try to find optimal solutions.

They managed to solve 9 of the 90 problems from XSokoban. One problem was trivially solved (no search effort) because the starting position was already in the desired subclass.

4.2.3 Takes: reversed solving

Takes [34] used a different approach for solving Sokoban: They searched backwards from the goal position, where the man is *pulling* boxes instead of pushing them. This has the advantage of avoiding deadlock positions[2] altogether, and hence there is no need to detect deadlocks, which was a necessary feature in the *Rolling Stone* program mentioned above. Instead of deadlocks, it is possible, however, that the man can trap himself in a corner, surrounded by blocks.

They used a standard brute force search algorithm which has two main options: Condition X and condition Y. Condition X specifies when to stop moving a box. Options include after each step, until a box is at a final position, and until a box is k steps away from a final position. Condition Y specifies which box to pull next. Options include every box and every unplaced box. One specific set of conditions is especially well suited to solve puzzles from the *Lishout subclass* defined in the section 4.2.2. This approach does not try to find optimal solutions.

They managed to solve two puzzles of the 90 puzzles from XSokoban. In addition, they solved several puzzles from the easier Microban.

[2] A deadlock is a position from which it is impossible to reach a solution. This includes positions where a box is stuck in a corner, or there exists a group of 2×2 boxes.

Table 4.1: Truth table for the majority (median) function.

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

4.2.4 Binary decision diagrams

A *binary decision diagram* (BDD) is a data structure used to represent a boolean function. A BDD can be seen as a rooted, directed acyclic graph which consists of decision nodes and terminal nodes representing *true* and *false*. Figure 4.2 shows a BDD for the majority or median function shown as a truth table in table 4.1. This function returns true if at least two of its three inputs are true.

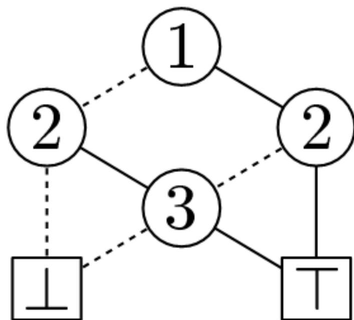

Figure 4.2: BDD for the majority (median) function. $\bot$ represents the *false* terminal node, $\top$ represents the *true* terminal node. The dotted lines represents *false* edges, and the solid lines represent *true* edges.

There are different kinds of BDD's, the most common ones are:

- **OBDD: Ordered BDD**, the order of the variables is the same for every path through the graph.
- **ROBDD: Reduced ordered BDD**, given an ordering of the variables, the number of nodes in the diagram is minimized. ROBDD's form a canonical representation of boolean functions. Two functions are equal if and only if they have the same ROBDD [35].

Throughout this thesis we will use the term BDD to denote a reduced ordered binary decision diagram.

BDD's have some nice properties, for instance basic operations like intersection and union between BDD's can be implemented efficiently (in polynomial time in the size of the BDD), and for many practical instances the BDD for a given function is quite small [35].

Knuth [20] is an extensive reference on BDD's which also lists several uses in combinatorics.

Wesselink, Zantema [35] describe in detail how to encode Sokoban positions as boolean functions:

Each of the n non-wall squares numbered from 0 to $n-1$ is given a boolean variable b_i. b_i is true if and only if there is a stone on position i. The contents of wall squares never change, so these aren't included in the encoding. In addition, the man can be located at any of these n squares. $m = \lceil \log n \rceil$ boolean variables s_j are introduced to describe the bit pattern of the number corresponding to the square the man is located. In total, $n + \lceil \log n \rceil$ boolean variables are needed to describe an arbitrary position of a Sokoban problem.

To describe transitions from one position to another, a transition function $T(x, x')$ is defined, where x and x' represent a vector of boolean variables $(b_0, b_1, \ldots,$ $b_{n-1}, s_0, s_1, \ldots, s_{m-1})$. $T(x, x')$ is true is x' is a position which can be reached from x in one move.

The main idea here is to encode a subspace of the search space into one BDD. The transition function is possible to implement entirely using BDD's. Hence, it is possible to perform a state space search using only BDD's.

Wesselink, Zantema [35] used BDD's to solve puzzles from Sokoban using the representation described above. Using a reachability algorithm which resembles BFS combined with deadlock elimination (disallowing boxes on positions where they would be stuck), they managed to solve 4 of the 90 XSokoban puzzles. The most interesting result is the memory usage: Puzzle 6 has a state space of $3.8 \cdot 10^{10}$ reachable states, and they managed to solve the puzzle using a maximal BDD size of $2.1 \cdot 10^6$. Similar savings were reported on the other solved puzzles.

Edelkamp, Reffel [9] combined the ideas of BDD's and A* to create a new algorithm which they call BDDA*. They represented all states with the same f-value in one BDD. Using deadlock elimination and simple heuristics, they managed to solve one puzzle from the 90 XSokoban puzzles. Compared to BFS (which was also implemented using BDD's), their BDDA* algorithm needed only half the maximum number of states in the open list, but BDDA* ran much slower than BFS.

4.3 Atomix

Atomix is a puzzle game where atoms are moved around in a maze. All atoms have unit size. Atoms move one at a time, and atoms move in a straight line until their movement is hindered by a wall or another atom. The objective is to move the atoms in a certain configuration relative to each other, so that they together constitute a molecule. Figure 4.3 shows a screenshot from the computer game Atomix 2.13.4 for Linux.

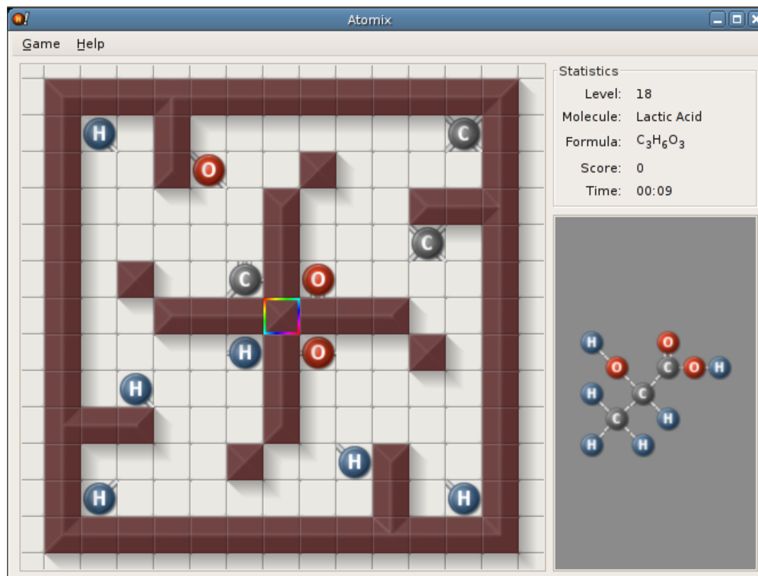

Figure 4.3: Level 18 from a computer implementation of Atomix. The main window is the game playing area, and the small window to the right shows the objective - the molecule to be assembled.

Atomix can be considered a generalization of $(nm - 1)$-puzzles. Every instance of $(nm - 1)$-puzzles is also an Atomix instance within a frame of size $n \times m$, where each numbered tile is an atom, and the resulting molecule consists of all atoms in "numerical" order.

Holzer and Schwoon [13] showed that Atomix is PSPACE-complete. They also provided a puzzle (based on an n-bit counter) with an exponentially long optimal solution.

Hüffner [16] used two different algorithms for solving Atomix puzzles: *A* without priority queue* and *Partial IDA**. Both approaches are explained below.

If the heuristic function $h(x)$ is *admissible* and *monotone*, A* can be implemented without a priority queue. They show that a state with the optimal f-value can be found in amortized time $O(\text{branching factor})$, compared to $O(\log |S|)$ (S is the set

of open nodes) which is the time needed to find the optimal f-value when the open set is implemented as a priority queue.

The variant of A* without a priority queue was implemented with two tables, a *state table* and a *hash table*. The state table will contain both the open and closed set. The states are appended to the state table as they are expanded, and each state has a bit indicating whether it is open or closed. States are never deleted from this table. The hash table contains pointers to the state table, and a linear displacement scheme is used to resolve hash collisions. To find an optimal open state, a linear scan is performed in the state table, starting from the first open node.

Their implementation of A* without priority queue was several times faster than a naïve implementation in C++ using STL `priority_queue` and `set`.

Partial IDA* is a way of reducing the memory usage of IDA* with transposition tables. They used *bitstate hashing*: Instead of storing the complete state, only a single bit corresponding to the hash value is set. However, hash collisions can occur, and this scheme doesn't handle collisions. Hence, there is a possibility that states on optimal paths can be pruned and therefore Partial IDA* loses admissibility. To decrease the probability of a hash collision, two bits were set and checked for each state (two arrays, one bit in each, using different hash functions, thus doubling memory usage). All solutions found with Partial IDA* were optimal.

With both algorithms, they used a heuristic function which they characterize as "rather uninformed". They manage to solve most of the puzzles from the Atomix computer game, with Partial IDA* being the better performing algorithm.

They also investigated backward search (from a goal state to the start state). They proved that the average number of children for states was the same for backward and forward search. This result was confirmed by experiments, as forward and backward search performed similarly.

4.4 Attempts at solving games with reasoning

There are many examples in existing literature describing how to solve two-player games using knowledge. For instance, research in chess-playing programs began at a very early stage. Usually minimax search is used (often in combination with enhancements as *alpha-beta pruning*. Since searching all the way to positions where the game is concluded, an evaluation function is used on the nodes where we decide to cut off searching.

Typically, domain-specific knowledge is employed in the evaluation function, and in form of opening books and endgame databases.

In this section, we will only look at examples where knowledge and reasoning is used in other ways than in the evaluation function of a minimax search.

4.4.1 Go

Houeland [15] used case-based reasoning in the move selection portion of his Go-playing program. For finding similar board positions, *influence maps* are compared, rather than the actual board positions. The influence map is an estimate of the areas of the board each player control. He used the UCT algorithm (a Monte Carlo tree search algorithm) for choosing which nodes to examine in the move tree. In addition, a CBR component is used for storing board positions from professional games, along with the move performed in the professional game. When the current board position is similar to a position in the case base, the recommended move for this position is given a bonus in the scoring used by the UCT algorithm.

However, while the program is able to identify strategically interesting areas of the board to play on, it is not able to find the actual strong tactical moves. Also, the program can overlook tactical threats in the beginning of the game if the opponent performs premature attacks.

4.4.2 Othello

Russell and Wefald [29] used meta-reasoning to play Othello. They used an algorithm called MGSS*, which they describe as "some kind of lookahead search using minimax backup." They use real utility instead of "quasi-utility" as the value of a step. They use meta-reasoning to find the computational action which has the highest utility.

Their program using the MGSS* algorithm manages to beat an Othello program using alpha-beta search, even though both programs use the same evaluation function. However, they show that reasoning over just one single step is untenable in general.

4.4.3 Connect-Four

Allis [2] used nine game-specific strategic rules to make an optimal game-playing program for the Connect-Four. These rules are similar to the rules an expert human player would use. Their approach can be seen as *rule-based*; if the board is of a certain type, a certain move should be done.

Chapter 5

Considering algorithms

This chapter contains the ideas we have for solving sliding-block puzzles, both at the object level (sections 5.1 and 5.2) and the meta-level (section 5.3), as well as the underlying data structures at all levels. Some of the ideas are our own, but most have a basis in existing literature.

5.1 Search

In this section, we will come up with ideas for improving the solver program from our specialization project. Some of the ideas are from the literature and applied to our domain, while some ideas are our own.

5.1.1 Search algorithms

In section 4.3 we described how Hüffner, et al [16] attempted to solve Atomix. They used A* with a priority queue and a non-admissible variant of IDA* (which they call Partial IDA*) which requires less memory. We will discuss how suitable those algorithms are for the search level of our domain.

A* without priority queue

In our program we can choose between three heuristics, and only one of our heuristics (the trivial one based on Manhattan distance) fulfills the requirements for using this algorithm: the heuristic must be *admissible* and *monotone*.

Unfortunately, this heuristic is very weak, and A* using this heuristic barely beats the performance of standard BFS and does not contribute to solving puzzles which

BFS cannot solve. In order for the heuristic to be able to guide our search efficiently, the h-value needs to be heavily weighted ($w > 10$ works well).

The asymptotic runtime of this algorithm is $O(\log(\text{branching factor}))$. Depending on the branching factor of the puzzle to be solved, the asymptotic runtime can be slower than standard A*.

We will not add this variant of A* to our program, because it only works with the weakest of our heuristics, and will therefore be of little use in our program.

5.1.2 Partial IDA*

IDA* did not perform very well on our domain. The main reasons are the difficulty of finding a good pathlimit (because of our weak heuristic function), and puzzles with long solution sequences having search spaces exhibiting exponential growth early in the search tree, followed by a reduction in the branching factor which breaks the advantage of iterative deepening; the last iteration is not necessarily the largest.

The improvement in partial IDA* over regular IDA* does not directly address the problems we encountered. The main benefit of partial IDA* is saving memory by not storing previously visited positions directly, but by using bitstate hashing instead: mapping positions to single bits (which also loses detection of hash collision). While we will be able to search slightly further with the memory savings, the fundamental problem of a weak heuristic will hinder this benefit. Therefore, we will not add partial IDA* to our program.

5.1.3 Enhancing A* with bitstate hashing

This idea is inspired by partial IDA* mentioned above. Our current implementation of A* uses a set to keeping track of the closed set (positions that are already evaluated).

If this set is implemented using bitstate hashing, we will end up with an algorithm similar to partial IDA*. It is natural to call it *partial A**. The savings in memory usage will be similar to IDA*. Likewise, we also get the disadvantages: the loss of ability to detect hash collisions. In the worst case, we can lose the actual solution.

It still remains to create hash functions that keep the probability of hash collisions low.

Because of the possibility that hash collisions can cause a solution to be lost, and the difficulty involved in designing a good hash function, we choose to not implement A* with bitstate hashing.

5.1.4 Solving an easier version

This idea is an original idea of ours, and has no basis in the literature study.

The idea is to first solve an easier version of the puzzle, and use the solution to aid in finding a solution to the original puzzle, using a plan with subgoals.

A puzzle can be made easier in several ways:

- Remove some blocks.
- Break down larger blocks into smaller blocks.
- Merge small blocks.

For the first two ways, the search space size can increase by a significant amount. The intention is to increase the mobility of the blocks. Also, the solution to the easy puzzle might not be transferrable to the real puzzle. Reaching the next subgoal can in some instances be unfeasible using our algorithms.

Earlier experiments show that the solution to the easy version can be very far from the solution for the real puzzle, in this case the solution lengths differ by factors up to 50 [32].

The last way (merge two blocks) might sound counterintuitive, but it has the effect of reducing the search space size. If this reduction is significant, it can make the puzzle feasible to solve, as long as the change does not render the puzzle impossible to solve. It is trivial to transfer the solution of the easy version to the real puzzle, as the merged blocks always move together. This way is expected to work well on huge puzzles containing many small blocks.

This is an interesting way of solving hard puzzles, and we will pick one of these methods and conduct an experiment to determine how well this idea works.

5.1.5 MGSS*

This algorithm (as implemented by Russell and Wefald [29]) works "greedily" by looking ahead one step at a time. Performing meta-reasoning on utility values based on looking one step ahead is clearly not feasible for our domain. Hence, we choose to not implement this algorithm.

5.2 Data structures

5.2.1 Binary decision diagram

In this section, we will outline a way to apply BDD's to solving sliding-block puzzles using BFS with no other improvements, using steps. Later in the section, we will

discuss some problems which can arise when attempting to use other algorithms, using moves and using pruning.

First of all, we need to represent puzzles as functions of multiple binary variables. We will use the puzzle in figure 5.1 as an example throughout this section.

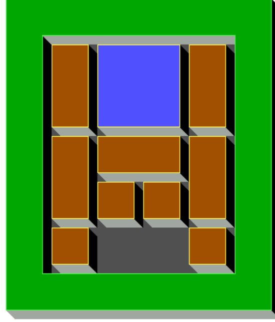

Figure 5.1: The sliding-block puzzle *Forget-me-not*

We choose to describe a puzzle using boolean variables in the following way. For each cell in the puzzle (20 in total for our example), we specify which block this cell holds. This can be an identifier referring to the upper left cell of a specific block type, an available cell, or an occupied cell (occupied by a non-upper left part of a block). In addition, two distinguishable blocks of the same size will require different ID's (for example if one of them is a master block).

The puzzle in figure 3.1 has the following parts:

ID	Block
0	space
1	occupied
2	1×1
3	2×1
4	1×2
5	2×2

Let n be the number of bits needed to encode the block ID. Let x and y be the size of the grid in the x and y axis, respectively. Then, for each cell, we need $n = \lceil \log_2 6 \rceil = 3$ variables to distinguish between the subblocks in the table above.

Let $b_{i,j,k}$ be the boolean variable representing the position (i, j) in the grid, and k be the k-th bit in the ID encoding.

The variables $b_{0,0,0}, b_{0,0,1}, \ldots, b_{0,0,n-1}, b_{0,1,0}, \ldots, b_{x-1,y-1,n-1}$ will then fully describe a position for any given puzzle. We will use b when we mean a fully described position.

For example, the position in figure 5.1 can be represented by the formula

$$\begin{aligned}
&\neg b_{0,0,0} \wedge \neg b_{0,0,1} \wedge b_{0,0,2} \wedge \\
&b_{1,0,0} \wedge \neg b_{1,0,1} \wedge b_{1,0,2} \wedge \\
&b_{2,0,0} \wedge \neg b_{2,0,1} \wedge \neg b_{2,0,2} \wedge \\
&\neg b_{3,0,0} \wedge \neg b_{3,0,1} \wedge b_{3,0,2} \wedge \\
&\vdots \\
&\neg b_{2,4,0} \wedge \neg b_{2,4,1} \wedge \neg b_{2,4,2} \wedge \\
&\neg b_{3,4,0} \wedge b_{3,4,1} \wedge \neg b_{3,4,2}.
\end{aligned}$$

A step can be described by the transition relation $T : \mathbb{B}^{x+y+n} \times \mathbb{B}^{x+y+n} \mapsto \mathbb{B}$ (where B is a boolean true, false), given by $T(b_{0,0,0}, b_{0,0,1}, \ldots, b_{x-1,y-1,n-1}, b'_{0,0,0}, b'_{0,0,1}, b'_{x-1,y-1,n-1})$ which is true if and only if b' is a neighbouring position of b. That is, b' can be reached from b by moving one block one step in any direction.

The relation T is a disjunction of all possible ways to move a block one step. For example, the formula

$$\begin{aligned}
&(\neg b_{0,0,0} \wedge b_{0,0,1} \wedge \neg b_{0,0,2} \wedge \\
&\neg b_{1,0,0} \wedge \neg b_{1,0,1} \wedge \neg b_{1,0,2}) \wedge \\
&(\neg b'_{0,0,0} \wedge \neg b'_{0,0,1} \wedge \neg b'_{0,0,2} \wedge \\
&\neg b'_{1,0,0} \wedge b'_{1,0,1} \wedge \neg b'_{1,0,2}) \wedge \\
&(b_{2,0,0} \leftrightarrow b'_{2,0,0}) \wedge (b_{2,0,1} \leftrightarrow b'_{2,0,1}) \wedge \\
&\ldots \wedge (b_{3,4,2} \leftrightarrow b'_{3,4,2})
\end{aligned}$$

describes a movement of a 1×1 block at the upper left one step to the right. In this formula, b denotes the position before movement, and b' the position after movement.

To perform a search for a goal position, we can start with the starting position b and repeatedly apply the transition relation T until we reach a goal position, or if we find no more positions.

An algorithm which accomplishes this (and is similar to BFS) is given as follows: [35]

ReachAlgorithm(I, F, T)
  $i \leftarrow 0$
  $R_{-1} \leftarrow \emptyset$
  $R_0 \leftarrow I$
  while $(R_i \cap F = \emptyset \wedge R_i \neq R_{i-1})$ **do**
    $R_{i+1} \leftarrow R_i \cup \{y | \exists x, x \in R_i \wedge T(x, y)\}$

$i \leftarrow i + 1$
end while
return $(R_i \cap F \neq 0, \{R_j\}_{j=0,1,\ldots,i})$

Here, I is the set of initial positions, F is the set of goal positions and T is the transition relation. R_i is the set of all positions that are reachable in up to i steps. The set

$$\{y|\exists x, x \in R_i \wedge T(x, y)\}$$

contains all positions y such that there is some position x in R_i having y as a neighbouring position. That is, the set contains all neighbouring positions of R_i.

For our domain, this algorithm can be changed slightly: We can redefine R_i so that it only holds positions that are reachable in *exactly* i steps. Then, we can exploit that we only need to keep the two last iterations in the BFS. R_{i+1} can be calculated as

$$R_{i+1} \leftarrow \{y|\exists x, x \in R_i \wedge T(x, y)\} \cap \overline{R_i} \cap \overline{R_{i-1}}.$$

After this operation, we can delete the set R_{i-1}.

However, using BDD's to solve for optimal moves instead of steps is far more difficult. In this case, one needs multiple transition relations T_i, one for each block, and they need to be applied repeatedly within each iteration until a block cannot be moved further during one iteration. To avoid the case of moving two different blocks of the same type during the same iteration, we need a way to distinguish the blocks from each other. Unfortunately, each block (even similar ones) would need different ID's, increasing the number of required variables. Also, in order to avoid storing essentially similar positions (positions where similar blocks are swapped), we need another boolean function to convert a position, such that essentially similar positions are converted to the same canonical representation.

Space value pruning can be implemented in the following way: After determining the set R_{i+1}, execute the statement $R_{i+1} \leftarrow R_{i+1} \cap S$ where S is the set of all positions having space value $\leq s$ for some cutoff s. This set can be precalculated before the search starts, as S is unchanged during the search, and across different instances (shufflings) of the same puzzle. This set consists of up to $\binom{xy}{k}$ disjunctions of formulas for positions of all the spaces in the puzzle, where k is the number of spaces in the puzzle and x, y are the dimensions of the interior of the puzzle. Even though the set might be feasible to *store* using BDD's, building the set requires exponential time, which is infeasible for large puzzles with many spaces.

Implementing other heuristics and prunings using BDD's is an open problem.

We choose to not implement the above BDD approach as a part of the thesis. This choice is done on the basis of the following: The implementation is expected to be rather complex and involving, even with the use of a BDD library, as care must be taken to get all the details correct. In addition, the BDD approach is incompatible with most of our improvements. We described above how to implement a BFS algorithm without enhancements using BDD's. With BDD, we would not be able

to use improvements like A* with our non-admissible heuristic, and pruning based on resting blocks. Although we described a way above to implement space value pruning, it involves the disjunction of an exponential number of formulas in the size of the puzzle, and is therefore infeasible for larger puzzles. The prunings are essential for solving the harder and larger puzzles, as the results from the specialization project showed.

5.3 Reasoning

The problem we want to solve at the meta-level is the following.

The first step: Given an instance of a sliding-block puzzle, return a set of parameters such that our solver program is likely to find a solution using these parameters. Then, based on the results from running the solver program, determine new sets of parameters to use for further runs of the solver program. The last step will be repeated multiple times and is called the *meta-control cycle*.

In this section we will discuss various reasoning methods, and discuss out how well they will be able to solve the above problem. The goal of this section is to choose one or more methods to implement and test.

Some aspects of this problem include:

- Analyzing the starting position of the puzzle and make an intermediate representation which forms a better basis for reasoning.
- From this basis, find a set of parameters which will allow us to solve a given puzzle.
- Finding a good balance of resource usage between searching and reasoning about the search.
- Determine whether to follow through with the current strategy, make changes, or to cancel.
- Monitor the resource usage and progress of the underlying algorithm, manage the information gathered from the monitoring and making use of this knowledge.

Luger [27] (p.281) mentions some guidelines which can be followed to determine whether a problem is appropriate to solve using knowledge from human experts.

(1) **The need for the solution justifies the cost and effort of building an expert system.**

(2) **Human expertise is not available in all situations where it is needed.**

(3) **The problem may be solved using symbolic reasoning.**

(4) **The problem domain is well structured and does not require common sense reasoning.**

(5) **The problem may not be solved using traditional computing methods.**

(6) **Cooperative and articulate experts exist,** meaning that the knowledge in an expert system comes from human experts in the field and that they are willing and able to share knowledge.

(7) **The problem is of proper size and scope.**

The availability of human experts is a requirement. In this thesis, we have access to the author, who is a decent Bricks player and can be considered a human expert.

Point (1) is irrelevant to us, we have already chosen the problem we want to solve. Regarding point (2), our main goal is to be able to solve puzzles without human expertise, but we retain the option to let a human expert aid the computer in solving the puzzles to a certain degree.

The problem of finding the correct parameters to arbitrary instances with symbolic reasoning (point 3) is difficult. We expect to be able to solve it for instances of easy and medium difficulty (the easier an instance is, the larger the set of working parameter sets is), but we don't expect to solve it for very hard instances. Some instances are beyond our reach no matter which parameters we set, because of lacking capabilities in our solver program.

About point (4): The problem domain can be considered very well structured, as the object level is a single-agent search problem with perfect information. In addition, the interface to the solver program invoked at the meta-level is also well-defined and specified. For a human to excel at solving puzzles, he needs extensive domain knowledge and experience in solving easier puzzles, or puzzles from similar domains. The person needs to come up with several plans, consider which ones are most likely to work, and execute the plan. Such knowledge is probably too narrow to be considered within the realm of common sense reasoning [7], which typically involves well-known facts that an ordinary person is expected to know.

About point (5), since our domain is PSPACE-complete, arbitrary instances are not practically solvable using traditional computing methods. However, we are focusing on instances that are solvable by humans (although these instances can be extremely hard for humans). By using domain-specific knowledge, we believe it is possible for a computer to solve every puzzle that a human can solve. Results from other PSPACE-complete domains such as Sokoban [33] suggest that one can solve a large portion of instances from computer games.

Point (6) is no obstacle for us, as our available domain expert is willing to share his knowledge.

Regarding point (7), we believe the problem is of proper size and scope. There exist many problem instances that are possible to solve using reasoning, but some

of them can be difficult.

In order to reason about the initial position, we need to somehow extract information from this position which forms a basis for deciding how to control the object level.

The problem of reasoning on the initial position of a puzzle and the limit of how human expertise can be used, makes this a very challenging problem for reasoning methods.

5.3.1 Rule-based systems

It's very hard to do reasoning on the starting position of an arbitrary instance. Therefore, we should preprocess the starting position and identify features, so that we can do reasoning on the features instead.

We recall that a rule-based system contains rules like these:

If
mobility is high, and
estimated search space size $\geq 10^{11}$, and
number of spaces ≤ 4
then
use algorithm A* with heuristic = 3 and weight = 20.

One potential problem with this approach, is that there can be a lot of rules, unless we augment the rules and add expressions. For example:

If
mobility is high, and
estimated search space size $\geq 10^{11}$
then
use algorithm A* with heuristic 3 and weight = $\frac{80}{\texttt{num_spaces}}$.

There are two hard problems with this approach:

- Given a puzzle, find its features. This problem is common for several reasoning approaches that need to reason on the features of a puzzle.
- Making a mapping from the feature space to the parameter space, such that the parameters will allow us to solve as many instances as possible. Once we have this mapping, creating the actual rules should be straightforward.

This is a promising approach, given that we manage to make such a mapping.

5.3.2 Model-based reasoning

In model-based reasoning, one attempts to get a deeper understanding by simulating the problem domain in greater detail, in contrast to trying to infer from

symptoms and behaviour that can be observed directly. For example, in the domain of finding a medical diagnosis, one could try to detect the causes of a disease (presence of infecting agents) instead of basing a diagnosis on observable phenomena as headache, nausea or high fever [27]. Model-based reasoning typically tries to model some aspect of the physical world.

Our domain is a domain with perfect information, and is a problem rooted in combinatorics rather than the real world. Luger [27] only mentions difficult problem domains rooted in the real world where some kind of modelling need to be done. Our domain is unusual in this sense, as we use deterministic algorithms on a domain with perfect information. Model-based reasoning seems to be a bad fit for our domain, so we choose to not go with this approach.

5.3.3 CBR

We recall that case-based reasoning (CBR) is a way of reasoning based on the reuse of past cases containing solutions. When given a problem, a typical CBR system searches through its case base and retrieves the case which is most similar to the problem given. Then, the solution to the retrieved case is adapted to the current problem, and then applied. If we managed to solve the new problem, a new case can be added to the case base, consisting of the new problem and its solution.

In order to solve our problem with CBR, we need to define a case. If we let the case consist of the problem instance itself, and retrieve past cases based on how similar the puzzles look, we are going to get some bad results. It's easy to construct puzzles which differ in only one cell (for instance, replacing a space by one 1×1 block where the original puzzle is trivially solvable, while the modified puzzle is unsolvable. One such example is shown in figure 5.2. The puzzle to the left is easy to solve, while the puzzle to the right is impossible to solve, despite the minimal change of one added 1×1 block.

Therefore, a case must contain additional attributes that are easily compared to other cases. Such attributes are called *features*, and a collection of features belonging to one case is called a *feature vector*. A feature vector contains features containing values, like `mobility=high` and `packing_problem=true`.

The problem of finding features given a puzzle is the same as for rule-based systems discussed in section 5.3.1. The performance of a CBR approach depends on the quality of the feature identification. With a good way of identifying features, we believe that CBR can be a good method for our problem domain.

5.3.4 Other methods

We have decided to not consider other methods, including machine learning, subsymbolic methods and Bayesian belief networks.

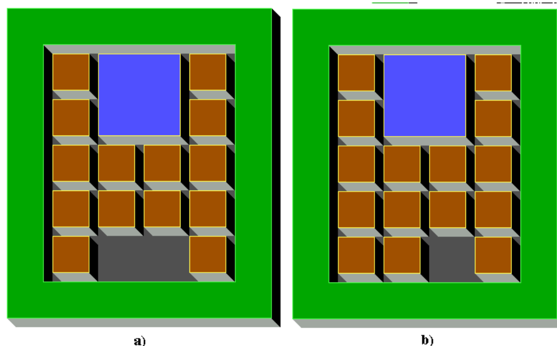

Figure 5.2: a) shows an easy puzzle, while the similar-looking puzzle in b) is unsolvable. The objective is to move the master block to the center of the bottom row.

Even though evolutionary algorithms have been used on problems with ill-behaving search spaces like our parameter space (for instance, the evolution of circuit designs [12], pages 62-85), it is infeasible for our domain because of the computational cost needed for evaluating the fitness of each individual in the evolutionary algorithm.

5.4 Other issues regarding metareasoning

In a case-based or rule-based approach, after analyzing a problem instance one will usually end up with a specific set of parameters to apply at the object level (search). Results from the specialization project [32] show that small changes in parameters can result in an instance not being solved. It is likely that the meta-reasoning algorithm will sometimes find parameters that are slightly wrong with regard to solving the puzzle. It is therefore a good idea for the meta-reasoner to execute the search at the object level with slightly different parameters.

In this section, we will discuss ideas on how to explore similar parameters.

- If the meta-reasoner finds several sets of parameters for a given instance, and the sets of parameters are sufficiently different, we will try all these parameter sets at the object level. This can be all matches q such that $d(p_0, q) < r$ for some r (where p_0 is our best match), or simply the k best matches.
- For each parameter set we try (and within a reasonable usage of resources), try out different parameters on a "breadth-first" basis.

A little comment on why two sets of parameters should be sufficiently different. The idea is that if two sets of parameters are "close" ($d(p, q) < r_1$ for some small r_1), the following search in the parameter space from one set of parameters will

eventually discover the other set, resulting on overlapping work. (Other approaches are possible though, like doing BFS with several starting nodes.)

All runs at the object level will run for a limited amount of time with a limited amount of resources, for instance by not allowing to examine more than q positions for some sensible value of q (somewhere around $10^7 - 10^9$).

One way to implement the BFS search on parameters is the following.

Let the initial set of parameters $P = (p_1, p_2, p_3, \ldots, p_n)$ be defined as the starting point in our "breadth-first" search. Then, let one reasonable change in one parameter have a unit cost. The scale of such a change will vary between parameters. For instance, changing the space value by 1 (in either direction) has unit cost. A parameter such as the maximal size of the queue before we drop all bad positions (which is in the order of the total memory size we wish to use, divided by a constant proportional to the memory usage of one position) will naturally change by much more, likely in the order of 10^5 or more.

For each problem we wish to solve, we can then let the program run for as long as we desire (given our resources).

We can guide this search to a limited degree. For instance, if the meta-reasoner determines that a certain parameters definitely should have some value, we will not allow the parameter search to explore "forbidden" parameter settings.

We can also do other approaches than search: We can define some kind of metric (a distance measure function) on our parameter space. For two parameter sets p and q, let $d(p, q)$ be the "distance" between the parameter sets. Let the distance be defined somewhat similar to the BFS case above.

We can use rejection sampling to find sets of parameters to try. Randomly generate a parameter q, and only do further processing (search) if $d(q, p_0) \leq r$ for some radius r which could increase with time (here, p_0 is the initial set of parameters as found by the meta-reasoner).

The approach we have described in this section can, in conjunction with CBR, be used to achieve a form of learning. For instance, we could experience a situation where we want to solve a specific puzzle. The meta-reasoner suggests a set of parameters which doesn't solve the puzzle, but the BFS-search/rejection sampling method manages to find good parameters.

We will implement an approach similar to the rejection sampling approach mentioned above. We will do a randomized proximity search and accept sets of parameters (denote it p) if they are not identical to any of the parameters already tried, and if there exists a previously tested set of parameters q such that the distance $d(p, q)$ is less than some chosen radius r_1.

5.5 What we will implement

Here is a quick summary of what we will implement in our system.

We will implement a meta-reasoning system that tries to find a set of parameters to solve a given puzzle. The system will be based on CBR. The system will first try to find the best cases from a case base. If none of the cases manage to solve the puzzle, the meta-reasoner will search for parameters as described in section 5.4.

In addition, we will enhance the underlying solver with the following features:

- Any needed functionality for interfacing with the meta-reasoner: Communication and feature extraction from a given puzzle.
- An option to output the solution if the puzzle is solved.
- Optimized memory usage (permutation rank). This is one of the improvements from the *future work* section of the specialization project.

In order to make the parameter space more well-behaving and nice, we decided that the meta-reasoner will not use all the options available in the solver program. Here are the options we will not use:

- **The IDA* algorithm**: The results from the specialization project showed that this algorithm was dominated by the A* algorithm for all our test cases.
- **Never move the master block away from the goal**: The results from the specialization project showed that many puzzles became unsolvable when this option was turned on. In addition, the existence of this parameter was not sufficient to solve new puzzles. Also, we wish to keep just one parameter that can make puzzles unsolvable (space value pruning), to make it easier to reason over why the solver couldn't solve a puzzle.
- **Subgoals**: This option is not specified via parameters to the solver program, but is done by adding multiple goals (waypoints) in the puzzle input file. Finding subgoals is a hard problem on its own, and it would take considerable time to choose suitable subgoals for the hard puzzles, or to automate this task.
- **Delete inferior states when memory is exhausted, and resume search**: In order to be able to run all tests in time, we will enforce a limit on the number of positions the search is allowed to before the meta-reasoner resumes control. Hence, searches will be aborted before memory is exhausted and this option serves no purpose.
- **Disallow movement of certain blocks**: This option is not specified via parameters to the solver program, but is done by changing the puzzle input file. Reasoning about which blocks to lock in place is a hard problem in its own right, and thus we will not include this option in this thesis.

5.5.1 CBR features

We have chosen to only include quantitative features as part of the CBR system, and to drop qualitative features. The reason is that qualitative features are very complex to extract from a puzzle.

We believe that a reasoning system using only qualitative features can perform well, as long as there are sufficiently many features and that their values vary somewhat independently of each other.

We have decided to include the following features:

- Estimated (upper bound) of the search space size
- Estimated branching factor
- Block shape niceness
- Average block size
- Number of spaces
- Puzzle size (total number of cells in the puzzle area)

It was decided to let each feature contain a value range $[a, b]$ where a and b are floating-point values. The reason for this is to take into account the uncertainty of some of the values.

Chapter 6

System design

In this chapter, we will present the high-level design and description of our resulting system.

6.1 System overview

The system consists of two separate programs: The meta-reasoning program and the solver program. The solver program takes a puzzle description as input, and different parameters can be set to control the search for a solution. The programs are separate executable files, and are written in C++.

Figure 6.1 shows a screenshot of the meta-reasoning program in action.

6.2 Solver

The solver was written in the specialization project. The following additions were done in this thesis:

- Added a means for the solver program to communicate with the meta-reasoning program.
- Support for new block types: disconnected block and slider.
- Added a new, more memory efficient way of representing a game position (permutation rank).
- Added an option to display the solution sequence of a solved puzzle.
- Added an option for analyzing a puzzle and gather quantitative features.

```
Command Prompt - reason casebase.txt b04.txt
read block_size 2.000000 2.000000 (4)
read num_of_spaces 6.000000 6.000000 (5)
read puzzle_size 56.000000 56.000000 (6)
read features.
our case:
- search_space_size [38209180095646556000.000000 38209180095646556000.000000]
- search_space_size_type [20.000000 20.000000]
- branching_factor [9.739910 9.739910]
- regular_shapes [0.960000 0.960000]
- block_size [2.000000 2.000000]
- num_of_spaces [6.000000 6.000000]
- puzzle_size [56.000000 56.000000]


Case 0: 52.806753 (Puzzle 01 easy - just search)
Case 1: 56.867500 (Puzzle 02 forget-me-not)
Case 2: 0.005731 (puzzle 04 american pie)
Case 3: 17.629680 (puzzle 05 still easy)
Case 4: 18.815128 (puzzle 06 rose)
Case 5: 27.660535 (puzzle 11 isolation)
Case 6: 22.471228 (puzzle 12 hyperion)
Case 7: 14.168138 (puzzle 13 san)
Case 8: 19.069953 (puzzle 16 triathlon)
Case 9: 19152.453186 (puzzle 08 huge and open)
Case 10: 9578.172866 (puzzle 29 huge 1x1)
Case 11: 9.243983 (puzzle 17 magnolia)
Case 12: 88.792210 (puzzle 18 turtle)
Case 13: 10.734387 (puzzle 24 warmup)
Case 14: 18.084209 (puzzle 25 get ready)
Case 15: 29.172123 (puzzle 26 climb game 15d)
Case 16: 42.315298 (puzzle 28 the devil's nightcap)
0: 0.005731 puzzle 04 american pie
1: 9.243983 puzzle 17 magnolia
2: 10.734387 puzzle 24 warmup
3: 14.168138 puzzle 13 san
4: 17.629680 puzzle 05 still easy
5: 18.084209 puzzle 25 get ready
6: 18.815128 puzzle 06 rose
7: 19.069953 puzzle 16 triathlon
8: 22.471228 puzzle 12 hyperion
9: 27.660535 puzzle 11 isolation
10: 29.172123 puzzle 26 climb game 15d
11: 42.315298 puzzle 28 the devil's nightcap
12: 52.806753 Puzzle 01 easy - just search
13: 56.867500 Puzzle 02 forget-me-not
14: 88.792210 puzzle 18 turtle
15: 9578.172866 puzzle 29 huge 1x1
16: 19152.453186 puzzle 08 huge and open
```

Figure 6.1: Screenshot of the meta-reasoning program

6.3 Meta-reasoner

The meta-reasoner is a stand-alone program that launches the solver. The program uses CBR to find suitable parameters to run the solver program with. The meta-reasoner was written in its entirety as part of this thesis.

When the meta-reasoner is called, the puzzle is analyzed and quantitative features are extracted. This is done via a call to the solver program.

After the features are extracted, the case base is searched for previously solved puzzles with similar features. The similarity between the puzzle to be solved and every puzzle in the case base is calculated, and then sorted in increasing order.

Next, the solver is called with the parameters from the most similar case. If this set of parameters manages to solve the puzzle, we are done. If not, the second best case is applied. Cases from the sorted list of cases are applied until we have tried the three most similar ones, or until the next case has a similarity value more than three[1] times of the similarity of the best case. For each run, the solver has a limit on the number of positions examined. If this limit is exceeded, the solver cancels the search and returns the control to the meta-reasoner.

Calling the solver has three outcomes: Either the puzzle is solved, we exceeded the limit of positions to examine, or a solution was not found with the current parameters. The last result can only happen if the space value cutoff is too low. When this happens, the meta-reasoner will remember this, and future space value cutoffs will always be selected to be higher than the highest known cutoff which leads to no solution being found.

At this stage, we have tried to apply all the good cases to the puzzle. From now, a randomized search is performed in the parameter space. A random set of parameters is generated, and a search is performed using these parameters only if:

- The set of parameters must not be too close to any set of parameters that have been used for a previous search.
- There must exist a previously used set of parameters which is not too far away from the newly generated set of parameters.

To determine if two sets of parameters are "close" or not, we have designed a function which returns the distance between two sets of parameters. It is explained in detail in section 7.2.2 in the next chapter.

During the randomized search, the same limit for the number of positions examined applies to the solver. The randomizes search continues until 25 attempts to solve the puzzle has been made.

After 25 attempts without finding a solution, a final attempt to solve the puzzle is done, without a limit on the number of positions to examine. The parameters chosen for the final attempt are taken from the best run out of the previous 25,

[1]The number 3 was chosen arbitrarily.

where best is defined as the shortest distance from the master block to the goal. If several runs had the same shortest distance to the goal, the parameters from the run where this distance was reached in the fewest number of positions examined is chosen.

6.3.1 CBR

The CBR component of the program has a case base containing previously solved puzzles. A case consists of the following features:

- Estimate (upper bound) of the search space size
- Branching factor
- Block shape niceness - the number of rectangular blocks divided by the total number of blocks
- Average block size
- Number of spaces
- Puzzle size (total number of cells in the puzzle area)

In addition, each case stores the parameters used for solving the puzzle. We have chosen to have *numerical values* only associated with each feature, represented as a range of floating point numbers (lower and upper bound). The reason for having a range is to account for uncertainty in the values.

The case base itself will consist of the 15 puzzles that was solved during the specialization project, along with the parameter sets that solve them. For each puzzle that we managed to solve with multiple parameter sets, we chose the parameter set that solves the puzzle with the least search effort, measured in number of positions examined before a solution is found.

Here is a brief summary of what happens during the 4 CBR phases:

Retrieve

When the meta-reasoner is given a new puzzle to solve, the case base is searched for the cases that are most similar to the given puzzle. The three most similar cases are always retrieved, in increasing order of similarity. Further cases are also retrieved, if their similarity is less than three times the similarity of the best case.

Reuse

For each case that is retrieved, the solution for this case is applied to the current puzzle to solve. The solver is run, which results in either finding a solution, showing

that no solution exists with the settings used, or we exceed the limit of how many positions can examine. This limit is set to 5 million positions.

Revise

If the case wasn't solved, the meta-control cycle begins. Run the second, third, $\cdots, n$-th best case. Run the cases again, and slightly alter one/some of the parameters. After 25 attempts to run the solver, a final run will be performed with the best parameters found so far. This run is not subject to any limitations of number of positions examined. In the first 25 runs, the solver exits when it exceeds a certain number of examined positions.

Retain

Solutions found by the meta-reasoner are not retained. We wish to ensure that each test case gets equal conditions. Otherwise, the last cases run could have a higher chance of being solved, since they get the benefit of a larger case base.

Chapter 7

Implementation

In this chapter, we will present more in-depth details about the implementation of our system. Our system consists of two main components, the *solver* and the *meta-reasoner*, that are two separate executable files.

7.1 Solver

Several additions were made to the solver program, which was originally programmed during the specialization project. These additions will be explained in detail in this section.

7.1.1 Communication between the meta-reasoner and the solver

When the meta-reasoning program calls the solver, the result needs to be passed back to the meta-reasoner.

When the meta-reasoner invokes the solver, command line parameters are used to specify what the solver should do. This includes parameters controlling the search algorithms. In addition, when the solver is invoked from the meta-reasoner, the number of positions the solver is allowed to perform is set. The solver will return when this limit is exceeded.

The results from the solver are passed back to the meta-reasoner via temporary text files. A text file is written to disk just prior to exiting the solver, and the file is read by the meta-reasoner as soon as it resumes control.

The following information is passed from the meta-reasoner to the solver via command-line parameters:

- Parameters controlling the search (choice of algorithm, heuristic function and other options).
- Choice of trying to solve the puzzles or gather features to be used by the CBR component.

The following information is passed from the solver to the meta-reasoner via text files:

- The result of the search (solved, no solution found, no result), including additional information: Best distance from master block to goal found and the lowest number of positions examined this happened, number of steps needed to solve the puzzle (if it was solved), and the number of positions examined.
- Information about whether the solver exited gracefully, or if it crashed or was cancelled by the user.
- The features extracted from the puzzle.

Below is an example of the text file generated from the solver after extracting features from a puzzle.

```
search_space_size 25955.000000 25955.000000
search_space_size_type 0.000000 0.000000
branching_factor 3.232364 3.232364
regular_shapes 1.000000 1.000000
block_size 1.800000 1.800000
num_of_spaces 2.000000 2.000000
puzzle_size 20.000000 20.000000
```

7.1.2 Display of solution sequence

The solver is now able to output the solution to a puzzle. This functionality is activated via a command line option to the solver.

This was implemented in the following way: For each position examined, it is linked to its parent position. This is done using a map data structure, where pairs of positions are stored. Each pair contains a position and its parent position. The `map` container from C++ STL was used to implement this functionality.

When a puzzle is solved, the final position is looked up in the data structure in order to find the second last position. This procedure is repeated until we reach a position with no parent. This is the starting position. The solution sequence is the positions found using the procedure above, in reverse order.

The disadvantage with this method is that in addition to keeping a queue of positions to examine, and a set of positions previously examined, a map linking positions to parent positions need to be maintained. This roughly doubles the memory usage of the solver. If there is a puzzle which was barely solvable without this

option, it is likely that the solver would run out of memory with this option on before finding a solution.

7.1.3 New block types

In order to add the two block types (disconnected blocks and sliders), the input format was changed. In the old format, each subcell of a block was assumed to be connected, and BFS was used to find all the connected cells of one block. In the new format, there is a separate section which describes the shape of each block and specifies movement restriction. Figure 7.1 shows a puzzle, along with its definition file in both formats. The `@` character denotes the master block.

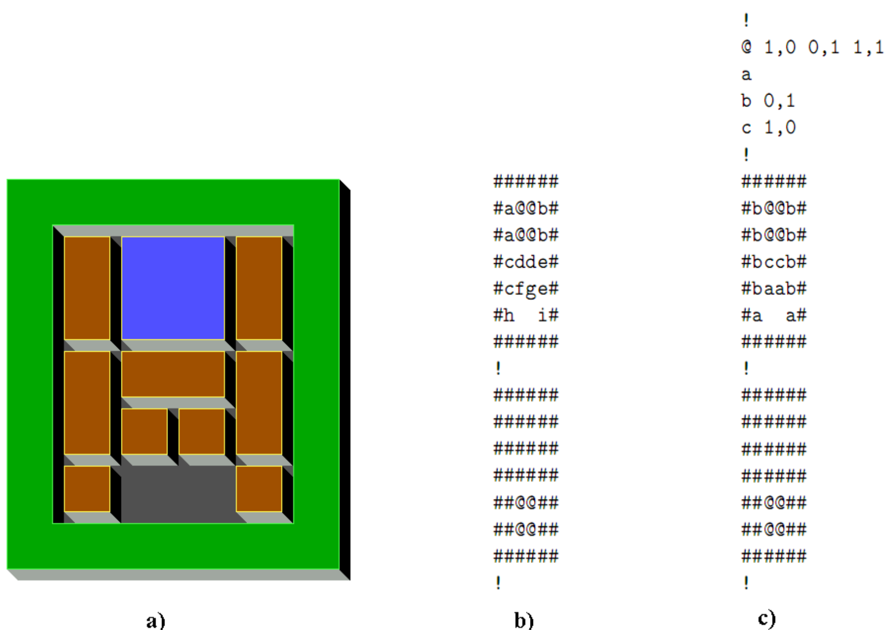

Figure 7.1: a) shows a puzzle, b) shows how it is defined in the old format, and c) shows how it is defined in the new format.

Figure 7.2 shows two puzzles, one with disconnected blocks, and one with sliders. Figure 7.3 shows how they are defined in the new input format.

Consider the puzzle defined in figure 7.3 b). The puzzle definition consists of several sections. Each section is separated by a line containing the character `!`. The first section is a header defining the block shapes. The second section defines the starting position of the puzzle. Successive sections define goal positions. The header where the blocks are defined are given by the following section:

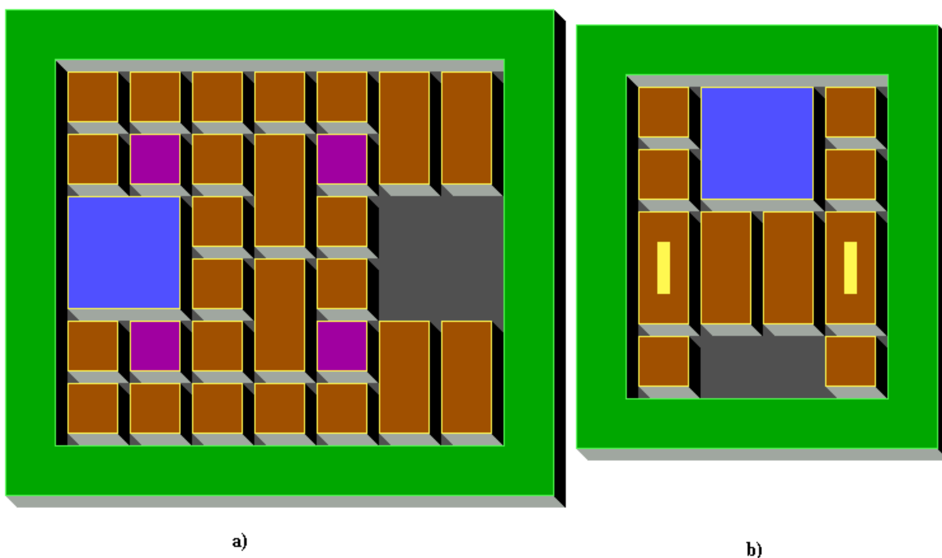

Figure 7.2: The puzzle to the left contains a disconnected block, and the puzzle to the right contains two vertical sliders.

```
@ 1,0 0,1 1,1
a
b 0,1
v | 0,1
```

Each line defines a different kind of block. Each block has a cell at (0,0), in addition to the coordinates specified. The first line defines a block with identifier `@` with cells at (0,0), (1,0), (0,1) and (1,1), a 2×2 block. The second line defines a 1×1 block, the third line defines a 1×2 block (a cell at (0,1) in addition to (0,0)) and the fourth line defines a 1×2 block which is a vertical slider. The `|` character, which immediately follows the identifier, defines the block as a vertical slider. It is possible to define a horizontal slider by using `=`. A block cannot be a vertical and a horizontal slider at the same time. The master block (`@`) must be the first block defined.

The second section defines the starting position:

```
######
#a@@a#
#a@@a#
#vbbv#
#vbbv#
#a  a#
######
```

```
!
@ 1,0 0,1 1,1
a
b 0,1
c 3,0 0,3 3,3
!
#########
#aaaaabb#
#acabcbb#
#@@aba  #
#@@aba  #
#acabcbb#
#aaaaabb#
#########
!
#########
#########
#########
######@@#
######@@#
#########
#########
#########
!
```

a)

```
!
@ 1,0 0,1 1,1
a
b 0,1
v | 0,1
!
######
#a@@a#
#a@@a#
#vbbv#
#vbbv#
#a  a#
######
!
######
######
######
######
##@@##
##@@##
######
!
```

b)

Figure 7.3: The puzzle to the left contains a disconnected block, and the puzzle to the right contains two vertical sliders.

For each block in the puzzle (see figure 7.2 a)), there is at least one corresponding block in the puzzle definition. All cells of a block are filled in with the block's identifier. For example, the master block (`@`) is a 2×2 block, which is defined by having four `@` characters near the top of the position arranged in a 2×2 square.

In subsequent sections, not all the blocks in the puzzle need to be specified. Often, only the master block is specified. Let the sections following the starting position be denoted $p_1, p_2, \ldots, p_n$. The solver will start from the starting position, and try to reach position p_1. When this is accomplished, the solver will start a new search, with the position that reached p_1 as a new starting position. The goal of this search is to reach position p_2. The solver proceeds like this, until position p_n is reached or the solver fails to reach any of the positions. We call the positions $p_1, p_2, \ldots, p_{n-1}$ *subgoals*, and p_n the *goal*, the final goal position of the puzzle. However, for our main testing, only one goal is specified.

Both the old and new formats are supported by the solver. The first line of the new format will always contain the `!` character alone, while in the old format, the first line will contain the first line of the puzzle's starting position. This line will never contain the `!` character. By checking the first line, the solver can determine which format is used.

7.1.4 Storing positions more efficiently with permutation rank

In addition to encoding each position with Huffman coding (described in detail in our specialization project report [32]), we added another representation: the lexicographic permutation rank. A position can be seen as a permuted sequence of the blocks that occur in the puzzle. For instance, the puzzle defined in figure 7.3 a) contains one master block (`@`), six 1×1 blocks (`a`), two 1×2 blocks (`b`), two 1×2 sliders (`v`) and two blank cells (space, shown as `_`). When we scan the starting position from top to bottom, and for each row, from left to right and put new blocks in a list as we encounter them, we get the list `a@aaavbbva__a`. Let this list be the *string* of the position. The position can be reconstructed by traversing the string and putting each block at the topmost (leftmost) not yet occupied cell in the new board (this procedure is described in section 2.6.1 in [32]). For each possible position, there is a permutation of the list such that the position can be reconstructed.

Consider an arbitrary puzzle. Let n be the number of different block types (*including* space). For the example above, $n = 5$. Let $a_1, \cdots, a_n$ denote the amount of each block present in the puzzle. Then, there are a total of

$$\binom{a_1 + a_2 + \ldots + a_n}{a_1, a_2, \ldots, a_n} = \frac{(a_1 + a_2 + \ldots + a_n)!}{a_1!a_2!\ldots a_n!}$$

ways to rearrange the puzzle's string.

Consider the list of every permutation of the string from a puzzle. Sort all permutations in lexicographic order. Given an arbitrary position of a puzzle, its string can be found in the sorted list of all strings. Given that the sorted list is 0-indexed, the rank of the string in this list is the *permutation rank* of the position. The rank is an integer between 0 and $\binom{a_1+a_2+\ldots+a_n}{a_1,a_2,\ldots,a_n} - 1$.

We chose to only implement permutation rank for 64-bits unsigned integers. The reason is that preliminary tests showed that calculating the permutation rank was about 30% slower than calculating the Huffman code for a position. The time needed for calculating a multi-word permutation rank would be proportional to the number of words needed to hold the number.

The permutation rank of a string is calculated using the following recursion: Count the number of permutations having a letter with a lower value than the first letter in the string. This is done by looping through each letter with a lower value, removing it temporarily, giving a new multiset of letters. Then, the multinomial coefficient corresponding to the remaining set of letters is evaluated, and added to the partial result. Finally, remove the first letter and repeat the entire procedure for the rest of the string, until the string is empty.

In order to make the calculation of the multinomial coefficient

$$\binom{a_1+a_2+\ldots+a_n}{a_1,a_2,\ldots,a_n} = \frac{(a_1+a_2+\ldots+a_n)!}{a_1!a_2!\ldots a_n!}.$$

faster and to avoid overflow in intermediate calculations, the identity [19]

$$\binom{a_1+a_2+\ldots+a_n}{a_1,a_2,\ldots,a_n} = \binom{a_1+a_2}{a_1}\binom{a_1+a_2+a_3}{a_1+a_2}\cdots\binom{a_1+a_2+\ldots+a_n}{a_1+a_2+\ldots+a_{n-1}}$$

was used, along with a precalculated table of binomial coefficients.

The *unrank* operation (given a permutation rank, return the position string) is basically the algorithm above in reverse. For each character position, find the letter with the lowest value such that the number of permutations having a first letter lower than the letter under consideration is smaller than or equal to the given permutation rank. Remove this letter from the set of remaining letters and repeat the procedure.

The reason for implementing permutation rank was to reduce the memory requirements for the program, so that we could examine more positions in a search before running out of memory, increasing the chance of solving more puzzles.

7.1.5 Feature extraction

Feature extraction is invoked by calling the solver with a special command, along with a file name specifying the puzzle from which to extract features. The solver analyzes the puzzle, writes the features to a temporary text file and returns program control to the meta-reasoner.

The average block size, number of spaces, puzzle size and block shape niceness (number of rectangular blocks divided by the total number of blocks) are calculated immediately from the starting position of the puzzle. In order to find the branching factor, a BFS is performed from the starting position. The BFS is cancelled when the examined positions exceed 1 megabyte in storage, or when the entire search space has been examined. The estimated branching factor is the average number of neighbours over all positions examined.

The estimate of the search space size is calculated in one of three ways:

- The exact size is returned if the BFS manages to search the entire search space.
- If it is possible to find the *packing bound* using less than 1 million recursive calls in the packing bound calculation algorithm, this bound is returned.
- Otherwise, the combinatorial bound is returned.

The type of search space size (exact, packing bound, combinatorial bound) is returned in a field of its own and receives a value based on the following table:

Exact size	0
Packing bound	10
Combinatorial bound	20

The values 0, 10 and 20 were chosen arbitrarily, but the interval between them was chosen to be equal by design. Formula 7.1 (which is used in the similarity calculation, see section 7.2.2) was in turn designed so that the search space size gets an appropriate weight in the similarity function.

The *packing bound* is the number of ways it is possible to pack the blocks in the puzzle frame. The *combinatorial bound* is the number of permutations of the multiset consisting of all the block's puzzles and 1×1 spaces. The calculation of these upper bounds were part of the specialization project, and are described in the report [32].

7.2 Meta-reasoner

The meta-reasoner is a separate program which interacts with the solver. When we want to solve a puzzle, we call the meta-reasoner with two parameters: The file name of the case base, and the file name of the puzzle to be solved. The format of the case base is specified in section 7.2.1. There are no additional options apart from the two file names.

Since the case base is small, no advanced data structures are used to store it in memory. The case base is stored in a C++ STL `vector`. Given a puzzle to solve, a simple loop through all the cases is used for finding the most similar case.

During the last stage of the meta-reasoner (the proximity search), a STL `vector` is used to hold the list of previously visited parameter sets. For each newly generated position in the search, a linear search is performed through the entire list to check if the generated position isn't visited before, and that it is close enough to a position in the list.

7.2.1 Case base

The case base is stored in a text file. The text file is divided into two parts: The definition of the fields, and the cases. Each case consists of two sections, the features and the solution.

The text file contains the following sections:

- Field definition
- Case 1's features
- Case 1's solution attributes
- Case 2's features
- Case 2's solution attributes
- ...
- Case n's features
- Case n's solution attributes

Each section is followed by the character `!`. A line containing `!!` marks the end of the case base.

The field definition section looks like this:

```
search_space_size
search_space_size_type
branching_factor
regular_shapes
block_size
num_of_spaces
puzzle_size
```

Features and solution attributes share the same format. They are of the form `string range_lower range_upper`, which means that the field has a numerical range. An uninitialized field is represented by any range where `range_lower` $>$ `range_upper`. We use the values `range_lower`$=10^{100}$ and `range_upper`$=-10^{100}$ in the program. There is also a special `comment` field, which is ignored by the parser that reads the case base into memory.

Here is an example case:

```
comment                  puzzle 04 american pie
search_space_size        110000000000000000 110000000000000000
search_space_size_type   20 20
branching_factor         9.73991 9.73991
regular_shapes           0.96 0.96
block_size               2 2
num_of_spaces            6 6
puzzle_size              56 56
!
algorithm                2 2
heuristic_weight         30 30
spacevalue               45 45
restingblock             2 2
!
```

7.2.2 Measuring similarity

Let $a_1, \ldots, a_n$ be the features of a case a. We use Manhattan distance to measure the similarity between two cases a and b:

$$s_{a,b} = \sum_{i=1}^{N} |a_i - b_i|.$$

The features are not scaled, with one exception. The *search space size* and the *search space size type* are scaled non-linearly before the similarity is calculated. Let x be the search space size, and let y be the search space size type. a_1 is derived from x and y using the following formula:

$$a_1 = \frac{\ln x}{\left(\sqrt{2}\right)^y}. \tag{7.1}$$

The reasoning for the above formula is the following: The search space size is a value that grows exponentially in the number of blocks of a puzzle. Therefore, we take the logarithm of the value before calculating the similarity. In addition, the search space size is calculated in three different ways. The tightness of the bound x is decreasing when y decreases. The denominator of the fraction above was chosen arbitrarily, and was not changed since the formula gave acceptable values.

Chapter 8

Experimental results

This chapter contains the results of testing our implementation against our test suite. A discussion of these results will be presented in chapter 9.

8.1 How the tests were run

We did multiple tests, as described below.

For each puzzle, we ran the meta-reasoner, and let it run until it either solved the puzzle, until it terminated because it ran out of memory (or entered the stage where it started using virtual memory, slowing down the program by a factor of 100 and more, at which point we cancelled the program). We then recorded the result. The result includes whether the puzzle was solved, search effort (number of positions examined), solution length (if the puzzle was solved), closest distance to goal (if the puzzle wasn't solved). Two sets of results were stored, the best result achieved while being limited to examining 5 million nodes, and the best result achieved without this restriction.

For each puzzle where we had the puzzle and its solution in case base, we ran the meta-reasoner twice: Once with the particular case in the case base, and once with the case removed.

For each puzzle, we ran the solver with 4 pre-defined settings not tailored to any puzzle. All searches were limited to examining 5 million positions for the test where we compared the results from using these settings to the results from running the meta-reasoner.

In addition, additional tests were run to test specific components of our system:

- Test memory usage/speed of permutation rank, compared to Huffman code. Also, report on which puzzles the permutation rank was actually used.

- Solve an easier version of the puzzle.

The following machines were used for testing:

- tesla at the HPC-lab at NTNU: Intel i7 3.2 GHz (4 cores), 12 GB RAM running Linux Ubuntu 64-bit.
- hpc05 at the HPC-lab at NTNU: Intel Q9550 2.83 GHz (4 cores), 2 GB RAM running Linux Ubuntu 64-bit.
- Multicom Xishan (laptop): Intel i7 1.6 Ghz (4 cores), 4 GB RAM running Windows 7 64-bit.
- Komplett-PC: AMD64-X2 3800+ 2.01 GHz (2 cores), 2 GB RAM running Windows XP 32-bit.

To give an indication of the relative speed of the machines, we ran a benchmark. The table below shows the time needed for solving puzzle 11 using BFS. Solving the puzzle requires 260 steps with 2635906 positions examined.

Machine	Time needed to solve puzzle 11
tesla	43 seconds
hpc05	64 seconds
Multicom	131 seconds
Komplett	283 seconds

8.2 Test suite and case base

The test suite consisted of 46 puzzles, of which 26 were used in the specialization project. The case base contained 17 puzzles with accompanying solutions. 15 of the entries in the case base were solutions found during the specialization project. For most the these puzzles, we found multiple solutions during the specialization project. For each puzzle, we chose the solution requiring the least search effort for the case base. The two last cases in the case base are two 100×100 puzzles, one with only a master block, and one filled with 1×1 blocks leaving only one space.

Our puzzles are simply numbered from 01 to 46, and the case base contains the puzzles 01, 02, 04, 05, 06, 08, 11, 12, 13, 16, 17, 18, 24, 25, 26, 28 and 29.

Some additional tests were run, where the limit (5 million positions) were equal for all tests. These tests included running the meta-reasoner on all puzzles (disregarding the 26th run without a limit), and testing all puzzles with 4 generally good parameter sets (not tailored for any specific puzzle):

- BFS.
- Aggressive BFS (with strict resting block pruning and space value cutoff $2 \cdot s^2$, where s is the number of spaces in the puzzle.

- A* with weight 5 and Manhattan distance heuristic (h0).
- Aggressive A* with weight 10, the h2 heuristic with parameters $(6{,}4 \cdot s^2{,}2)$ where s is the number of spaces in the puzzle.

In addition, the results from the tests above will be compared against the results from the specialization project, where the results were found after extensive parameter tweaking. This comparison will only be done for the 26 puzzles which were part of the specialization project.

8.3 Raw results

In this section we present the results from running the meta-reasoner on all puzzles, as well as comparing them to the results from solving the puzzles with the solver using a couple of standard settings.

Table 8.1 shows which puzzles we solved across all the tests and parameter settings we used. The table also marks whether the puzzle was part of the case base. The puzzles solved in the specialization project and not in this thesis, are not marked as solved.

8.3.1 Results from running the meta-reasoner

This section contains results from running the meta-reasoner on all puzzles, without any limit on positions examined. Table 8.2 shows the results for every puzzle which has an entry in the case base; for each puzzle it is shown whether it was solved, and how many times the meta-reasoner launched the solver. We ran the meta-reasoner on each of these puzzles twice; once with the complete case base, and once where we removed the puzzle we were trying to solve. The meta-reasoner gave up if it couldn't find a solution after running the solver 26 times.

The results from running the meta-reasoner on the other puzzles is shown in table 8.3. The table shows whether the puzzle was solved, and how many times the meta-reasoner launched the solver.

Puzzle 46 deserves special mention. During testing, there was a bug in the feature extraction routine which caused it to crash on puzzle 46. This caused the meta-reasoner to use the features of the puzzle analysed in a previous run of the system. We managed to solve puzzle 46, but using the wrong data. After correcting the bug, our system could not solve puzzle 46.

Table 8.1: Overview of which puzzles we solved.

Puzzle	Case base	Solved	Puzzle	Case base	Solved
01	✓	✓	24	✓	✓
02	✓	✓	25	✓	✓
03			26	✓	✓
04	✓	✓	27		
05	✓	✓	28	✓	✓
06	✓	✓	29	✓	✓
07			30		✓
08	✓	✓	31		
09			32		✓
10			33		
11	✓	✓	34		
12	✓	✓	35		✓
13	✓	✓	36		
14			37		
15			38		✓
16	✓	✓	39		✓
17	✓		40		✓
18	✓		41		✓
19			42		
20			43		✓
21			44		✓
22			45		✓
23			46		✓

Table 8.2: Results from running the meta-reasoner on the puzzles in the case base.

	Puzzle in case base		Puzzle not in case base	
Puzzle	Solved	Number of times solver was run	Solved	Number of times solver was run
01	✓	1	✓	1
02	✓	1	✓	1
04	✓	1		26
05	✓	1	✓	1
06	✓	2	✓	1
08	✓	1	✓	1
11	✓	1	✓	1
12	✓	1	✓	5
13	✓	18	✓	26
16	✓	1	✓	20
17		26		26
18		26		26
24	✓	1	✓	2
25	✓	1	✓	1
26	✓	26	✓	26
28	✓	1	✓	1
29	✓	1	✓	2

Table 8.3: Results from running the meta-reasoner on puzzles not in the case base.

Puzzle	Solved	Number of times solver was run
03		26
07		26
09		26
10		26
14		26
15		26
19		26
20		26
21		26
22		26
23		26
27		26
30	✓	1
31		26
32	✓	2
33		26
34		26
35	✓	26
36		26
37		26
38	✓	1
39	✓	1
40	✓	3
41	✓	2
42		26
43	✓	1
44	✓	1
45	✓	3
46	✓	26

8.3.2 Comparing the meta-reasoner against a few preselected parameter settings

In this section we present the raw results from solving the same same puzzles with three different methods: The meta-reasoner, the solver using 4 preselected parameter settings and the hand-tweaked parameters from the specialization project. We only have hand-tweaked results for puzzle 01-06 and 09-28.

Table 8.4 shows the results from running the meta-reasoner and running the solver with the 4 preselected parameter settings. In every run, the search was limited to examining 5 million positions. For solved puzzles, the result is shown as a pair x/y where x is the number of steps needed to solve the puzzle, and y is the number of positions examined. If a puzzle was solved multiple times with the same method, the result that examined the fewest positions is shown. If a puzzle is not solved, a number in parenthesis is shown, indicating the fraction of the distance to the goal the master block managed to move.

Figure 8.1 shows a chart comparing the three methods for all the puzzles they managed to solve. Since the hand-tweaked parameters are included, this figure only contains the solved puzzles among 01-06 and 09-28, the specialization project test suite. The puzzles are sorted by increasing search effort to better compare the methods as a whole. Figure 8.2 shows a similar chart comparing the meta-reasoner and the general settings. From figure 8.1 we see that the hand-tweaked settings manage to solve the puzzles with slightly less search effort than the two other methods. From the same chart, we also see that the general settings solve two less puzzles than the meta-reasoner and the hand-tweaked settings. Figure 8.1 compares the meta-reasoner and the general settings with more puzzles. The meta-reasoner manages to solve two more puzzles, but with slightly more search effort, especially for the easier puzzles.

Figure 8.3 shows a chart comparing the three methods for all the puzzles they didn't manage to solve. The progress is measured by $1 - \frac{d_m}{d}$, where d_m is the lowest Manhattan distance of the master block to the goal encountered in the puzzle, and d is the Manhattan distance of the master block to the goal at the initial position of the puzzle. A value of 0 means that the master block was never moved at all. We see from the chart that with the hand-tweaked parameters, the master block is moved a shorter distance towards the goal compared to the meta-reasoner and the general settings. In figure 8.4 we can see that with the meta-reasoner, the master block is moved slightly farther towards the goal compared to the general settings.

8.3.3 Memory usage of permutation rank

We have measured the savings in memory usage and difference in speed of using permutation rank, compared to using Huffman coding to encode positions.

We solved four puzzles with both position representations. The results are shown

Table 8.4: Detailed results of running all methods on all puzzles. The solver was limited to examining 5 million positions for each attempt.

Puzzle	Tweak	General	Meta-reasoner
01	27 / 321	31 / 180	27 / 1047
02	116 / 23688	174 / 12881	116 / 24030
03	(0.00)	(0.33)	(0.33)
04	186 / 3246310	(0.70)	(0.70)
05	83 / 35885	139 / 374571	87 / 31286
06	85 / 1731647	(0.57)	94 / 2551339
07	-	(0.13)	(0.25)
08	-	194 / 194	194 / 8689
09	(0.00)	(0.10)	(0.00)
10	(0.64)	(0.64)	(0.45)
11	260 / 2605176	281 / 2436570	936 / 1257520
12	159 / 599941	258 / 266717	159 / 613845
13	(0.73)	(0.73)	817 / 4561291
14	(0.00)	(0.00)	(0.00)
15	(0.00)	(0.09)	(0.09)
16	264 / 52376	352 / 307004	325 / 832917
17	(0.29)	(0.29)	(0.29)
18	(0.25)	(0.88)	(0.81)
19	(0.11)	(0.22)	(0.22)
20	(0.05)	(0.05)	(0.05)
21	(0.00)	(0.00)	(0.00)
22	(0.10)	(0.10)	(0.10)
23	(0.60)	(0.40)	(0.60)
24	251 / 53396	282 / 657406	216 / 939832
25	228 / 63329	255 / 119759	193 / 81927
26	(0.67)	(0.83)	(0.66)
27	(0.25)	(0.25)	(0.38)
28	888 / 138678	1168 / 104096	1196 / 96070
29	-	773 / 1540	773 / 1540
30	-	110 / 2139	71 / 3921
31	-	(0.40)	(0.40)
32	-	123 / 2737064	63 / 3060180
33	-	(0.67)	(0.78)
34	-	(0.00)	(0.00)
35	-	(0.25)	(0.58)
36	-	(0.38)	(0.38)
37	-	(0.27)	(0.45)
38	-	81 / 2225	41 / 21748
39	-	246 / 2123637	343 / 1044016
40	-	135 / 524732	136 / 45780
41	-	402 / 2745675	902 / 1794829
42	-	(0.67)	(0.67)
43	-	413 / 1114124	214 / 1451724
44	-	248 / 33247	232 / 108451
45	-	58 / 37353	448 / 263365
46	-	(0.00)	(0.00)

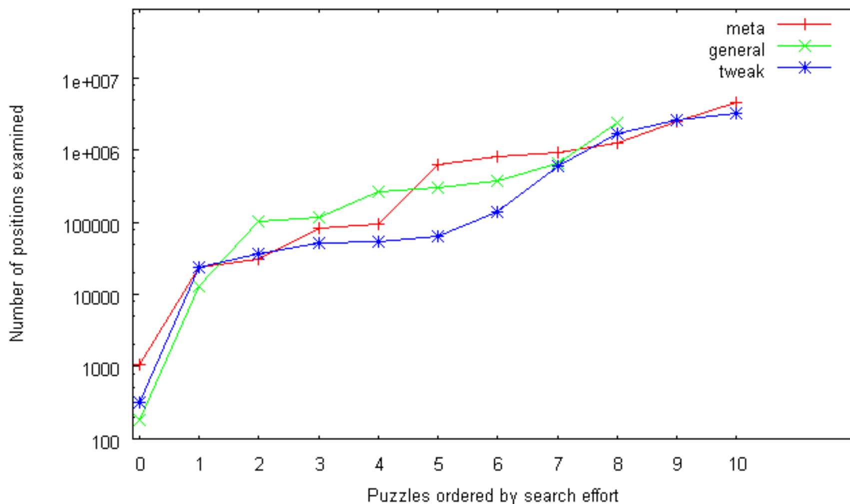

Figure 8.1: Positions examined for solved puzzles, all three methods

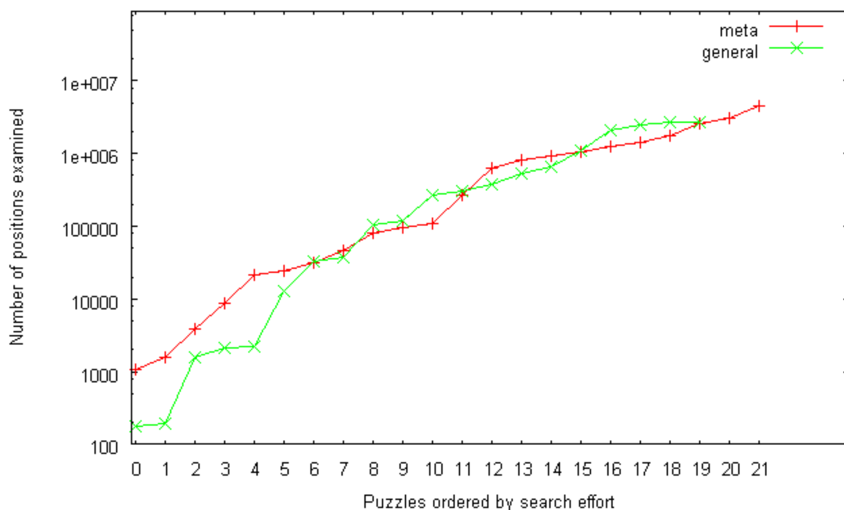

Figure 8.2: Positions examined for solved puzzles, two methods and all puzzles

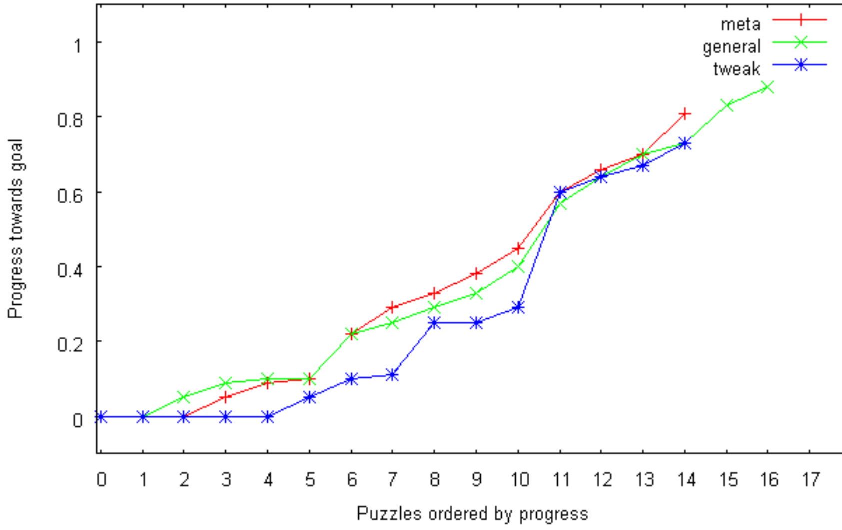

Figure 8.3: Progress for unsolved puzzles, all three methods

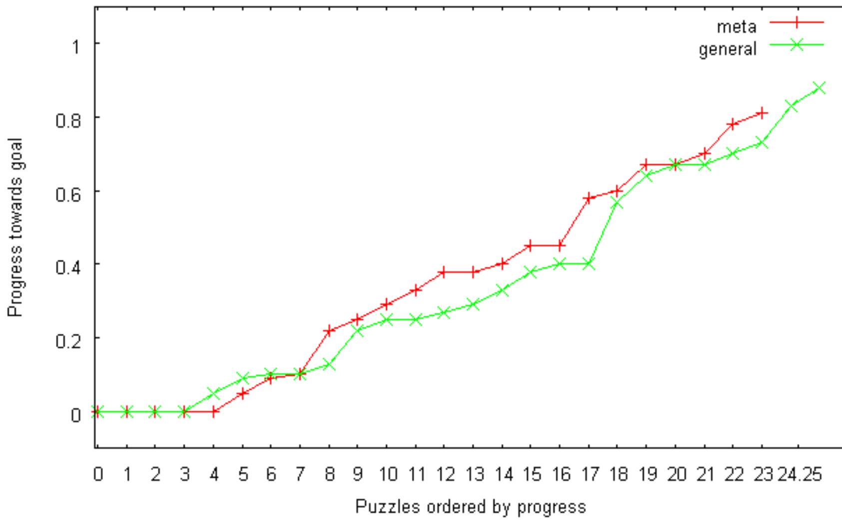

Figure 8.4: Progress for unsolved puzzles, two methods and all puzzles

Table 8.5: Runtimes of permutation rank and Huffman coding.

Puzzle	Time Huffman (s)	Time Permutation (s)	Speed difference
06	64	93	1.453
11	45	63	1.400
13	176	229	1.301
26	461	598	1.297

in table 8.5. From the table, we see that solving a puzzle using permutation rank is a factor of 1.3-1.45 slower than using Huffman coding. For the puzzles that require more time to be solved (puzzle 13 and 26), we see that the difference is closer to a factor of 1.3.

Table 8.6 shows how many bytes we need to store a position with permutation rank and Huffman coding, as well as the multiplicative difference. Puzzles requiring more than 8 bytes for the permutation rank are not included, since our program supports a maximal permutation rank of 64 bits (8 bytes). Hence, we could use permutation rank for 25 of our 46 puzzles. The required actual storage in bits is lower, but we chose to show the requirement in bytes instead of bits, because we store each position in arrays of bytes and hence we cannot make use of the unused bits for each position stored. In average, using permutation rank uses 1.239 times less memory than using Huffman coding.

However, C++ STL (at least the implementation we used) seems to align our data structures at 8-byte boundaries. So if a position needs 8 bytes when represented by its permutation rank, and 4 bytes when encoded with Huffman coding, we are not actually achieving any savings in memory. The only case where we achieve savings, is when the Huffman code representation needs 9 or more bytes and the permutation rank needs 8 or fewer bytes. From the 25 puzzles in table 8.6, this is the case for puzzles 13, 17, 31, 42 and 45, or 20% of our puzzles.

We made a small program which measured the memory usage of allocating n vectors, each containing a vector of k characters (bytes). We ran this test for different values of k near borders of alignment (3, 4, 8, 9, 16, 17, 24, 25, 32) in order to determine to which memory boundaries STL aligns its structures. We also ran the test for different values of n $(2^9, 2^{10}, 2^{11})$ to determine how much memory each structure occupies. From this test, we can conclude that our STL implementation aligns the start of `vector<char>` structures to 8 byte boundaries. Also, it shows that there is a large overhead in memory usage when using STL. For each `vector<char>`, the actual memory usage seems to be close to $\lceil \frac{k+7}{8} \rceil \cdot 8 + 22 \pm 2$.

Table 8.7 shows how many bytes STL need to store a `vector<char>` of size k for several values of k.

We can then find out the savings in memory usage for the solver itself. As long as we choose a puzzle where the Huffman code representation requires at least 9

Table 8.6: Number of bytes needed for permutation rank and Huffman coding.

Puzzle	Bytes needed for Huffman	Bytes needed for Permutation	Difference ratio	Permutation rank used?
01	3	2	1.500	
02	3	3	1.000	
05	7	5	1.400	
06	8	7	1.143	
11	5	4	1.250	
12	8	7	1.143	
13	9	7	1.286	✓
16	8	7	1.143	
17	10	8	1.250	✓
23	8	7	1.143	
24	7	6	1.167	
26	7	6	1.167	
28	5	4	1.250	
30	6	5	1.200	
31	9	7	1.286	✓
32	7	6	1.167	
33	7	6	1.167	
38	4	3	1.333	
39	6	5	1.200	
40	7	6	1.167	
41	7	5	1.400	
42	9	7	1.286	✓
43	8	6	1.333	
44	5	4	1.250	
45	9	6	1.333	✓

Table 8.7: Number of bytes actually needed by STL

Bytes to be stored	Actual memory usage in bytes
1-8	28
9-16	40
17-24	44
25-32	56

bytes, and the permutation rank requires at most 8 bytes, we save 12 bytes per position, or a ratio of 1.429. We have not measured the memory savings when using other STL structures like `map`, `set` or `queue`. The savings might be smaller, since these structures are slightly more advanced and need more internal class variables in order to perform their more advanced functions.

8.3.4 Solving an easier version of the puzzle

We tried to find a puzzle that we haven't solved yet, and make an easier version of it. We made two easier variants, one where we removed one 1×2 block, and one where we removed two 1×1 blocks. See figure 8.5. We solved both of the easier variants using the meta-reasoner. The number of steps and search effort for solving both variants are shown in table 8.8.

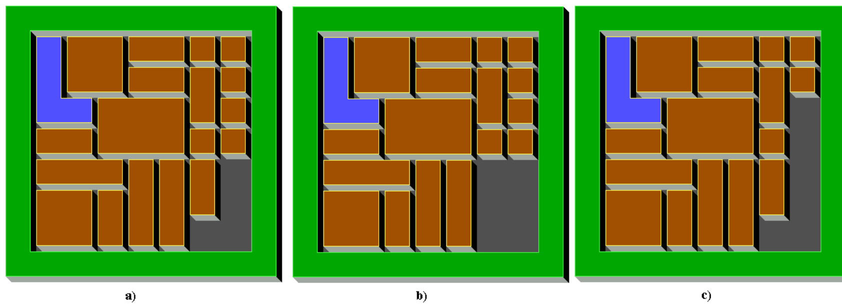

Figure 8.5: In a), the original puzzle 23 is shown. In b) one 1×2 block has been removed, and in c) two 1×1 blocks have been removed. The goal of the puzzle is to move the blue master block to the upper right. In addition, the two cells enclosed by the master block and the wall must not be occupied by any other blocks.

We then used the solutions to the easier versions as subgoals for solving the full variants, where each subgoal consisted of requiring all blocks to be at the same location as in the particular subgoal. The re-added blocks were permitted to be in any open space of the subgoal.

Since each subgoal required all blocks to be specified, we did not expect most of

Table 8.8: Results from solving the easier variants of puzzle 23

Variant	Puzzle	Steps	Positions examined
1	1×2 removed	239	67799057
2	two 1×1 removed	250	49516543

Table 8.9: Results from solving the full version of puzzle 23 using the two easier variants.

Variant	Description	Subgoal reached	Total number of subgoals	Progress (fraction)
1	1×2 removed	46	239	0.192
2	two 1×1 removed	130	250	0.520

our heuristics to work well, since they only take the master block into account. Therefore we ran the solver with plain BFS for the full puzzle. Plain BFS allows us to search deeper (examine more positions) than the other options.

We did not manage to solve the full puzzle with either of the solutions of the easy variants. In the first variant, we reached subgoal 46 of 239. In the second variant, we reached subgoal 130 of 250. We have summarised this progress in table 8.9.

Figures 8.6 and 8.7 show the number of steps needed for reaching subgoal i for variant 1 and variant 2, starting from subgoal $i - 1$, for every subgoal i that we reached. The number of steps is sorted in the charts. While the majority of subgoals could be reached in one or two steps, one subgoal in variant 2 required 77 steps, and three subgoals in variant 1 required 93, 101 and 102 steps.

Figures 8.8 and 8.9 show the number of positions needed to be examined before reaching subgoal i, for variant 1 and variant 2. The majority of subgoals could be reached while examining less than 100 positions, but the two most expensive subgoals required 33.5 million and 43.9 million positions.

8.4 Problems encountered during testing

When running the tests, our test machines frequently ran out of memory. For the hardest tests (puzzles requiring over 10 million positions to be examined before finding a solution), only the Tesla machine (12 GB RAM) was useful. The reason is the excessive memory requirements of the solver program, which caused the program to quickly run out of available memory on the weaker machines (ranging from 2 to 4 GB). The efficiency of the program was drastically reduced when memory was exhausted and started using virtual memory. In addition, on our 32-bit operating systems, the program was aborted by the memory manager in STL

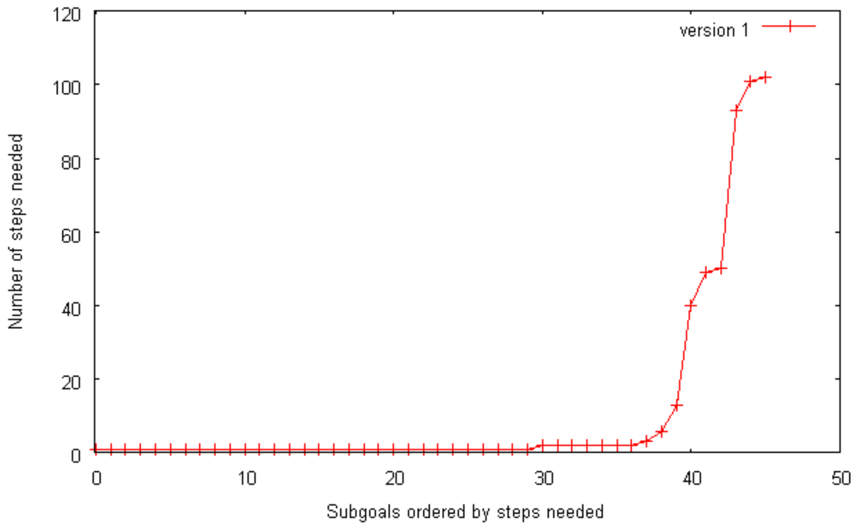

Figure 8.6: Puzzle 23 variant 1, subgoals ordered by number of steps needed to reach the subgoal starting from the previous subgoal.

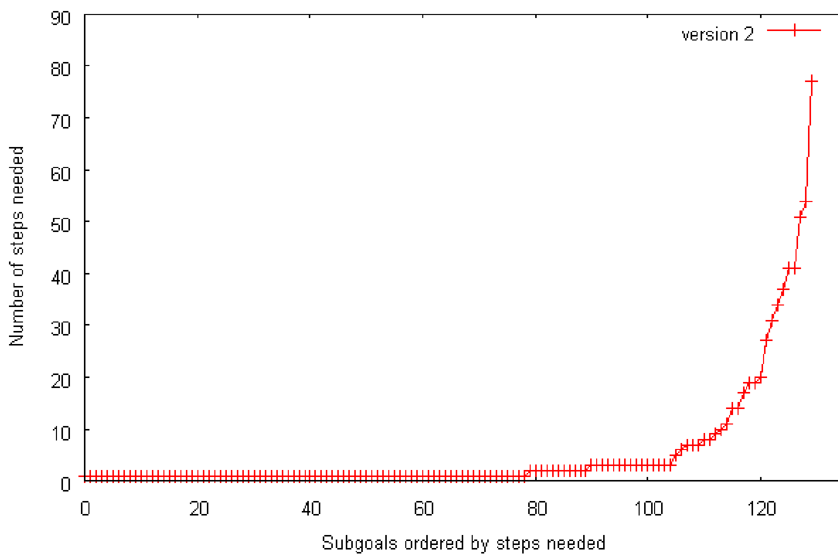

Figure 8.7: Puzzle 23 variant 2, subgoals ordered by number of steps needed to reach the subgoal starting from the previous subgoal.

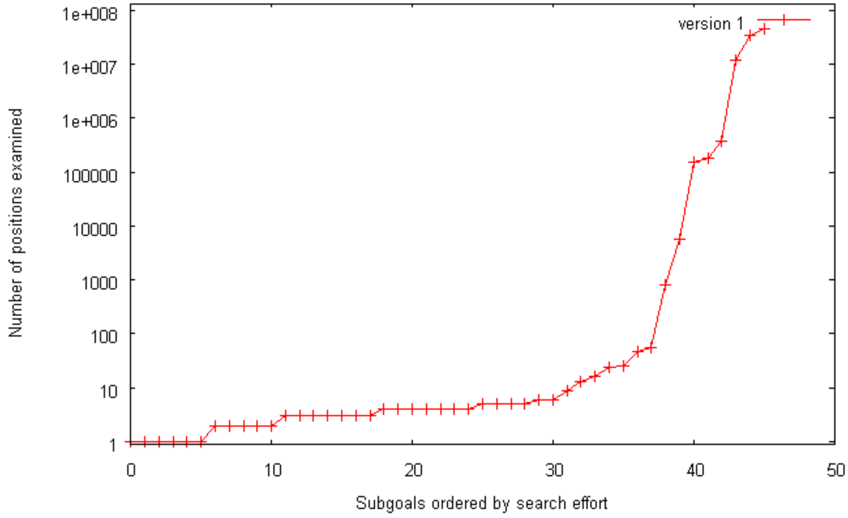

Figure 8.8: Puzzle 23 variant 1, subgoals ordered by search effort needed to reach the subgoal starting from the previous subgoal.

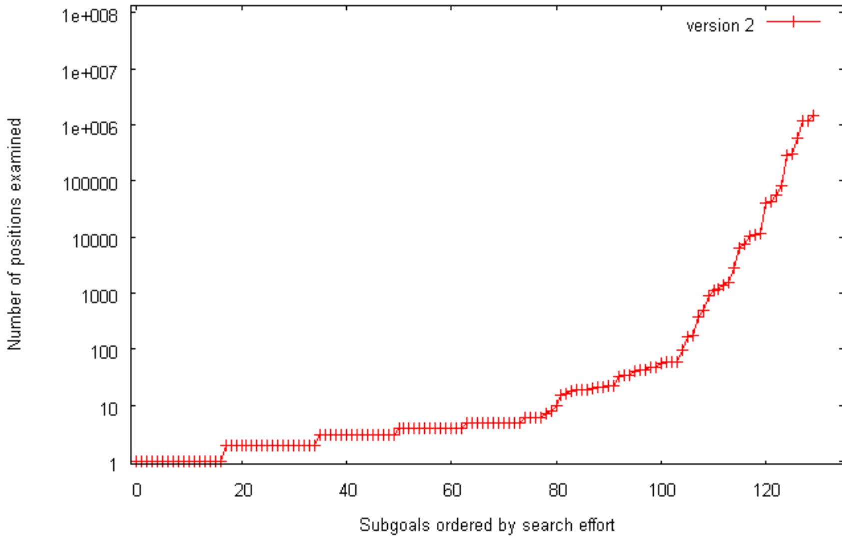

Figure 8.9: Puzzle 23 variant 2, subgoals ordered by search effort needed to reach the subgoal starting from the previous subgoal.

after using around 1.5 GB RAM, which seems to be the effective memory limit for programs running in 32-bit versions of Windows XP and Windows 7.

We had a bug in the meta-reasoner which was discovered quite late in the testing, and close to the thesis deadline. The bug caused the permutation rank to never be used by the solver. This caused the solver to require more memory for 5 of the puzzles. For 3 of these 5 puzzles this was of no consequence, as we managed to solve them anyway. We ran the meta-reasoner again for the two remaining puzzles. With permutation rank, the program was able to search deeper before exhausting the memory. However, this was not sufficient for solving the two puzzles.

Another bug was discovered just one day before the deadline of the master thesis. The bug happens on only one puzzle, puzzle 46. The routine that gathers features from the puzzle crashes because one of the blocks cannot fit inside a 64-bit bitmask used in the routine for calculating the packing bound. The crash causes the results of the feature extraction to not be written to disk. The meta-reasoner then proceeds to solve the puzzle, using a file from a previous run of the meta-reasoner. Puzzle 46 was one of the puzzles we considered to be difficult (we did not manage to solve it using the general parameters), and we managed to solve it using data from the wrong puzzle. The bug was fixed by adding one line to the code, which exited the routine gracefully. The reason why the bug went undiscovered for so long, was that our program generates a lot of output, and the error message went by unnoticed. We were not able to determine the puzzle the features belonged to.

Chapter 9

Discussion

In this chapter we present our discussion and analysis of the implementation and the results from running the program. We will analyse each subcomponent of the CBR system, and each improvement in the underlying solver program, and give an explanation of the test results.

We will discuss each of the multiple tests we have performed.

9.1 Solver

In the solver program, we added the option of displaying the entire solution when a puzzle was solved, support for two new block types (sliders and disconnected blocks) and permutation rank, a slightly more memory-efficient way of representing positions. Also, a helper function for the meta-reasoner system was added into the solver, which analyses a given puzzle, extracts features and communicates them to the meta-reasoner.

The option of displaying the solution to a puzzle gave us a new way of attacking unsolved puzzles. Given a hard puzzle, we could modify it, solve the modified version and use its solution as a plan to solve the original puzzle. We will analyse whether this idea was successful.

9.1.1 Use the solution of an easier version to solve the full puzzle

We tested on puzzle 23, which we haven't managed to solve so far. Figure 8.5 in the previous chapter shows the puzzle, as well as the two easier versions we made. In version 1 we removed one 1×2 block, and in version 2 we removed two 1×1 blocks. As reported in section 8.3.4, we didn't manage to solve the full puzzle using

the solution for the two easier versions. Progress towards the goal was 19.2% for version 1 and 52% for version 2.

We experienced some small problems while running these tests. When the solver is given a puzzle containing multiple subgoals, the standard option is to not erase the set of visited positions after completing a subgoal. However, it turned out that the progress towards the final goal isn't necessary linear. For several of the subgoals, we could read the subgoals in fewer steps if we allowed the search to visit positions which were visited during the search for previous subgoals. We therefore changed the options in the solver so that the set of visited positions was deleted each time a subgoal was reached.

The results from trying to solve the full puzzle using the solution from the first version made us realise a flaw in our idea: Some positions are not possible to reach. The 47th subgoal, the one we failed to reach, is shown in figure 9.1. This subgoal specifies the position of every block in the puzzle, except one 1×2 block which is allowed to be put anywhere permissible. However, the block cannot be placed anywhere; there is no 1×2 subrectangle of free space. Hence, the full puzzle was impossible to solve using this set of subgoals.

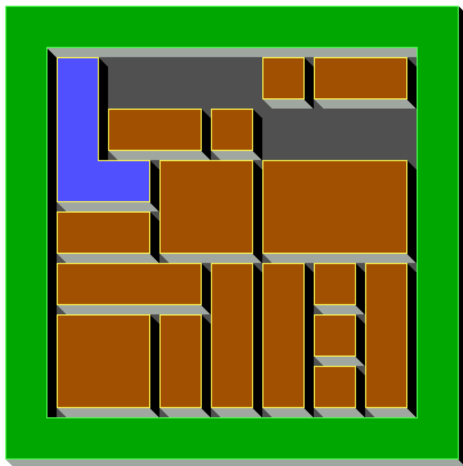

Figure 9.1: A subgoal which is impossible to reach, since the removed 1×2 block can't be reinserted into this position.

We created a second easier version of the full puzzle (figure 8.5 c)). This time, there existed no situation like the above where we couldn't reinsert the blocks. This search failed to reach the 131st subgoal. Figure 9.2 shows the 128th and the 131st subgoal (a) and b), respectively). (The 129th and 130th subgoal have the 2×2 block in intermediate positions.) Since the full puzzle has two additional 1×1 blocks, they need to exchange places with the upward-moving 2×2 block. Our program didn't manage this transition after searching to depth 124 (from the 130th subgoal), having examined 741 million positions. Moving a 2×2 block three

steps in one direction require 6 spaces, which the full puzzle doesn't have, so an undetermined number of intermediate steps are needed.

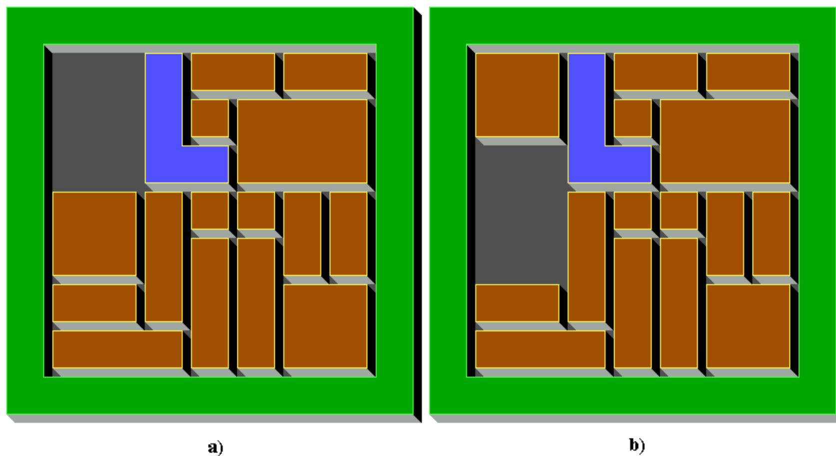

Figure 9.2: An example of a very hard transition. We didn't manage to reach position b) starting from position a) when two 1×1 blocks were added.

In order to have greater success with this approach, care should be taken when designing modified versions of the original puzzle to avoid unsolvable subgoals (like version 1 above) and very hard subgoals (version 2). In general, it is desirable to avoid to be able to move certain key blocks a longer distance in the modified version than in the full puzzle. The hard subgoal from version 2 could be prevented by removing fewer blocks (removing only one 1×1 block), this would disallow movement of the 2×2 block by 3 steps, or other potential hard transitions like moving the 3×2 block up two steps. Solving the original puzzle with one 1×1 would probably be much harder than solving the two versions mentioned above. Finding the correct amount of blocks to remove is a difficult tradeoff between the hardness of solving the modified puzzle and the usefulness applying its solution to the full puzzle.

Another problem with this approach is the increased memory requirements of the solver when trying to solve the modified version of a puzzle. In order to be able to input the solution, we must store the parent position for each position we visit. This roughly doubles the memory requirements, and renders some of the harder otherwise solvable puzzles infeasible to solve. Solving versions 1 and 2 took around 30 minutes without displaying the solution, but around 8 hours with the option turned on.

9.1.2 Permutation rank

This improvement was only effective for 5 of 46 the puzzles. 21 of 46 puzzles were immediately excluded because they would need a permutation rank of larger than 64 bits, which we chose to not implement. Hence, we had 25 eligible puzzles where permutation rank could be used. For 20 of these, permutation rank didn't yield any savings, because of the way STL allocates memory in blocks of 8 bytes. If permutation rank reduced the memory requirements per positions from 7 to 5 bytes, no savings were achieved because each position occupied a multiple of 8 bytes. If we have allocated memory in 1-byte blocks, 24 of the 25 puzzles would achieve savings in memory usage, and the average savings for representing a position in memory would be a factor of 1.239. Only puzzle 2 has the same memory requirement for both permutation rank and Huffman coding - 3 bytes for each position.

Because of the way memory allocation worked, and the fact that it only lead to memory savings for 5 out of 46 puzzles, this improvement was of little value for us in the thesis. 3 of these 5 puzzles were solvable without this improvement. The two remaining puzzles could still not be solved with this improvement.

See section 11.3.6 for suggestions on how to make this optimisation effective.

9.1.3 New block types

Support for disconnected blocks and sliders were added mainly in order to add some more puzzles of the desired difficulty level into the test suite. We wanted a few more puzzles which we estimated to be just a bit harder than the hardest puzzles we solved in the specialization project.

To test this new addition, we added some puzzles ranging in difficulty from trivial to intermediate (puzzles 30, 32, 38, 40). As we expected, we solved all of them.

Harder puzzles were added (31, 33, 34, 46). These proved to be a challenge, and in the end we solved one of them. Puzzle 46 (which contains sliders) was solved by the meta-reasoner and it was solved in the 26th run of the solver, with the same parameters as the most similar case (A* with the aggressive h2 heuristic, and resting block pruning, which helped reduce the search space size). However, because of a bug, this puzzle was solved using data from another puzzle. See section 8.4 for more details. We were not able to solve the puzzle again after fixing the bug.

Disconnected blocks was problematic for the space value pruning method. In order to move a disconnected block, spaces have to be spread in front of each disconnected component of the block. This is not achieveable if a too low space value cutoff is chosen. Hence, this block type effectively nullifies this particular improvement.

Sliders work better together with our solver program and its enhancements. The presence of sliders cause restrictions on the number of possible moves in a position,

which in turn causes the branching factor to be reduced. Puzzles with many sliders also typically need more steps to be solved.

9.2 Meta-reasoner

During a typical attempt to solve a hard puzzle, the meta-reasoner will launch the solver program multiple times. Since it could take 10 hours or even more for just one run of the solver to run out of memory, we limited the number of examined position to 5 million, and the number of solver runs to 26 (of which only the last is allowed to examine an unbounded number of positions). Each solver run then took up to half an hour. This reason for this decision was to ensure that we would be able to run all the tests in time.

Even though we didn't have time to perform tests using higher limits, we believe that both limits, 26 runs and 5 million positions, held back the performance of our system. We will go into further details of the impact of the limits in the following sections.

9.2.1 Meta-control cycle

The meta-control cycle consists of three phases:

- Reusing cases from the case base.
- Doing a randomized proximity search for finding new parameters to try.
- A final attempt at running the solver using the most promising parameter set, without the limit of 5 million expanded positions.

The solver program is launched a total of 25 times during phase 1 and 2. For the 26th and final run, the parameter set which managed to move the master block closest to the goal, is chosen.

Almost all of the puzzles that we solved, was solved using a solution from the case base. This either happened in phase one or in phase three. The reason for why some of the puzzles was solved in phase three instead of phase one, was that the solution required more than 5 million positions to be examined. We would have been able to solve more puzzles in phase 1 if the 5 million limit was raised.

Only puzzles 13 and 16 were solved in phase 2, and this did not happen consistently because of the random nature of phase 2. (Puzzle 13 was also solved in phase 3 in another run, and puzzle 16 was solved in phase 1 when the case base contained the case with its solution.) This shows that the randomized search was able to come up with better parameters than the case base, even though it happened rarely. It is likely that more puzzles would be solved in phase 2 with a higher limit than 26 on the number of runs.

The meta-control cycle maintained a lower bound for legal space value cutoffs. Whenever the solved returned after not being able to find a solution, this could only be caused by a too low space value cutoff (by design, no other options could cause a solvable puzzle to be unsolvable). The lower bound started with a value of 1, and if a puzzle could not be solved using space value n, the lower bound was set to $n+1$. This lower bound was not stored between different runs of the meta-solver. This lower bound was necessary in order to avoid wasted runs with the solver, but 26 runs were far too few to be able to establish a close lower bound, and during the course of running the meta-reasoner we rarely ended up calling the solver with good values for the space value cutoff, except when retrieving them from cases.

For the 26th and final run of the solver, the most promising case of the 25 previously tried ones are taken. However, when we ran the meta-reasoner on puzzle 17 with its correct solution in the case base, it mistakenly picked another case to use for the final run. The chosen case managed to move the master block 5 squares away from the goal with less search work than the known good case, which got the master block within the same distance. However, the chosen case did not have the needed space value cutoff, so the puzzle could not be solved. On the other hand, many puzzles were solved in the 26th run, including several where it was not easy to find good parameters. We did not manage to solve puzzle 46 with the general parameters we prepared, but the meta-reasoner managed to solve it (due to a bug described in section 9.3, it was solved based on a feature vector from another puzzle). From this, we can conclude that the way the meta-reasoner selects the case to use for the 26th run worked satisfactory in most cases, even it it did not pick the correct case for one difficult puzzle.

9.2.2 Matching of cases

In section 8.3.1 we presented the results from running the meta-reasoner on every puzzle in the case base, both with and without the puzzle itself in the case base. The main results are repeated in table 9.1 for convenience.

The main difference is that we lost puzzle 4 when we took out the case, and several puzzles needed more work (in number of calls to the solver) before they were solved. Most notably, puzzle 16 went from 1 to 20 calls to the solver, and puzzle 13 went from 18 to 26.

Surprisingly, puzzle 6 went from 2 to 1 call when we took out the case. As the case base consisted of the best solutions we found by hand during the specialization project. This means that the case with the second best similarity was a better match for puzzle 6, since this case managed to solve the puzzle with fewer examined positions. The case chosen was the solution to puzzle 16 and used a more aggressive heuristic, taking the positions of spaces relative to the master block into consideration.

The meta-reasoner was not able to solve puzzles 17 and 18. For puzzle 17, the meta-reasoner couldn't solve the puzzle using the correct case within 5 million examined

Table 9.1: Results from running the meta-reasoner on the puzzles in the case base.

	Puzzle in case base		Puzzle not in case base	
Puzzle	Solved	Number of times solver was run	Solved	Number of times solver was run
01	✓	1	✓	1
02	✓	1	✓	1
04	✓	1		26
05	✓	1	✓	1
06	✓	2	✓	1
08	✓	1	✓	1
11	✓	1	✓	1
12	✓	1	✓	5
13	✓	18	✓	26
16	✓	1	✓	20
17		26		26
18		26		26
24	✓	1	✓	2
25	✓	1	✓	1
26	✓	26	✓	26
28	✓	1	✓	1
29	✓	1	✓	2

positions. The program eventually found another case which was considered to be better, this case was used for the 26th run without a limit. This case did not turn out to be better than the correct case, it missed the correct space value cutoff which was vital for solving this problem. We solved puzzle 18 in the specialization project, but we used one of the options (manually locking specific blocks) which we decided to exclude in this thesis.

The meta-reasoner was able to find a much better solution to puzzle 13 than the one found manually in the specialization project. It managed to solve the puzzle using 4.56 million nodes, compared to the old result of 609 million nodes. The old solution used BFS, but the meta-reasoner managed to find good parameters for A* during its second stage, the randomized parameter search. The good parameters we found are very similar to a set of parameters tried in the specialization project which didn't solve the puzzle. The only significant difference between the parameter sets is that we used resting block pruning in the specialization project for this puzzle, and this optimization might have pruned away the solution path.

9.2.3 Finding similarity between cases

The similarity function was designed early, and it was never tweaked afterwards, because the system as a whole gave acceptable results. Except from the two very hard puzzles in the case base, we managed to solve all of them except one when we removed the puzzle we were trying to solve from the case base. This is an indication that in the absence of the correct solution from the case base, the system managed to find a similar case which worked. Since multiple cases were always reused, our similarity function didn't need to be perfect.

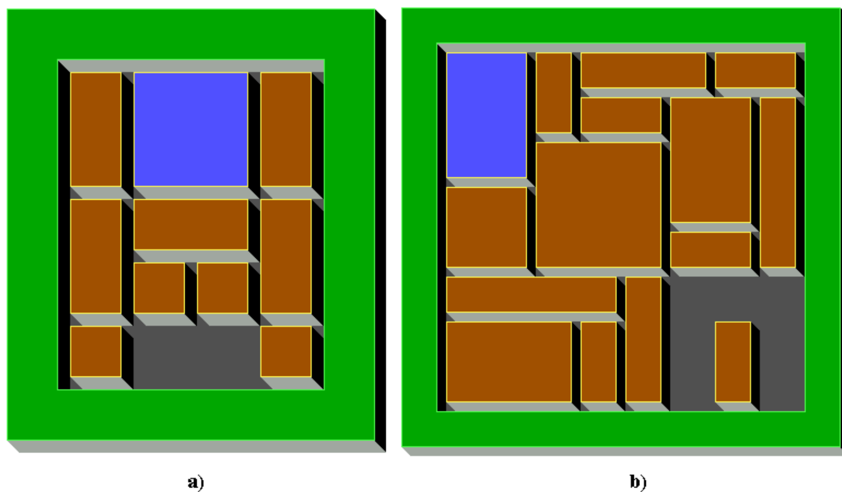

Figure 9.3: Puzzle 02/44 (a)) and puzzle 13 (b))

However, it is easy to design cases that trip up our similarity function. For instance, our puzzle 44 is identical to puzzle 02 (see figure 9.3 a)), except that each block is twice the size. According to our similarity measure, puzzle 13 (figure 9.3 b)) is the best match, and several puzzles are more similar to puzzle 44 than the correct match, puzzle 02. Puzzle 44 got matched up with a puzzle of similar size and similarly sized blocks instead of the much smaller, identical puzzle. Puzzle 13 is much harder to solve than puzzle 44, as it requires 4.5 million positions to be examined to solve it (in addition to needing a hard-to-find set of parameters), compared to puzzle 44 which needs 33247 positions to be examined. Our similarity function was not designed for puzzles scaled in this way.

9.2.4 Randomized proximity search

The randomized proximity search was performed during phase 2 in the meta-control cycle. Its goal was to find eligible parameters to pass on to the solver, parameters that haven't been tried before that are similar to parameters already tried.

As mentioned in section 9.2.1, the contribution of this search to the result of the meta-reasoner was limited. Out of 72 attempts to solve puzzles with the meta-reasoner, two of these attempts were successful because of this search method. This is, however, more a result of the few iterations the search was allowed for perform, rather than a weakness in the search algorithm. Of the 25 calls to the solver in phase 1 and 2, around 20 of them use parameters from the randomized search. We believe the search could have found more good parameters if it was allowed to run for more iterations. In order to allow this, we would need a much larger time frame for testing.

9.3 Other issues

We had one major bug in the code which changed the outcome of the tests. Our program managed to solve puzzle 46, despite a bug which made the feature extraction portion of the program to crash, forcing the meta-reasoner to use the features from the previous puzzle.

Chapter 10

Summary and conclusion

In this thesis, the goal was to construct a system which solves sliding-block puzzles using meta-reasoning, and it should be based on the search-based solver program from the specialization project. The meta-reasoner should automatically find good parameters to call the solver with, with the purpose of solving a given puzzle. In the specialization project, the parameters had to be found manually.

After studying relevant literature, we decided to go for a CBR approach, where each case consists of the features describing a puzzle and a set of parameters that solve the puzzle. We enhanced this approach by adding a meta-control cycle where multiple cases would be retrieved from the case base. After applying all the promising cases on the problem to solve, a randomized proximity search would start generating new parameters, close, but not identical to the parameters already tried. For each such set of parameters, the solver would run with a limit on the allowed search work. After the meta-control cycle has run for a specified number of iterations, the set of parameters which managed to move the master block closest to its goal would be run again, this time with no limit on the allowed search work.

We enhanced the solver program with three new features. We added the ability to store and display each step of the puzzle's solution. A new memory-efficient way of representing puzzle configurations was added, based on its permutation rank when the configuration was seen as a permutation of a multiset. Two new block types, sliders and disconnected blocks were added, so that we could add more interesting puzzles to our test suite.

The resulting system was able to find parameters that solves the puzzles with slightly less search work than a set of general parameters which were not adapted to any specific puzzle. In addition, the system was able to solve one new puzzle (which unfortunately was the result of a bug, causing the system to reason from the wrong data) compared to the same set of parameters. We also compared the results against the parameters found manually in the specialization project. These parameters were adapted to each puzzle. The meta-reasoner was not able to fully

match these results, except in one exceptional case where the meta-reasoner reduced the search work by a factor of 133. This result was due to the randomized proximity search in the meta-control cycle. Compared to the manually found parameters, the meta-reasoner couldn't solve one puzzle that required a very specific set of parameters. The meta-reasoner made better progress on the unsolved puzzles than the manually found parameters. If we took out the solution from the case base for the puzzle we tested, the meta-reasoner solved one less puzzle, and needed more search work in general to solve the other puzzles.

The memory-efficient representation (permutation rank) was not as efficient as it could have been, because of the way C++ STL allocates memory. There was significant overhead because each variable was aligned to 8-byte boundaries.

The new functionality of storing the solution sequence for a solved puzzle was used in another test. We tried to solve a hard puzzle by using the solution of a slightly easier version of the same puzzle where some blocks were removed. The plan was to solve the hard puzzle by reaching intermediate positions. Each step in the solution of the easier version of the puzzle corresponded to one such intermediate position, where the removed blocks from the full puzzle could be placed in any free position. Our first attempt failed on an early intermediate position because one of the intermediate positions was illegal; the removed blocks didn't fit in the shape of the free space. The second attempt failed around halfways because it couldn't find the path between a particularly hard transition between two intermediate positions.

The domain of sliding-block puzzles is a very challenging domain, both for heuristic search methods and reasoning methods. Research on other domains such as Sokoban has reached a more advanced stage, where most of the puzzles designed for people are now possible to solve with computer programs. We believe that our domain can reach such a stage as well where most puzzles can be solved by computer programs.

There are still many interesting ideas to explore in this domain, these are explained in more detail in chapter 11.

Chapter 11

Future work

In this section we will identify areas where our system can be improved. Some of the improvements are suggested based on weaknesses identified in our system. Other suggestions are based on areas we haven't researched yet and areas that were out of scope for this thesis.

11.1 System architecture

The current code base consists of 3000 lines divided into two programs (solver and meta-reasoner). After one year of development, the solver code is now rather messy and it is hard to maintain. During this thesis, we had at one serious bug which affected the results of the system. This is described in detail in sections 9.1.3 and 9.3. The program is quite slow, since it was designed to support many features rather than optimized for raw speed. We suggest redesigning and rewriting the solved program, and split into multiple executable files, one for each main algorithm (BFS, A*) and one for each memory representation (Huffman coding, permutation rank). This would make it much easier to optimize the program for speed.

11.2 Meta-reasoning program

In our meta-reasoning program, we had many arbitrary constants and limits, for instance the 5 million limit for the number of examined positions. Because of the time it takes to run tests, we did not have time to change and tweak these limits as much as we had hoped to.

11.2.1 Case matching and similarity

Our similarity function can be improved. Currently, it overemphasizes the importance of the puzzle size, this was pointed out in section 9.2.3. With a more accurate similarity function, we would increase our chances of finding a better match for the input case, increasing our chances of solving the puzzle.

11.2.2 Other AI methods

Quite early in the project, we decided to restrict ourselves to looking at reasoning methods. It remains to investigate how useful other AI methods could be, like decision tree learning, neural networks and machine learning.

11.3 Solver program

Our solver program is very slow, and can be optimised to run faster. In addition, there are several areas of improvement that we haven't investigated.

11.3.1 Pattern databases

Pattern databases have been used with great success in similar domains like the $nm-1$-puzzle[8]. The benefit of the domain of $nm-1$-puzzles is that the puzzles are built from only 1×1 blocks. While our domain can contain arbitrarily shaped blocks, and hence it looks harder to apply pattern databases to our domain, it remains to research whether this approach is useful.

11.3.2 Macro moves

Macro moves could be a valuable addition for making the solution sequences shorter and reducing the search space size by identifying commonly occurring block-moving situations and transition. Because of the variety of block shapes, it is likely that we have to define new macro moves for each puzzle we consider. It remains to be investigated whether this approach is useful for our domain.

11.3.3 Moves

In section 3.1 we defined two different actions for taking us from one state to another, moves and steps. Moves have some desirable properties when used in combination with some of our improvements. Moves would be a useful addition to the program.

11.3.4 BDD

It remains to investigate the memory usage of BDD, and compare the memory usage and efficiency of a BDD implementation to an implementation where positions are stored in a traditional way. With BDD, the memory usage of representing Sokoban positions were greatly reduced, and similar savings could be achieved in other domains.

11.3.5 Optimizations for speed

As previously mentioned, our solver program is slow, and there exists a specialized BFS program [25] which is 5.2 times faster than our program. Making our program faster would be of enormous benefit, since we then could raise the limits of solver runs and positions examined in the meta-reasoner, and increase our chances to solve more puzzles.

11.3.6 Optimizations for reduced memory usage

In section 8.3.3, we showed that there is a large overhead in memory usage for storing positions, because of memory alignment to 8-byte boundaries and to store the size field of each vector holding the characters that represent a position. This overhead varies with the size of the puzzle, but for smaller puzzles this overhead is larger than 100%. In addition, the memory alignment nullified the more efficient permutation rank representation for the majority of the puzzles. This overhead can be reduced in several ways:

- Custom allocators in C++ STL. It remains to investigate this approach in detail, but it is likely that this approach won't reduce the entire overhead without redesigning our data structures.
- Manage the memory ourselves. We can allocate a large buffer, and manage this chunk of memory by ourselves. In this way, we can align positions to every byte (or even every bit) and remove all the overhead. The implementation complexity is higher than custom allocators, especially for data structures like priority queues and sets, which are used for our A* algorithm.

11.4 Other areas of improvement

11.4.1 Parallelism

Korf[24] used a parallel BFS algorithm for solving random instances of the 24-puzzle. A similar approach could be used to speed up our BFS algorithm, in order to be able to search even deeper within reasonable time.

Also, parallelism can be implemented in a more simple manner, by identifying areas in our program where several independent operations are performed serially. For instance, the most costly portion of the solver, the loop which identifies child positions for a given position, could benefit from running these in parallel. We suggest looking closer at the `pthreads` library for doing light weight parallelism like this.

11.4.2 External storage

On large puzzles, our solver program quickly runs out of memory, which either results in termination of the program, or the program starts using virtual memory. The efficiency of the program is reduced dramatically when it starts using virtual memory, we measured a slowdown by a factor of around 100.

We could reduce the slowdown factor significantly by designing our own scheme for using disk as secondary storage, instead of letting the operating system decide the portion of memory to store on disk. The BFS algorithm which uses sequential traversal of queues, can be implemented efficiently using disk storage. The challenge lies in keeping track of the set previously visited positions, which is typically accessed in a more random manner.

It remains to investigate efficient ways for implementing A* with disk usage.

11.4.3 GPU

GPUs (graphical processing units) with their massive parallelism have been used with success in recent years in highly calculation-intensive tasks. Can our domain benefit by utilizing the additional computing power a GPU offers?

11.5 Open problems

We conclude this chapter by posing some unsolved problems we have identified.

Currently, the permutation rank is our most memory efficient way of representing individual positions. Does it exist a more efficient representation which can be calculated efficiently for every position in every puzzle of reasonable size?

Is it possible to apply all our algorithms, enhancements heuristics and prunings from the solver to the BDD structure?

Is it possible to get a better estimate (upper bound) of the state space size for huge puzzles where it is infeasible to count the number of ways to pack the blocks? The multinomial coefficient evaluated by considering the blocks in the puzzle as a permutation (combinatorial bound) leads to a rather loose upper bound.

Bibliography

[1] Agnar Aamodt, Enric Plaza, *Case-based reasoning: Foundational issues, methodological variations, and system approaches.* Artificial intelligence communications, vol. 7, pp.39-59, 1994.

[2] Victor Allis, *A knowledge-based approach of Connect-Four.* Master's degree, Vrije Universiteit Amsterdam, 1988.

[3] "Atomix 2.13.4" http://blogs.gnome.org/gpastore/2006/01/05/atomix-2134/. Retrieved on 08.06.2010.

[4] Matthew S. Berger, James H. Lawton, *Multi-agent planning in Sokoban.* Lecture notes in computer science, volume 2883, pp. 360-375, 2007.

[5] "Bricks Game Home Page", http://www.bricks-game.de/. Retrieved on 18.06.2010.

[6] Michael T. Cox, Anita Raja, *Metareasoning: A manifesto.* BBN Technical Memo TM-2028, 2007.

[7] "Commonsense reasoning" http://en.wikipedia.org/wiki/Commonsense_reasoning. Retrieved on 23.05.2010.

[8] Joseph C. Culberson, Jonathan Schaeffer, *Efficiently searching the 15-puzzle.* Technical report 94-08 (unpublished), 1994.

[9] Stefan Edelkamp, Frank Reffel, *OBDDs in heuristic search.* Lecture notes in computer science, volume 1504, pp. 81-92, 1998.

[10] "Fifteen puzzle" http://en.wikipedia.org/wiki/Fifteen_puzzle. Retrieved on 18.06.2010.

[11] Gary W. Flake, Eric B. Baum, *Rush Hour is PSPACE-complete, or "Why you should generously tip parking lot attendants".* Theoretical Computer Science, volume 270, pp. 895-911, 2002.

[12] Dario Floreano, Claudio Mattiussi, *Bio-inspired artificial intelligence.* MIT Press, 2008.

[13] Markus Holzer, Stefan Schwoon, *Assembling molecules in Atomix is hard.* Technical report TUM-I0101, Technische Universität, München, 2001.

[14] Edward Hordern, *Sliding piece puzzles - Recreations in mathematics vol.4.* Oxford University Press, 1986.

[15] Tor Gunnar Houeland, *Reuse of past games for move generation in computer Go.* Master's Thesis, NTNU, 2008.

[16] Falk Hüffner, Stefan Edelkamp, Henning Fernau, Rolf Niedermeier, *Finding optimal solutions to Atomix.* Lecture notes in computer science, vol. 2174, pp.229-243, Springer-Verlag, 2001.

[17] Andreas Junghanns, *Pushing the limits: new developments in single-agent search.* PhD thesis, University of Alberta, Department of Computing Science, 1999.

[18] Andreas Junghanns, Jonathan Schaeffer, *Sokoban: Enhancing general single-agent search methods using domain knowledge.* Artificial Intelligence, volume 129, pp. 219-251. 2001.

[19] Donald E. Knuth, *The art of computer programming, volume 1: Fundamental Algorithms.* Third edition, Addison Wesley, 1997.

[20] Donald E. Knuth, *The art of computer programming, volume 4, pre-fascicle 1B: Binary decision diagrams.* Addison Wesley, 2009.

[21] Donald E. Knuth, *The art of computer programming, volume 4, pre-fascicle 2B: Generating all permutations.* Addison Wesley, 2004.

[22] Richard E. Korf, Larry A. Taylor, *Finding optimal solutions to the twenty-four puzzle.* Proceedings of the national conference on artificial intelligence (AAAI-96), pp.1202-1207, 1996.

[23] Richard E. Korf, Ariel Felner, *Disjoint pattern database heuristics.* Artificial Intelligence, volume 134, January 2002, pp.9-22, 2002.

[24] Richard E. Korf, Peter Schultze, *Large-scale parallel breadth-first search.* Proceedings of the 20th National Conference on Artificial Intelligence (AIII-2005), pp. 1380-1385, 2005.

[25] Jim Leonard, "JimSlide", http://statlerandwaldorf.org/jimslide/ . Retrieved on 20.03.2010.

[26] François Van Lishout, Pascal Gribomont, *Single-player games: Introduction to a new solving method combining state-space modelling with a multi-agent representation.* Proceedings of the 18th Belgium-Netherlands Conference on Artificial Intelligence (BNAIC'2006), pp. 331-337, 2006.

[27] George F Luger, *Artificial intelligence - Structures and strategies for complex problem solving.* Addison-Wesley, 2005.

[28] "Rush Hour - Play Me", http://www.puzzles.com/products/RushHour/RHfromMarkRiedel/Jam.html. Retrieved on 18.06.2010.

[29] Stuart Russell, Eric Wefald, *On optimal game-tree search using rational meta-reasoning*. Proceedings of the 11th international joint conference on Artificial intelligence - Volume 1, pp. 334-340, 1989.

[30] Jerry Slocum, Dic Sonneveld, *The 15 puzzle*. The Slocum Puzzle Foundation, 2006.

[31] "Sokoban", http://en.wikipedia.org/wiki/Sokoban. Retrieved on 18.06.2010.

[32] Ruben Spaans, *Solving sliding-block puzzles*. Specialization project at NTNU, 2009.

[33] Ken'ichiro Takakashi, http://www.ic-net.or.jp/home/takaken/e/soko/index.html. Retrieved on 20.03.2010.

[34] Frank Takes, *Sokoban: Reversed solving*. Bachelor thesis, Leiden University, 2003.

[35] Wieger Wesselink, Hans Zantema, *Shortest solutions for Sokoban*. Proceedings of the 15th Belgium-Netherlands Conference on Artificial Intelligence (BNAIC'2003), pp. 323-330, 2003.