Solving sliding-block puzzles

Ruben Spaans

December 15, 2009

Abstract

We take a look at the complex domain of sliding-block puzzles, which offers significant challenges for the field of artificial intelligence. By analysing the properties of this domain, as well as similar domains where more published literature exists, we develop new domain-specific methods in an attempt to overcome the combinatorial explosion of this domain. We implement a sliding-block puzzle solving program that uses the traditional search algorithms BFS, A* and IDA* in combination with our domain-specific enhancements. We evaluate the performance of our program against a state-of-the-art implementation of BFS, and show that we can reduce the search work by several orders of magnitude using domain-specific enhancements. We conclude our work by suggesting new techniques and areas that can be researched in order to further combat the complexity of solving sliding-block puzzles.

Contents

1 Introduction **11**
- 1.1 Goal . . . 11
- 1.2 Background and motivation . . . 12
- 1.3 Organisation of this report . . . 12

2 Sliding-block puzzles **13**
- 2.1 Description . . . 13
- 2.2 Variants . . . 15
  - 2.2.1 $n \times m - 1$-puzzle . . . 15
  - 2.2.2 Rush Hour . . . 15
- 2.3 Implementations of sliding-block puzzles . . . 16
  - 2.3.1 Klotski . . . 16
  - 2.3.2 Supersliders . . . 16
  - 2.3.3 Bricks . . . 17
  - 2.3.4 Physical variants . . . 17
- 2.4 Definition of our problem domain . . . 18
- 2.5 Test suite . . . 18
- 2.6 Formal properties of the problem domain . . . 20
  - 2.6.1 Search space size . . . 20
  - 2.6.2 Average branching factor . . . 23
  - 2.6.3 Reversible moves and parity . . . 24
  - 2.6.4 Solution length . . . 24
- 2.7 Play strategies . . . 26
- 2.8 Sliding-block puzzles in literature . . . 26
- 2.9 Previous solving attempts . . . 27
  - 2.9.1 JimSlide . . . 27
  - 2.9.2 Our old sliding-block puzzle solving program . . . 28
  - 2.9.3 Comparison between our old solver and JimSlide . . . 30

3 Similar puzzles **31**
- 3.1 Introduction . . . 31
- 3.2 Sokoban . . . 32
  - 3.2.1 Properties . . . 32

3.2.2 Status of solving 33
3.3 15-puzzle, or $n \times m - 1$-puzzle 34
3.3.1 Properties 34
3.3.2 Status of solving 35
3.4 Rubik's cube 36
3.4.1 Properties 37
3.4.2 Status of solving 37
3.5 Summary and comparison of the problem domains 37

4 Solving sliding-block puzzles 41
4.1 Uninformed algorithms 42
4.1.1 Random walk 42
4.1.2 Breadth-first search 42
4.1.3 Depth-first search 42
4.1.4 Depth-first iterative-deepening (DFID) 42
4.1.5 Bidirectional search 43
4.2 Informed algorithms 43
4.2.1 A* 43
4.2.2 IDA* (Iterative-deepening A*) 44
4.2.3 SMA* (simplified memory-bounded A*) 44
4.3 Algorithmic enhancements 45
4.3.1 Transposition table 45
4.3.2 Move ordering 46
4.3.3 Weighted heuristics 46
4.3.4 Pattern databases 46
4.3.5 Macro moves 48
4.3.6 Endgame databases 49
4.3.7 Relevance cuts 49
4.3.8 Pattern search 49
4.4 Domain-specific enhancements 50
4.4.1 Pruning based on properties of positions 50
4.4.2 Don't allow certain moves 51
4.4.3 Don't allow the master block to move away from the goal 52
4.4.4 Assume that some blocks are never used 52
4.4.5 Cleanup after running out of memory 52
4.4.6 Subboard (local) solving 53
4.4.7 Plan with subgoals 53
4.4.8 Post-processing non-optimal solutions 54
4.5 Parallelism 54
4.6 Memory hierarchy 54
4.7 What we will implement 55

5 The implementation 59

5.1 Description of the program 59
5.2 Implementation of heuristics and improvements 62
5.2.1 The A*/IDA* heuristic function 62
5.2.2 Pruning the search space 63

6 Results and analysis 67
6.1 How the tests were run 67
6.2 The puzzles we solved 68
6.2.1 Analysis of the solved puzzles 69
6.3 The puzzles we didn't solve 72
6.3.1 Analysis of the unsolved puzzles 73
6.4 The efficiency of the algorithms, heuristics and improvements 75
6.4.1 Comparison of algorithms 75
6.4.2 The heuristic function 76
6.4.3 Pruning: Space value 76
6.4.4 Pruning: Resting blocks 79
6.4.5 Pruning: Never move the master block away from its goal 80
6.4.6 Plan with multiple subgoals 81
6.4.7 Queue management 82
6.5 Summary 82

7 Future work 87
7.1 Improved heuristic function 87
7.1.1 Pattern database 87
7.2 Transposition table for IDA* with more features 88
7.3 Store the solution 89
7.4 Post-processing 89
7.5 Considering moves instead of steps 89
7.6 Optimisations 89
7.6.1 Speed 89
7.6.2 State representation using permutation rank 90
7.6.3 State representation using a trie 91
7.7 Macro moves 91
7.8 Parallelism 91
7.9 Further potential for improvements 91

8 Conclusion and summary 93

9 Acknowledgements 95

A Test suite 97

List of Figures

2.1 A sliding-block puzzle 13
2.2 The Pennant puzzle 14
2.3 Different ways to define a move 15
2.4 The end-configuration of the 15-puzzle 15
2.5 Example of a Rush Hour puzzle 16
2.6 The puzzle Nostalgy from Bricks VI. 17
2.7 The Devil's nightcap puzzle 21
2.8 Example of building a position from a string 22

3.1 A sample puzzle from Sokoban. 32
3.2 Some examples of deadlocks in Sokoban. 33
3.3 Rubik's cube. 36

5.1 Screenshot of our solver, options screen 60
5.2 Screenshot of our solver, solving the Triathlon puzzle 61

List of Tables

2.1 Known upper bound for the number of steps 20
2.2 Bounds of search space sizes and branching factors 25

3.1 Search space data for other puzzles 39

4.1 Using pattern databases on The Devil's nightcap 48

6.1 Summary of solved puzzles by algorithm 68
6.2 The puzzles we solved, number of nodes expanded. 69
6.3 The puzzles we solved, solution length in steps 70
6.4 The puzzles we didn't solve 72
6.5 Number of nodes expanded, by algorithm 75
6.6 The space value effect on Forget-me-not 77
6.7 The space value effect on Isolation 77
6.8 The space value effect on The Devil's nightcap 78
6.9 Search space and search work savings 78
6.10 The effect of the resting blocks pruning for Hyperion 79
6.11 Search space sizes after depth n on Ithaca 80
6.12 The contribution of the various improvements 85

7.1 Number of bits needed to represent positions 90

Chapter 1

Introduction

1.1 Goal

The main goal of this project is 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. We will then measure the performance and capabilities of our program against the current state-of-the-art software within the domain.

In order to accomplish this goal, the following steps will be done:

First, we will conduct a literature study, where we will discover how others have attempted to solve sliding-block puzzles. In addition, we will look into literature to see how similar single-agent puzzles have been solved.

We will study our problem domain. We will attempt to find properties of the domain, and we will attempt to develop new methods that make use of these properties in order to more efficiently solve sliding-block puzzles.

We will then develop a program that solves sliding-block puzzles. We will use ideas we have found during the literature study, as well as the new methods we have developed during the analysis of the problem domain.

We will select several puzzles from available computer implementations of sliding-block puzzles, and from puzzles mentioned in literature. These will form a test suite. The purpose of the test suite is to evaluate the new methods we have developed. The test suite will measure the efficiency of our program in terms of the number of puzzles solved, partial progress made on puzzles we don't solve, the amount of computational resources spent in order to find a solution and the length of the solution we found. We will measure each improvement in our program to find out how much it contributed towards

solving new puzzles and reducing the computational resources needed in order to solve them.

At last, we will document this process in this report, and evaluate the outcome.

1.2 Background and motivation

Solving sliding-block puzzles represent a significant challenge. Similar, but less complex problems have been attempted solved using traditional methods, like exhaustive search.

These methods fail on more complex domains. We consider sliding-block puzzles to be a complex domain. We hope that research done on complex domains like this can enhance our ability to solve single-agent problems and help develop new methods that can be used in the field of artificial intelligence, and be a benefit for real-world problems.

Similar puzzles, like the 15-puzzle, Sokoban and Rubik's cube have been researched for many years. However, no research has been done on sliding-block puzzles using other methods than exhaustive search. We think that this domain deserves to be worked on, as we think this problem is both fun, interesting and hard.

1.3 Organisation of this report

Chapter 2 gives a description of the domain of sliding-block puzzles, what kinds of puzzles we are trying to solve, properties of the problem, suggestions on how to solve puzzles, list of computer implementations and a description of previous solving efforts.

In chapter 3 we give descriptions of some similar problems, the efforts that have gone into solving them, and properties of these problems.

Chapter 4 is a discussion of different algorithms and methods and how they can be applied to sliding-block puzzles.

In chapter 5 we describe the resulting implementation of the sliding-block puzzle solver.

Chapter 6 contains the results from running our solver on the chosen puzzles, and an evaluation of the results.

In chapter 7 we list some areas in this domain where further research can be done, and we come with suggestions for future improvement.

Chapter 2

Sliding-block puzzles

2.1 Description

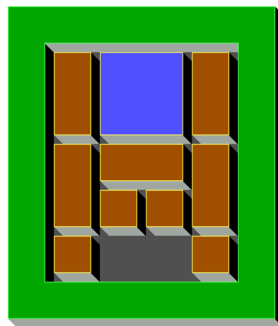


Figure 2.1: A sliding-block puzzle

A sliding block puzzle consists of blocks of possibly different sizes. A block consists of at least one or more connected unit squares. A block can slide in one of the 4 directions (up, down, left, right) if there is enough space. The objective is to reach a certain end-configuration. This end-configuration can require anywhere from one block to all blocks in the puzzle to be in a certain position. Often, the goal is to move one certain piece to a final position. Figure 2.1 shows the Forget-me-not puzzle, where the objective is to move the 2×2 block to the middle of the bottom row.

The first known sliding-block puzzle was the 15-puzzle, which was invented by Noyes Chapman, and became popular in 1880 [21]. The puzzle consists of 15 square tiles with the numbers from 1 to 15 on them within a 4x4 frame. The purpose of the puzzle is to arrange the numbers from 1-15 in order.

A puzzle named *Pennant puzzle*, which is a more typical example of the kind

puzzles we are going to look at, was patented in 1907. It contains blocks of different shapes, including $1 \times 1, 2 \times 1$ and 2×2. Figure 2.2 shows the starting configuration and the goal configuration of the Pennant puzzle.

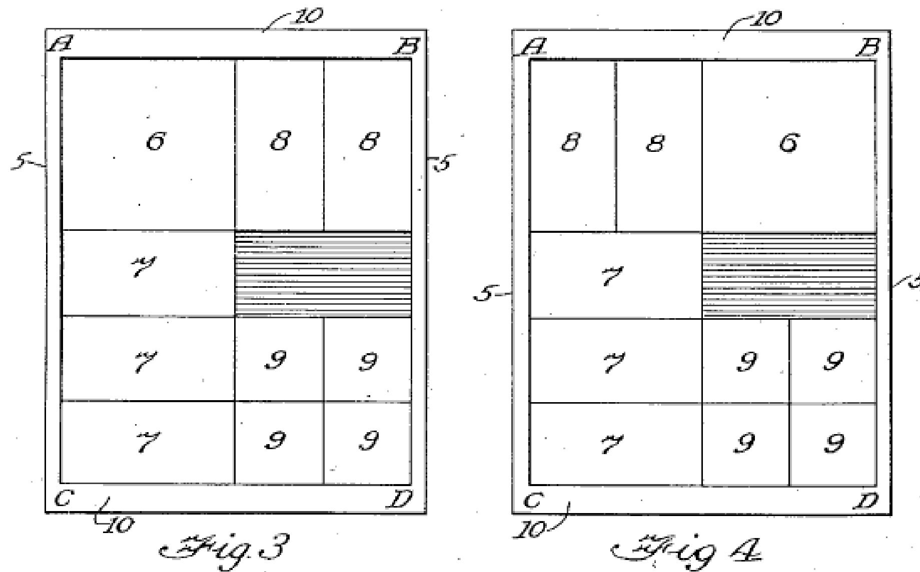


Figure 2.2: The Pennant puzzle. The left figure shows the starting configuration. The right figure shows the solved configuration.

Hordern [6] gives four possible ways for defining a transition from one position to another.

1. Slide one piece only in any direction or combination of directions. The piece may be slid any permissible distance without lifting and without rotating. See figure 2.3 a) for an example of moving a piece around a corner.

2. Slide one piece only in any one direction. The piece may be slid any permissible distance without lifting or rotating. See figure 2.3 b).

3. Slide any number of pieces together as a group without lifting or rotating. See figure 2.3 c) for an example where two pieces move together.

4. Slide one piece in any orthogonal direction, one unit (the same distance as the side of one 1x1 square). See figure 2.3 d).

Throughout in this report we will use the terms *move* and *step* when we mean alternatives 1 and 4, respectively. *Moves* are most commonly used, and is used by the implementations mentioned in this chapter. We will use *steps* in this project.

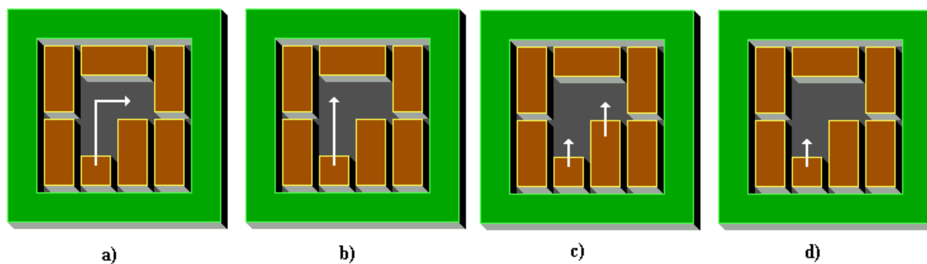


Figure 2.3: Different ways to define a move

2.2 Variants

Some variants will be mentioned here, all of which are specializations of the main definition.

2.2.1 $n \times m - 1$-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 they appear in increasing order.

The 15-puzzle is a special case of this puzzle. Common variants used in research are the 14-puzzle (3×5), the 19-puzzle (4×5) and the 24-puzzle (5×5). Figure 2.4 shows a solved configuration of the 15-puzzle.

Figure 2.4: The end-configuration of the 15-puzzle

2.2.2 Rush Hour

The Rush Hour puzzle is a variant with additional constraints. Each block can only move either horizontally or vertically The goal is to move a particular block out of the grid via a gap in the wall.

This puzzle has been released as a toy. In these versions, the board size is 6×6 and the blocks are 1×2, 2×1 (cars) or 1×3, 3×1 (trucks). The block's movement direction is the same as the direction they extend in.

Figure 2.5 shows an example instance of Rush Hour.

Figure 2.5: A puzzle of Rush Hour. The objective is to slide the red car out of the gap in the right wall.

2.3 Implementations of sliding-block puzzles

In this section, we will mention some implementations of sliding-block puzzles, both in form of computer programs, version playable via web pages and puzzles released as toys.

2.3.1 Klotski

Klotski is one of many games in the Microsoft Entertainment Pack for Windows, released in 1990. It contains 24 levels. All levels have one master block inside an inner frame. This frame has one or more barriers. A barrier may only be removed if all sections of a barrier has been adjacent to some part of the master block at some time. The destination square of the master block is always in the area outside the inner frame.

2.3.2 Supersliders

Supersliders contained around 50 puzzles, but it is no longer available. Some of these puzzles can be found on the Brainyday website [23]. Also, most of the remaining puzzles from that game can be found at the Puzzleworld website [22].

2.3.3 Bricks

Bricks [19], originally programmed by Andreas Rottler, consists of 7 main games with 48 levels each, bimonthly competitions since 1998 (over 110 levels), in addition to at least 6 fan-made games with 48 levels each (Custom Bricks series and Bricks Holiday series). Bricks adds a multitude of new elements to the standard ones. See figure 2.6 for a screenshot of the level *Nostalgy* containing all the element types available in the game[1]. Some new elements include magnets that stick to each other, keystones that convert blocked squares into blocks and holes that destroy blocks.

Figure 2.6: The puzzle Nostalgy from Bricks VI.

2.3.4 Physical variants

Hordern [6] contains a catalogue of more than 270 puzzles. The majority of these puzzles are of the sliding-block type, but a few other types are included. Most, if not all of these have been released as toys.

[1] For an interactive tutorial of all elements, see the following page on the Bricks website: http://www.bricks-game.de/html/tutor0.html

2.4 Definition of our problem domain

Some of the computer implementations mentioned in section 2.3 support extra elements. If we were to support all of these, our implementation (especially the move generator) would be very complex

The domain of the Bricks computer game is very complex and out of scope for this project. The generalised problem without any of the special elements from Bricks can be shown to be PSPACE-complete. The implementation, especially the move generator, would be very complex if we included full support for Bricks. The domain we are going to look at is defined as follows:

- Include only regular blocks, frames and space. As we will see in section 2.9.2, a puzzle containing only these kind of blocks leads to a very compact representation in memory, and the search space have some nice properties when all moves are reversible, see section 2.6.3. Many of the elements in Bricks lead to irreversible moves. Blocks are allowed to be concave and to have holes, but a single block must not be disjoint.

- Blocks of the same size and shape are normally considered indistinguishable. We will support puzzles containing distinguishable blocks of the same size and shape. For instance, the master block needs to be distinguishable from other blocks of the same shape.

- We will not support restricting the movement of blocks to only one of the horizontal or vertical directions, as in Rush Hour.

- We will use *steps* when doing a transition from one position to another. We will show later the search space has some very nice properties when using breadth-first search.

2.5 Test suite

We have a test suite containing several puzzles. The puzzles are taken from Bricks (see section 2.3.3), Hordern [6] and the Brainyday website (see section 2.3.2). Some are modified slightly to fit our domain. Many puzzles from Bricks containing special elements were changed to only contain standard blocks. See appendix A for illustrations. The solvability of some puzzles are known already, see section 2.9 for an overview.

The puzzles we have picked are chosen such that they are meant to test different aspects of our program. The list of puzzles and why they are picked follows.

- **Easy, Forget-me-not**: Trivial puzzles that are included to confirm that our program works.
- **Isolation, San, Climb game 15D**: Puzzles that are known to be solvable with BFS. We want to measure the savings we can achieve using new methods.
- **Hyperion, The Devil's nightcap**: Puzzles containing special goal conditions. The goal condition of The Devil's nightcap specifies the position of two blocks, and for Hyperion the requirement is to move the three vertical blocks one step down. These conditions can be hard to code into a heuristic function. We want to measure the impact these puzzles have on the efficiency of a search algorithm using A*.
- **American pie, Still easy, Warmup**: Easy puzzles with an infeasibly large search space. These puzzles will be used to measure the performance of A* and new methods we can come up with.
- **Rose, Triathlon:** Medium difficulty puzzles with an infeasibly large search space. We will use these puzzles to measure the performance of A* and our new methods, or eventually measure how much progress we can make if we don't solve these puzzles.
- **Magnolia, Apple, Get ready, Paragon 1FG, Turtle**: Difficult puzzles with an infeasibly large search space. We want to measure how much progress we are able to make into solving these puzzles. We do not expect to solve all of these puzzles in this category.
- **Chair, Little sunshine, Corona, Thunder, Schnappi, Climb pro 24, Salambo, Sunshine, Ithaca**: All these puzzles have a search space which is too large to handle for traditional search algorithms. We want to try out our new methods on these puzzles, and measure how much progress we can make into solving them. We don't expect to solve these puzzles, because of their complexity in terms of search space size and average branching factor.

As most of these puzzles are part of available computer games with highscore lists, we know in advance an upper bound on the number of moves needed to solve the puzzles. However, many of the puzzles had to be changed to fit our slightly simplified domain. Also, every implementation uses *moves* while we use *steps* to measure the length of a solution. The Bricks website [19] lists the currently known lowest number of moves needed for each puzzle. In addition, the lowest number of steps while still using the same lowest number of moves is listed. This information gives us an upper bound for the number of steps needed for each puzzle. These are listed in table 2.1. Some puzzles had to be changed in a way such that the result from the website is no longer guaranteed to be an upper bound. These are not listed in this

Table 2.1: Known upper bound for the number of steps

Puzzle name	Steps	Puzzle name	Steps
Easy	33	San	445
Forget-me-not	118	Corona	363
Ithaca	166	Thunder	532
American pie	143	Magnolia	211
Still easy	84	Schnappi	1069
Rose	89	Salambo	1547
Little sunshine	369	Sunshine	713
Chair	149	Paragon 1FG	333
Isolation	280	Warmup	160
Hyperion	176	Get ready	309

table. For some other puzzles where the number of steps at an intermediate stage of the solution was needed, or no solution length in steps was known, we have used the number of steps we have achieved ourselves in the game. This group of puzzles include Isolation, Warmup and Get ready. The upper bound of San includes steps needed for the full puzzle. Our version consists only of the first chamber, which is the hard part of the full puzzle. The mean solution length of the ones we found an upper bound for is 246.

2.6 Formal properties of the problem domain

The problem of determining whether a given sliding-block puzzle has a solution has been shown to be PSPACE-complete[2] [5], even when the objective is to move a given block to a given place and when all blocks are 1×2 rectangles (dominoes).

2.6.1 Search space size

The size of the search space of a particular instance depends on the block shapes and how many there are of each block shape. Consider a puzzle of interior size $n \times m$ (that is, the frame is not included in this size). Let a state be defined by the sequence $b_1, b_2, \cdots, b_k$, where b_i represents either a block or one unused 1×1 square and k is the number of blocks plus the number of spaces in the puzzle. b_i, b_j are allowed to be equal for $i \neq j$. Let $|b_i|$ be the

[2] For a definition of the complexity class PSPACE and the notion of PSPACE-completeness, we refer to chapter 9 of Kleinberg and Tardos [8]

number of unit squares the block occupies. Let the sequence be such that

$$\sum_{i=1}^{k} |b_i| = nm.$$

A position can be constructed from this sequence by the following algorithm:

1. Initialize empty grid of desired size $n \times m$.
2. For each block element b_i in the sequence do:
3. a) Find the first empty cell in the grid using linear scan
   b) Insert block b_i so that its upper left coincides with this empty cell
   c) If block is inserted on top of another block, on the frame or outside the grid, the block sequence is illegal.

A position that is legal according to the procedure above need not be a reachable position. For instance, the 3×1 block in The Devil's nightcap (see figure 2.7) is not able to leave its row; it cannot move up or down since the puzzle contains only two spaces.

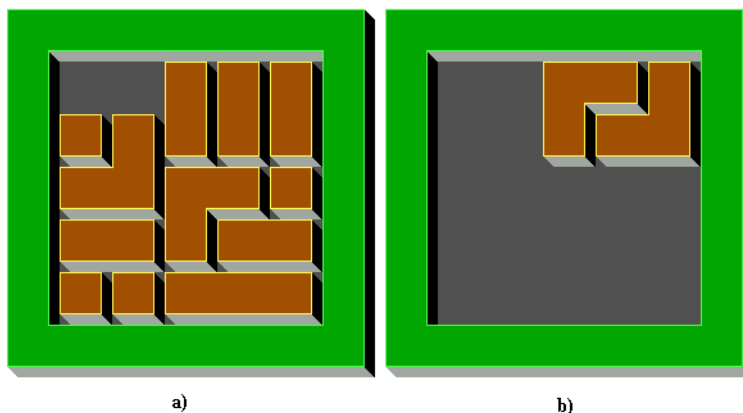


Figure 2.7: "The Devil's nightcap". a) shows the initial position, b) shows the requirement for winning the puzzle: The two blocks shown must be at the indicated positions.

Figure 2.8 shows how to construct a position from a sequence of blocks.

The upper bound for the search space for a given puzzle is therefore the number of distinct permutations of the sequence. For n elements where $\alpha_1, \alpha_2, \ldots, \alpha_k$ of them are mutually similar, we can use the standard formula from combinatorics for the number of permutations [18]:

$$P(n; \alpha_1, \ldots, \alpha_k) = \frac{n!}{\alpha_1! \alpha_2! \cdots \alpha_k!}.$$

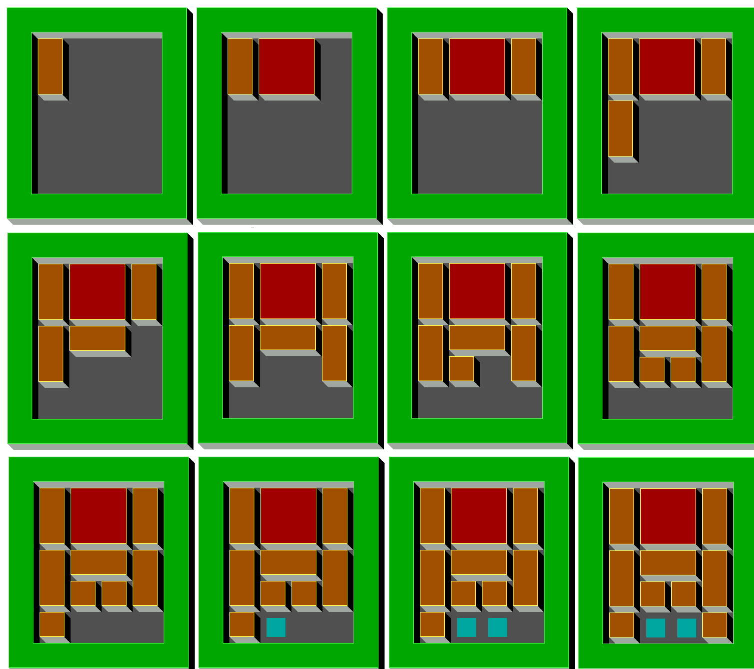


Figure 2.8: Consider a position with spaces (a) and blocks of sizes 1×1 (b), 1×2 (c), 2×1 (d) and 2×2 (e), each of these is represented by the letter in parentheses. Let the string *deddcdbbbaab* represent a position. Starting from the top left and going right, each diagram shows one block added one at a time. A block is added at the topmost unused cell (or leftmost if there is a tie). In the bottom row, spaces inserted into the position are shown as cyan squares. With the same blocks and letters, we see that a string starting with for instance *bbbe* will represent an illegal position, as the right half of the 2×2 block would be inserted on top of the right wall. Conversely, a string can be constructed from a position by doing the process in reverse.

For instance, a puzzle with k distinct blocks and l spaces has an upper bound of $\frac{(k+l)!}{l!}$ distinct positions.

A tighter bound can be found by counting the number of ways to pack the pieces into the board. We will call it the *packing bound*. Our basis algorithm is backtracking where we try every remaining block at the next unfilled cell in a partially filled-in grid. Our next step is to use memoization, using a vector holding information about the number of blocks left of each type, and a bitmask containing information about whether the next cells are occupied. This allows us to efficiently calculate the number of possible layouts for puzzles that have up to 10^{17} different layouts. For larger puzzles, our method is still not sufficient, as the number of different states in our algorithm becomes too large.

Table 2.2 shows the actual search space sizes, packing bounds, combinatorial bounds and estimated average branching factor for the different puzzles. In some cases they were not feasible to calculate, and will be shown by a dash in that cell. Some remarks: Some additional constraints were enforced on the bounds for *The devil's nightcap*: The 3×1 block never leaves the bottom row. We did not do any improved calculations for *Hyperion*, even though the three topmost blocks never leave their respective corridors entirely.

The median search space upper bound (using the best bound we have for each puzzle, which is the exact size for the smallest puzzles) is 10^{15}.

2.6.2 Average branching factor

The average branching factor for a puzzle is defined as the average of the number of neighbouring states for every unique position in the search space, or the average *degree* of each node in the search space graph. In the cases where we were able to do a full search in the puzzle, we have calculated the exact average branching factor. This is the case for Easy, Forget-me-not, Isolation, Hyperion, Climb game 15D and The Devil's nightcap. For the other puzzles, we have included the average over all positions we examined in a breadth first search before we ran out of resources or time. For puzzles with a very large search space we end up sampling only a very small part of it in a very non-random way. However, this is of little consequence to us, as we use these numbers for high-level reasoning and for comparing different puzzles. We expect the relative order of the actual branching factors between puzzles to be not too far off.

However, a high branching factor doesn't necessarily imply difficulty of searching. Triathlon has one of the smallest branching factors, but it's not feasible to solve with standard BFS. Also, San, which is solvable with BFS, has a higher branching factor than many puzzles which are not feasible to solve

with BFS. In this case, we are lucky that that solution occurs at a relatively short distance from the starting position. The median of the average branching factors of the puzzles we've considered is 5.43.

2.6.3 Reversible moves and parity

For any puzzle, the search space is an undirected graph, because a block can always move back the way it came from; the move is *reversible*. The graph can contain cycles.

The fact that the search space graph is undirected has a nice benefit when doing a breadth-first search: we don't need to keep track of the complete history of previously visited nodes. Let search depth i be all the positions that are first visited in iteration i of a breadth-first search. It suffices to keep information about the two previous search depths from the current depth the search is processing. Say we have already stored all positions at depth n and want to generate depth $n + 1$. Let p_1 be a position at depth n and p_2 be position reachable in one move from p_1. Since moves are reversible, position p_2 couldn't have occurred at depth $n - 2$ or earlier. If it had occurred at depth $m \leq n - 2$, p_1 would have occurred no later than depth $m + 1 \leq n - 1$. Hence we need not store positions from depth $n - 2$ or less in order to check if a position is already visited. This argument holds both when considering moves and steps. This result has previously been proven by the Bricks community [24].

We get another benefit when using steps rather than moves in a breadth-first search. Moving a block one step always changes the parity given by

$$\sum_i (x_i + y_i) \pmod 2$$

where x_i, y_i is the position of block i. Any step changes one coordinate by ± 1. Hence, every positions in the same search depth has the same parity. Hence, when checking if a position has been visited before, we only need to check against search depths having positions of the same parity. When combining parity and reversibility, a breadth-first search at depth n generating the queue for depth $n+1$ only needs to do duplicate checks against depths $n-1$ and $n + 1$. A breadth-first search using moves instead of steps has to check for duplicates against depth n as well.

2.6.4 Solution length

Since the problem is PSPACE-complete, it follows that the solution length is not bounded by any polynomial. Hence, it is expected that the solutions

Table 2.2: Known search space sizes, bounds and estimated average branching factor for the different puzzles

Puzzle name	Actual search space size	Packing bound	Combinatorial upper bound	Est. avg.bf
Easy	$1.4 \cdot 10^3$	$1.4 \cdot 10^3$	$2.0 \cdot 10^3$	5.00
Forget-me-not	$2.6 \cdot 10^4$	$6.6 \cdot 10^4$	$4.2 \cdot 10^5$	3.23
Ithaca	-	-	$2.1 \cdot 10^{30}$	18.22
American pie	-	$1.1 \cdot 10^{17}$	$3.8 \cdot 10^{19}$	9.65
Still easy	-	$1.0 \cdot 10^{11}$	$4.2 \cdot 10^{11}$	5.38
Rose	-	$7.0 \cdot 10^{13}$	$2.9 \cdot 10^{15}$	4.14
Little sunshine	-	-	$7.7 \cdot 10^{54}$	23.80
Chair	-	-	$1.9 \cdot 10^{19}$	9.80
Isolation	$5.1 \cdot 10^6$	$1.5 \cdot 10^7$	$1.0 \cdot 10^9$	5.40
Hyperion	$1.9 \cdot 10^8$	$1.0 \cdot 10^{10}$	$3.5 \cdot 10^{14}$	3.56
San	-	$1.1 \cdot 10^{12}$	$6.2 \cdot 10^{15}$	6.54
Corona	-	-	$2.0 \cdot 10^{32}$	10.18
Thunder	-	-	$1.1 \cdot 10^{27}$	11.37
Triathlon	-	$2.0 \cdot 10^{13}$	$6.9 \cdot 10^{15}$	3.50
Magnolia	-	$1.6 \cdot 10^{15}$	$8.1 \cdot 10^{18}$	4.96
Turtle	-	-	$3.6 \cdot 10^{30}$	6.16
Apple	-	-	$7.8 \cdot 10^{30}$	5.83
Schnappi	-	-	$1.7 \cdot 10^{39}$	6.88
Salambo	-	-	$2.9 \cdot 10^{47}$	20.56
Sunshine	-	-	$6.6 \cdot 10^{42}$	8.65
Paragon 1FG	-	$3.7 \cdot 10^{12}$	$2.7 \cdot 10^{15}$	5.45
Warmup	-	$6.2 \cdot 10^{11}$	$7.3 \cdot 10^{13}$	3.49
Get ready	-	$3.4 \cdot 10^{17}$	$2.1 \cdot 10^{20}$	3.69
Climb game 15D	$1.8 \cdot 10^9$	$4.9 \cdot 10^9$	$2.8 \cdot 10^{12}$	3.90
Climb pro 24	-	$1.9 \cdot 10^{16}$	$2.1 \cdot 10^{21}$	3.99
The Devil's nightcap	$5.0 \cdot 10^5$	$1.9 \cdot 10^6$	$1.5 \cdot 10^8$	3.06

lengths can be very long, even for relatively small puzzles. A good example of this phenomenon is the puzzle *The Devil's nightcap* (see figure 2.7). The size is 5×5 and the optimal solution needs 888 steps.

2.7 Play strategies

This section lists a few strategies used by humans when trying to solve puzzles.

- Make an overall plan for how to solve the puzzle. Identify blocks which are likely to be an obstacle. Find the most likely path that the master block will take to the goal. This plan can include several stages, which may involve repacking of several of the problematic blocks. On tight puzzles with little space, there can be a lot of such stages.
- Keep the spaces close to each other, and keep them near the master block.
- Keep blocks that are easy to move around near the master block. In most cases this will be blocks of sizes 1x1, 1x2, 2x1 and 2x2.
- Pack large blocks or irregularly shaped blocks away in corners or other places where they won't interfere with the master block.

When attempting to solve a puzzle in the least number of moves (steps), the above strategies are applied to a lesser degree. Instead, a number of potentially short paths are considered, and then brute force is applied: the user tries as many promising moves as possible.

2.8 Sliding-block puzzles in literature

There is little literature on the subject of solving sliding-block puzzles puzzles, except the special case of the 15-puzzle. To our knowledge, the only source is Hordern [6]. They mention two problems within the domain:

A. Can you get from one configuration to another?

B. What is the least number of moves required to get there?

They say that the mathematics of sliding piece puzzles is either moderately well known (in the case of the 15-puzzle) or is almost completely unknown. In general, as far as they know, problem A is only solved by solving problem B, and the only known method for solving problem B is systematic search. The challenge of loops in the search graph is mentioned, with the conclusion that

classic tree depth-first search techniques do not apply easily. Breadth-first search is their recommended algorithm.

2.9 Previous solving attempts

To our knowledge, no report of any attempt at making programs to automatically solve sliding-block puzzles exist in published literature.

Looking beyond published literature, hobbyist projects exist. The projects we know about are based on breadth-first search. Some of them use additional enhancements are very specific and are aimed at solving only one specific puzzle. They have published their results on the internet, mostly via discussion forums.

Rottler [24] mentions that it is possible to find the optimal solution to Magnolia, using a modified exhaustive search algorithm. He doesn't state the nature of the modification. According to [24], it is also possible to solve Get ready and Apple using a solver program.

Pflug [15] used another enhancement to be able to solve Paragon 1I (a slightly more restricted version of Paragon 1FG). He attempted to solve it using BFS, but ran out of memory after 111 moves. Whenever the search ran out of memory, positions were removed based on very specific criteria (master block has made little progress, and certain key blocks were in bad spots). Also, his program searched for a key position 11 moves away from the goal, rather than the goal position itself.

The next two paragraphs will describe two known implementations, our old one, and JimSlide, a very efficient BFS solver published on the internet. These two programs can be considered to represent the state of the art of BFS-based solver programs for sliding-block puzzles.

We will compare the results of these programs with the results we get from the new ideas we will be developing in this project. Different aspects, as which puzzles are solved, and the amount of computational work needed to solve the puzzles will be compared.

2.9.1 JimSlide

JimSlide [14] is a program that solves user-specified sliding block puzzles using breadth-first search, and it is capable of utilizing the hard drive. In addition to regular blocks, blocks moving only horizontally and vertically are supported, as are disjoint blocks (a block with disjoint elements which move together as a group).

They use an efficient representation in memory of each position. Each position is encoded into a string where each symbol in the string represents a block. The meaning of this string is explained in section 2.6.1. Let m be the number of unique blocks in a given puzzle, including the empty cell and let n be the number of blocks and empty cells in a puzzle. They use a constant number of bits to represent each symbol and they use the lowest number of bits such that each of the m symbols can be represented. Hence, each element in the string takes $\lceil \log_2 m \rceil$ bits and one position needs $n \cdot \lceil \log_2 m \rceil$ bits.

The algorithm used is breadth-first search with one enhancement: When a new search depth $n + 1$ is generated, only depths $n - 1$ and n are checked for duplicates, in addition to checking for duplicates within the same search depth. This enhancement is valid because the moves are reversible (see section 2.6.3.

When expanding new nodes from depth n, the new nodes belonging to depth $n + 1$ are put in a binary tree. When the tree has exhausted the designated memory, the tree is converted to a sorted linked list (in-place by converting pointers). Then a duplicate check is run against search depths $n - 1$, n and what's currently generated of depth $n + 1$, before storing this list to disk. This duplicate check takes $O(p^2)$ time for one search depth where p is the number of states in the search depth, because each partial list must be checked against each previously generated list from the same search depth.

It is not mentioned which puzzles the program is able to solve. We have tested the program, and it is able to solve puzzles like San and Isolation. We believe that this program and the BFS portion of the resulting implementation in this project will solve the same puzzles.

2.9.2 Our old sliding-block puzzle solving program

During the period 1999-2009 (before the project), we worked on and off (mostly off) on a sliding-block puzzle solver. This program uses BFS, and it uses the same way of encoding a position to a string of block elements as JimSlide. Memory usage optimisation is taken a step further though; given the different blocks and how many of each are used in a puzzle, Huffman coding is used to get the best possible encoding of elements. In addition, the last element is dropped from the string, since it is implied, given the rest of the string.

The program uses *moves* to detect child positions, but it can be changed to use *steps* with one simple modification. The program supports blocks that can move only horizontally and vertically (as in Rush Hour), as well as disjoint blocks.

As JimSlide, the program uses disk storage to store search depths and the BFS queue. The BFS algorithm consists of two phases:

- **Search**: the current queue is processed and all child positions are generated and stored.
- **Delayed duplicate checking:** After all the child positions are generated, they are checked for duplicates against the two previous search depths, and against itself.

During the search phase, memory is divided in two: The queue in the first half, and generated positions in the second half. New portions of the queue are read from disk as needed, and generated positions are sorted and then written to disk as the second half of the memory is filled up, creating a new file. No duplicate checks at all are done at this stage. At the end of this phase, we have generated all positions for the next search depth.

In duplicate check phase, the first step is to sort all the newly generated positions, and to remove the duplicate positions within this search depth. This is done via a disk mergesort. At first, the newly generated search depth consists of m files. Each iteration merges files pairwise, removes duplicate elements in the two files, and produces one new file of twice the size, or slightly less if duplicates are found. After $\lceil \log m \rceil$ iterations, we are left with one file containing distinct positions, where positions are sorted by the binary value of their encoding.

Then, another pass is done in order to check for duplicates against the two previous search depths. Memory is divided in four parts: Portion of search depth $n-2$, portion of search depth $n-1$, portion of current search depth n and the last area is written with the next search queue without duplicates. For each position in search depth n, we check if it exists in depths $n-1$ or $n-2$. If it doesn't, we keep it. Each portion in memory is read/written from disk as needed.

The duplicate check takes $O(kn \log kn)$ time to perform (because of the mergesort), using roughly $2kn = O(kn)$ space where n is the number of positions in the search depth we wish to remove duplicates from. k here is the branching factor of the puzzle.

With these improvements, this program is a very efficient BFS solver. We have previously solved puzzles like Isolation, Hyperion, The Devil's nightcap, Climb game 15D, San and Turtle (where 8 blocks are locked into place) using this program.

2.9.3 Comparison between our old solver and JimSlide

JimSlide and our old program have much in common. In fact, our old solver was heavily inspired by JimSlide. They are both based on breadth-first search, and they use the same basis for the encoding of a position.

Our old program takes memory optimisation one step further, as Huffman coding is used to shorten the memory representation of a position, as well as not storing the last element. In addition, the asymptotic growth of the run-time of the algorithm is improved. Performing duplicate check on n position is done in $O(n \log n)$ with our old program, while JimSlide needs $O(n^2)$ time to perform the same operation. The constant hidden behind the big-O notation is extremely low, so n needs to be very large to experience the quadratic growth of the JimSlide program.

JimSlide is slightly faster than our old program, it is faster by a factor of around 1.4.

Chapter 3

Similar puzzles

3.1 Introduction

As we mentioned in the previous chapter, the literature pertaining to solving this kind of puzzle is very scarce. According to Demaine [4], "there is little theory for analyzing combinatorial one-player puzzles." Nevertheless, there exist cases in literature where one-player puzzles somewhat similar to sliding-block puzzles have been attempted solved. In this section I will present my results of studying this literature, and find out how successful the attempts were. I will look into which algorithms which were chosen, and what enhancements were used. Also, I will compare the properties of these other problems to sliding-block puzzles. Some properties pertain to search space, and include branching factor, length of solution, how many solutions states there are in the search space (there can be as few as one solution), type of graph (tree, undirected graph, directed graph). The discussion on how difficult it is to find tight upper bound heuristics is contained in the subsection "Status of solving" for each game, as this is not a static property of the search space.

Enhancements to known algorithms will be studied in detail. According to Junghanns [7, p.22], "the choice of the right algorithmic enhancement(s) is more difficult and crucial to the performance of the program than choosing the right algorithm." We will keep this advice in mind, and have a focus on finding algorithmic enhancements.

Based on my findings, we will argue for an algorithm to use on sliding-block puzzles.

3.2 Sokoban

Sokoban is a one-player puzzle set in a maze, depicted by walls and open squares in a rectangular grid. The object is to push all stones to designated goal squares. The maze contains a man, whose possible actions are to move to a neighbouring square, and to push a neighbouring stone. The man can only push one stone at a time, and boxes cannot be pulled. Figure 3.1 shows puzzle 1 from the original Sokoban puzzle set.

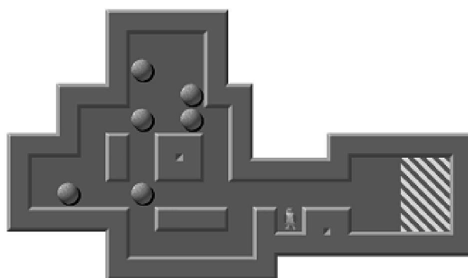


Figure 3.1: A sample puzzle from Sokoban.

3.2.1 Properties

The problem of finding any solution to a Sokoban puzzle is PSPACE-complete [2]. A consequence of this is that solution lengths are unbounded by any polynomial. Of the 90 puzzles in the test suite of Junghanns [7], puzzle 39 is the one that requires the most number of stone pushes to be solved. The shortest known solution needs 674 pushes. The average solution length is 260.

Unlike Rubik's Cube, the 15-puzzle and our sliding-block puzzles, moves in Sokoban are not reversible in general. Hence, the search space is a directed graph. In addition, it is possible to enter an unsolvable state by creating a configuration of stones such that it is impossible to push the stones to the goal. Such a state is called a *deadlock*. See figure 3.2 for several examples of deadlocks.

The search space contains only one solution; the state where every goal square contains a stone. However, a puzzle usually contains several stones, but they are interchangeable. So it does not matter which stone is pushed to a specific goal square.

The search space for a given puzzle can be huge. Before proceeding, one needs to define the neighbours in the search space graph. Junghanns [7] defined a move as a stone push. Two states are equal if the stone configuration is

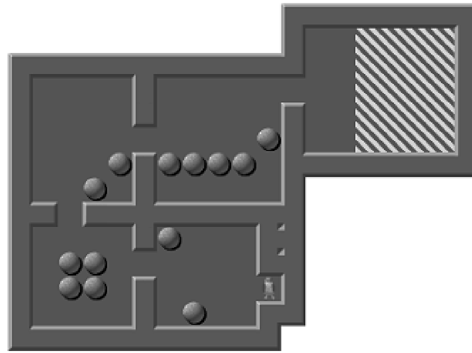


Figure 3.2: Some examples of deadlocks in Sokoban.

equal, and the man positions in the two states can reach each other. In their test suite, the puzzle with the largest state space is puzzle 90, with a state space in the order of 10^{29}. The puzzle has 25 stones and the maze has 181 squares. The median search space size for the puzzles they considered is 10^{18}.

The branching factor is also huge. Puzzle 48 has 34 stones, and each of these can be pushed in at most 4 directions. So in this case, the branching factor is up to 136. It's slightly less in practice, as some stones can be next to walls or other stones. The average branching factor is 12.

3.2.2 Status of solving

To our knowledge, Junghanns [7] is the most successful attempt at solving Sokoban among the ones that are published. Their program *Rolling stone* has solved 60 problems out of a test suite of 90 problems. Out of these 60, 12 were optimally solved. [7] also mentions other solving efforts. The best program they know about can solve 62 problems, but the creators haven't published their results.

The *Rolling stone* program uses the IDA* algorithm, with several enhancements:

- **Transposition table:** Used to avoid cycles and keep track of previously visited states.
- **Move ordering:** Try the most promising moves first. In their program, pushing the same stone as the previous move is tried first. Then, try all optimal moves (moves that decrease the lower bound to the goal state), sorted by stone distance to goal. Then, try all non-optimal moves.

- **Deadlock tables:** Prune moves by checking if parts of a configuration matches an entry in the deadlock table. The deadlock table is a set of subboards containing a deadlock position.
- **Tunnel macros:** When a stone is pushed into a tunnel, push it all the way through.
- **Goal macros:** When a stone is pushed to the goal area, push it all the way to a goal square.
- **Goal cuts:** When it is possible to push a stone to the goal area, don't consider any other moves.
- **Pattern search:** Small searches in partial configurations done in attempt to improve the lower bound heuristic.
- **Relevance cuts:** Restrict the search to not consider move sequences containing unrelated moves.

The quality of the lower bound heuristic function for IDA* varies, but it is close to the actual distance to the solution for a significant number of puzzles in their test suite. However, calculating the lower bound is expensive, $O(n^3)$, or an incremental cost of $O(n^2)$ (n is the number of stones), given that we have already calculated the lower bound for the parent state.

3.3 15-puzzle, or $n \times m - 1$-puzzle

We turn again to the 15-puzzle, a specialization of the sliding-block puzzle. As we recall, the objective is to move the tiles until all tiles in the grid are in order. The standard puzzle size is 4×4 with 15 numbered tiles and one tile missing.

3.3.1 Properties

It is easy to determine whether a given puzzle instance is solvable. Consider the sequence of numbers in the puzzle in the order they occur on the board. Then, the puzzle is solvable if and only if there is an even number of inversions (out-of-order pairs) in the sequence [4]. Likewise, a configuration can reach another if the number of inversions for both configurations have the same parity.

It has been proved via complete breadth-first search that no configuration of the 15-puzzle require more than 80 moves [13].

The exact size of the search space is $\frac{16!}{2} \simeq 10^{13}$, which consists of every reachable state with the same parity. The state space contains only one

solution, but there can be several optimal solution sequences. Different instances (starting positions) belong to the same search space, unlike Sokoban and sliding-block puzzles.

Given the natural generalisation of the puzzle to $n^2 - 1$ ($n \times n$ sized grid with $n^2 - 1$ numbered tiles): It's easy to find any solution, it can be done in polynomial time. Finding the optimal solution (the lowest number of moves) is NP-complete. The required number of moves is $\Theta(n^3)$ in the worst case [4]. The size of the search space is $\frac{(n^2)!}{2}$, where only reachable states are included.

The branching factor varies between 2 and 4, depending on the position of the blank square. The effective branching factor is therefore between 1-3, since undoing the last move is never considered.

3.3.2 Status of solving

As mentioned in the previous section, the 15-puzzle has been solved using breadth-first search, due to Korf [13]. They did a complete breadth-first search of the search space, using parallel processing and storage of nodes to disk. The average solution length across all states was 53 moves.

Culberson [1] uses IDA* with the following improvements:

- **Pattern databases:** This is a means of improving the lower bound heuristic. It contains all solutions to the subproblem of correctly placing N tiles. They called this technique *reduction databases* in their paper.
- **Transposition tables:** Keep track of previously visited states.
- **Endgame databases:** Store all states that are at most N moves away from the goal state. When the search reaches a position in this set, retrieve the distance and stop searching in this branch. They used $N = 25$ in their approach.

Using reduction databases for 8 pieces, they managed to reduce the total number of nodes searched by a factor of over 1700 on a standard test set containing of 100 positions.

Korf [10] has found optimal solutions to ten random instances of the 24-puzzle (5×5). They used IDA* with the following improvements:

- **Manhattan distance heuristic:** Standard heuristic for the $n^2 - 1$ puzzle, sum of the Manhattan distances from each tile to its goal.
- **Linear conflict heuristic:** This applies when two tiles are in their goal row or column, but are reversed relative to their goal positions.

- **Last-two-moves heuristic:** Based on the last moves of the solution, where the blank is returned to its goal position.
- **Corner-tile:** This improvement applies for certain configurations in the corners. These 4 heuristics are combined into an admissible heuristic.
- **Finite-state machine pruning:** Encode the movement in the state space as a finite-state machine such that duplicate nodes are avoided.

We refer to Korf [10] for further details regarding these improvements. Later, they improved their result to solving 50 random instances of the 24-puzzle, as well as improving the performance on the 15-puzzle by a factor of over 2000 [12]. They achieved this using disjoint pattern databases - multiple subgoals whose heuristic values can be added together, while ensuring that the sum is still an admissible heuristic function.

3.4 Rubik's cube

The *Rubik's cube* is a 3-dimensional mechanical puzzle invented in 1974 by Ernõ Rubik. Each face of the cube contains 9 colour stickers, arranged in a 3×3 grid. A face can be twisted around 90 or 180 degrees in any direction. The puzzle is solved when each face shows only one colour.

Figure 3.3 shows the Rubik's cube in a scrambled state.

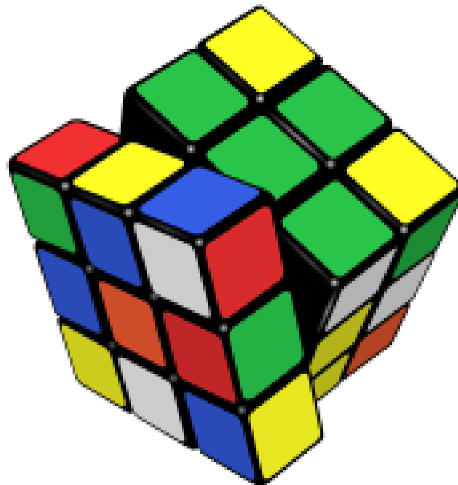


Figure 3.3: Rubik's cube.

3.4.1 Properties

A move is usually defined in either one of two ways:

(i) A face turn of either 90 of 180 degrees.

(ii) A face turn of only 90 degrees.

All the references cited here use move alternative (i), unless noted otherwise.

No instance of the $3 \times 3 \times 3$ cube require more than 22 moves [16] [17]. Also, many positions requiring 20 moves are known, but no position is known yet that requires 21 moves. The median optimal solution length appear to be 18 moves [11]. It is conjectured that no instance needs more than 20 moves.

Given the generalisation to $k \times k \times k$ cubes, the problem of finding the optimal solution is PSPACE-complete [4]. However, determining if a configuration is solvable can be done in polynomial time. Also, a representation of a non-optimal solution can be found in polynomial time.

The search space for the standard $3 \times 3 \times 3$ cube is $8! \cdot 3^8 \cdot 12! \cdot 2^{12}/12 \simeq 4.3 \cdot 10^{19}$ [11]. The search space is an undirected graph since a move can always be undone.

The maximal branching factor is 18, since there are 6 different faces with 3 different rotations each. However, it's not interesting to rotate the same face twice in a row, so the effective branching factor is at most 15.

3.4.2 Status of solving

Korf [11] describes a program that find optimal solutions to random instances of Rubik's Cube. They used IDA* with a lower-bound admissible heuristic function based on pattern databases. The pattern databases contain the exact number of moves required to solve various subgoals. In this case, the database contains the number of moves required to bring subsets of cubies to their correct position. Ten random instances were solved in 16-18 moves. They argue that A* is impractical on this problem, because of the exponential space requirement.

3.5 Summary and comparison of the problem domains

We notice that most solving efforts mentioned used IDA* as the preferred algorithm, in combination with a good admissible lower-bound heuristic func-

tion. The heuristics made use of pattern databases, which held information about the number of moves needed to solve subsets of the original problem.

This was sufficient for solving random instances of the Rubik's cube, the 15-puzzle and the 24-puzzle.

Sokoban seems like a harder domain to tackle than the 15-puzzle and the Rubik's cube. Domain-specific enhancements were needed in order to solve many puzzles. IDA* with transposition tables wasn't enough to solve more than a few puzzles. Adding tunnel macros, goal macros, goal cuts and relevance cuts helped cut down the search space, but this resulted in no guarantee of optimal solutions.

Sliding-block puzzles and Sokoban share several properties:

- The problem of finding any solution is PSPACE-complete.
- Different search space for each puzzle (unlike the Rubik's cube and the $n^2 - 1$-puzzle).
- On average, the solution sequences are longer than for the other domains.

In Sokoban, the deadlock tables were common for all puzzles. The pattern search in partial configurations used for improving the lower-bound heuristic, however, had to be done for each puzzle as different puzzles in general have different maze layouts.

These results from Sokoban suggest that we need to look into domain-specific enhancements in order to have any hope of solving difficult sliding-block puzzles.

The solving efforts for these problems also suggest that IDA* is the ideal choice. For difficult puzzles, algorithms like BFS and A* have huge memory requirements. Although in one instance, BFS was used to do a complete search of the 15-puzzle.

We are not aware of any attempt at making lower-bound heuristics for sliding-block puzzles. We must be prepared to have difficulties in attempting to construct a good lower-bound heuristic.

Table 3.1 summarizes some of the search space properties for these four problems.

Table 3.1: Average (mean) branching factor, search space size and solution lengths for the various domains.

	Sliding-block puzzles	Sokoban	15-puzzle	24-puzzle	Rubik's cube
Branching factor	5.43	12	3	3.2	18
Search space size	10^{15}	10^{18}	10^{13}	10^{25}	10^{19}
Solution length	246	260	53	100	18

Chapter 4

Solving sliding-block puzzles

For convenience, we repeat some of the significant search space properties for sliding-block puzzles.

- There are numerous goal nodes, but they can be located deeply in the tree.
- The search space is different for each puzzle, it can be huge, and the underlying graph is undirected.
- The branching factor varies, but it can be large.
- Small puzzle layouts can have long solution lengths.

In this chapter we will discuss several algorithms and enhancements and how well they could perform on solving sliding-block puzzles, given the properties we know about the search space, and the previous solving efforts of both sliding-block puzzles and the other kinds of puzzles mentioned. We will then recommend one or more algorithms to implement.

About the importance of algorithmic enhancements and the difficulty of making use of them, Culberson [3] says "When presented in the literature, single-agent search (usually IDA*) consists of a few lines of code. Most textbooks do not discuss search enhancements, other than cycle detection. In reality, non-trivial single-agent search problems require much more extensive programming (and often research) effort. For example, achieving high performance for solving [the 15-puzzle] requires enhancements such as cycle detection, pattern databases, move ordering and enhanced lower- bound calculations"

We have the choice between informed and uninformed algorithms. What we should choose, depends on the availability of domain knowledge. If domain knowledge is available, an informed algorithm is preferred.

4.1 Uninformed algorithms

4.1.1 Random walk

Walk to a random neighbour in the search space graph randomly until we reach a solution node. This might look like a silly algorithm, but it could actually be useful if we don't care about the quality of the solution, the goal density is high (the number of goal nodes compared to the total number of nodes in the search space graph), and we have little to no domain knowledge. Junghanns [7] suggests that in each step, we select a random open node instead of selecting a neighbour of the current node.

4.1.2 Breadth-first search

Examine all nodes in the search space. All nodes in search depth n are examined before any node in search depth $n + 1$. We are guaranteed to find the optimal solution if there is one. The disadvantage is the memory requirement, we need to be able to hold at least two complete search depths in memory at any one time, and having a way to detect duplicate positions will require even more memory.

4.1.3 Depth-first search

Examine the entire subtree before examining sibling nodes. The basic algorithm won't work well on our problem, since the search space graph has cycles. At the very least we need to avoid visiting nodes we visited in order to reach the current node.

4.1.4 Depth-first iterative-deepening (DFID)

Iterative-deepening is a technique that can be applied to depth-first search. The depth-first search is run multiple times, starting with a maximum depth of 1 which is increased by one for each run. When the algorithm terminates, we have found the optimal solution. In the case where DFS and iterative deepening DFS are run on trees, iterative deepening DFS is asymptotically optimal with regard to space, runtime (nodes examined) and solution length compared to BFS and DFS [9]. One could think that it's a waste of resources to run repeated DFS's for nodes of early depth, but the last iteration's runtime will dominate the cost.

However, this nice property no longer holds when searching on a graph that is not a tree. DFS in its purest form doesn't keep track of visited states, so we need to take care of that.

Even when we do keep track of visited states within each iteration, iterative deepening can be a worse choice than DFS for certain search spaces with a low branching factor, or where the effective branching factor decreases as a function of the search depth. In such cases, the last iteration will not dominate the cost of all the previous iterations and the total execution time for iterative deepening can be much higher than for a regular DFS.

The technique of iterative deepening can also be applied to other search algorithms such as bidirectional search and A*.

4.1.5 Bidirectional search

Bidirectional search is a technique using two searches - one from the starting position, and one from the goal position(s). The search terminates when the two search trees intersect.

Assume a constant branching factor b and a distance to goal d. Then, the two searches each have complexity $O(b^{d/2})$. The sum of these two search times are much lower than the time complexity $O(b^d)$ that would result from a standard search.

The disadvantage is that we need a way to determine when the search trees meet each other. Usually this means keeping one of the search frontiers in memory, and checking every newly generated node in the other tree if it is contained in the search frontier.

Other potential problems include goal nodes that are implicitly stated, or if there are multiple goal modes. Sliding-block puzzles can contain an exponential number of goal nodes, which clearly poses a problem for this algorithm.

Also, the effective branching factor for a problem could be higher when going backwards.

4.2 Informed algorithms

4.2.1 A*

A* is a best-first search algorithm. At any time, the node with the lowest $f(x) = g(x) + h(x)$ among the unexplored nodes is expanded. Here, $g(x)$ is the actual cost of moving from the start position to node x, and $h(x)$ is the

estimated distance from x to goal. In order for A* to find optimal solutions, $h(x)$ needs to be *admissible*, that is, it must never overestimate the cost from x to a goal.

The efficiency of A* depends on the heuristic function $h(x)$. Exponential growth will occur, unless

$$h(x) - h^*(x) \leq O(\log h^*(x)),$$

where $h^*(x)$ is the true cost of getting from x to the goal [20]. We don't expect to find a heuristic function satisfying this condition, and none of the previous efforts managed to satisfy this condition for any of the problems we've looked at.

In practice, the main drawback of A* is its memory usage, rather than computation time. This makes A* unpractical for problems with huge search spaces.

4.2.2 IDA* (Iterative-deepening A*)

IDA* is the combination of depth-first iterative-deepening with A*. IDA* was first described by Korf [9].

IDA* works as follows. The search is done over several iterations. During one iteration, a depth-first search is performed. IDA* does not expand nodes having a total heuristic cost $f = g + h$ higher than some cutoff p (also called the *pathlimit*. If no solution was found during the iteration, the cutoff p is increased and another iteration is run.

Given an admissible heuristic h, IDA* will find the optimal solution, if one exists. If the search space graph is a tree, IDA* will expand approximately the same number of nodes as A*. The search spaces we are looking at are not that nice, however. In order for IDA* to avoid expanding the same node twice, we need to keep track of previously visited nodes. In this case a transposition table is often used.

According to Junghanns[7, p.34], if the ratio between the correct distance $h*$ and the estimated distance h is large, IDA* will perform rather poorly, since many iterations will be run without finding a solution. In this case, A* will be a better choice.

4.2.3 SMA* (simplified memory-bounded A*)

Russell and Norvig [20] describes a memory-bounded alternative to A*, called SMA*.

Before running out of memory, SMA* works just like A* - expanding the best open node according to $f(x) = g(x) + h(x)$. When memory is full, some nodes need to be dropped in order to add new ones. Whenever we need memory, the stored node with the highest f-value is dropped. Then, the algorithm backs up to the node with the lowest f-value and expands it. SMA* will regenerate a forgotten subtree only when all other paths have been shown to look worse than the path which was forgotten. However, Russell and Norvig [20] says this is the most complicated search algorithm they have seen so far.

The advantage of SMA* is that it's possible to solve more difficult problems than A*, given enough memory, without significant overhead in terms of extra nodes generated. It performs well on problems with highly connected state spaces and real-valued heuristics, on which IDA* has difficulties.

On very hard problems, SMA* can perform badly. It can be the case that SMA* will switch back and forth between a set of candidate solution paths. In sliding-block puzzles, there is an exponential number of goal states, and hence it is likely that the number of solution paths is huge. Hence, SMA* could potentially jump back and forth between two or more non-overlapping solution sequences of near-equal length, forgetting and regenerating subtrees several times. Repeated expanding of the same node can make some problems intractable for SMA* even though A* could solve them given enough memory.

4.3 Algorithmic enhancements

In this section, we will discuss the different enhancements used for solving the 15-puzzle, Rubik's cube and Sokoban.

4.3.1 Transposition table

A transposition table fulfills two purposes: To avoid cycles in the search graph when using IDA*, and avoid expanding previously visited (and expanded) nodes. Additional information can be attached to a node, like the best lower-bound found so far.

If we continue to insert into the table, we will eventually run out of memory. We have a few alternatives:

- Give up, and output that we didn't find a solution because we ran out of resources.

- Drop entries from the table. It could be the oldest one, the one farthest from a goal state (according to the heuristic evaluation) or some other criteria.
- Utilize second-level storage.

A way to implement transposition tables is to use a hash table, using the current position as the key.

4.3.2 Move ordering

When there are several equal candidates for the next move, choose the most promising move among these.

Junghanns [7] used move ordering for Sokoban. They defined the concept of *inertia* - if a particular stone was pushed in the previous move, moves pushing the same stone again will be given priority among the moves with equal cost.

For sliding-block puzzles, one possible use of move ordering is to always consider master block moves before other moves.

4.3.3 Weighted heuristics

This is a tweak that can be applied to A* and IDA*. Given a non-tight heuristic function, the idea is to scale it so it better approximates the actual cost. The cost function becomes $f(x) = g(x) + w \cdot h(x)$. We will lose admissibility, but it can work well in some cases. Junghanns [7] reported mixed success when applying WIDA* (weighted IDA*) on Sokoban puzzles. They tried different values of w between 1.025 and 1.25. With $w = 1.15$ they managed to solve one more puzzle, but the erratic behaviour of the search made it difficult for them to justify the use of WIDA*.

4.3.4 Pattern databases

The idea of *pattern databases* is to store the exact distance to the goal for a subproblem, namely a relaxed version of the problem we are trying to solve. A pattern database stores the exact solution costs for every instance of the subproblem. This will be used as a lower-bound heuristic in a search algorithm like A* and IDA*.

Pattern databases were used by Culberson [1] and Korf [11] [12]. Culberson [1] used pattern databases to solve the 15-puzzle efficiently. They used a database containing exact solution costs of all positions containing 7 specific

numbered tiles and the empty cell. This lead to a better lower-bound heuristic than the sum of Manhattan distances to the goal for each tile. Korf [12] used disjoint pattern databases to solve the 24-puzzle. The main idea here is that the sum of lower-bounds from disjoint goals will be an admissible lower-bound for the full problem. They used four disjoint databases containing 6 tiles each, and each group was chosen so that the tiles within each group were close to each other in the goal state. The reason was that these tiles were more likely to interact with each other, creating a better lower-bound heuristic. For each group, all possible positions with the 6 tiles were stored along with the exact solution cost.

Korf [11] used pattern databases for Rubik's cube. Three databases were made: one considering only corner cubies, and two for the two groups of 6 edge cubies. In each case, the distance to goal were computed for every possible state. The lower-bound heuristic used was the maximum of cost found from the corner cubie database and the two edge cubie databases. The expected value for the maximum of these three heuristics were 8.878 moves. Given that the median solution length is believed to be 18, this resulted in an efficient search.

The deadlock tables in Sokoban is a special variant of a pattern database. Instead of improving the lower-bound heuristic, the deadlock tables are used to identify deadlocks in a position. Junghanns [7] generated all possible deadlocked positions for a 5×4 grid. If a move leads to a position contains a pattern that is contained in the deadlock tables, this move is not inserted into the move list.

If we were to use pattern databases on sliding-block puzzles, we would need to construct separate databases for each puzzle, since each puzzle contains differently shaped blocks. Hence, the cost of constructing the databases cannot be amortized over different problem instances and should count towards the effort going into solving the puzzle.

Performance test of pattern databases on sliding-block puzzles

We did a small test to check the performance of pattern databases on the puzzle *The Devil's nightcap*.

We removed one non-master block at a time, and searched from the new starting position to find the number of steps to a solved position. With this, we achieve the same effect as looking up the starting position in a pattern database in order to find a lower bound for the number of steps needed to solve the puzzle.

The results can be seen in table 4.3.4. For each block size removed, the

Table 4.1: Using pattern databases on The Devil's nightcap

Block removed	Solution length	Search space size
None	888	497,872
1×2	47	23,937,410
1×1	70	3,746,424
2×1	43	15,463,188
3×1	29	43,431,574

solution length is the maximum taken over each copy of the block removed.

The best pattern (removing one 1×1 block) is still only 0.079 times the real cost, which makes it a bad fit, and is likely to not be very helpful in directing a search using A* and IDA*.

From this test, it appears that pattern databases for sliding-block puzzles are of limited use for puzzles with little space and long solution sequences. Another disadvantage is the size of the pattern database, it can be larger than the search space of the puzzle itself. In such a case, it is more sensible to do a full search in the actual puzzle. (Though, in this case we can reduce the size of the pattern database by $\frac{1}{5}$ by locking the 3×1 block to the lowest row.)

However, we cannot conclude anything about the usefulness of pattern databases on other kinds of puzzles. It might happen that the lower bound will be a better fit in these cases, though it is likely that the work needed to construct a pattern database will be significant.

4.3.5 Macro moves

The idea of macro moves is to reduce the search space by combining several moves into one super-move - a macro.

This concept is useful when it doesn't make much sense to break up a sequence of moves. Two examples come from Sokoban:

- **Tunnel macro**: Whenever a stone is pushed into a tunnel, push it all the way through.
- **Goal macro**: Whenever a stone has a clear path to a goal square, push it all the way to the goal.

A disadvantage is that macros may not always preserve optimality of the search algorithm used. Junghanns [7] reports that the use of goal macros may result in non-optimal solutions for Sokoban.

4.3.6 Endgame databases

For problems where instances share the same search space (like the 15-puzzle and the Rubik's cube), it can be beneficial to construct an endgame database.

An endgame database contains all positions up to a distance d from the goal state, along with the exact distance to the goal. d is usually the largest possible value such that the endgame database fits on some storage device (either memory or disk).

Culberson [1] used endgame databases as one of several improvements for solving the 15-puzzle. Their endgame database contained all positions up to 25 moves away from the goal position.

Endgame databases have been commonly used in game-playing programs for two-player games such as Chess and Checkers.

However, since each sliding-block puzzle has separate search spaces and the goal positions are rarely fully specified, endgame databases appear to be less useful for this domain.

4.3.7 Relevance cuts

Relevance cuts are a way to emulate humans and the way they successfully manage to navigate through large search spaces. For huge problems, humans can make a plan on how to reach the goal, and this plan involves a series of steps. For a given position in a problem, not any move is relevant in order to achieve the goal. The search is restricting the next possible moves to moves that are related to the previous moves (with some exceptions).

Junghanns [7] used relevance cuts as an enhancement in Sokoban. They constructed a table *influencetable[a,b]* which returns the influence a move from position a has on a move from position b. Two moves influence each other if the entry in the table for a, b contains a value not above a user-tunable threshold. Two moves that aren't influencing each other, are *distant*.

A move was cut off if more than k distant moves were made within the last m moves. k was always set to 1 in their experiments.

With this technique they managed to solve two more puzzles. It also resulted in large reductions in the effort required to solve other puzzles.

4.3.8 Pattern search

Pattern search is a technique used only in Sokoban, as far as we know. For each position, a small search is done with some stones removed. The result

of this search is used to improve the lower-bound heuristic for the position. It can be considered a dynamic variant of the pattern database.

4.4 Domain-specific enhancements

In section 3.5, we concluded that we needed domain-specific enhancements in order to solve more difficult sliding-block puzzles. The domains of sliding-block puzzles and Sokoban share some of the properties that make these kinds of puzzles harder to solve than instances of the 15-puzzle and Rubik's cube.

In this subsection we are looking into suggestions for enhancements specific for sliding-block puzzles.

Some of these ideas have previously been mentioned on the public forum on the Bricks website [24]. More precisely, this source mentions enhancements as directed search, distributed computing, dropping of scattered space boards and subboard optimisation.

We refer to section 2.7 for a list of human strategies that we may want to emulate.

4.4.1 Pruning based on properties of positions

One possibility is to prune search trees based on static analysis of a position. If a position is considered unpromising, we don't add the position to the move list.

In order to move a block around, there needs to be open spaces in front if it in the direction of movement. Hence, it's hard to move blocks around if the open spaces are scattered across the board.

We can design an evaluation function that returns a value such that a lower value means that the open spaces in the position are grouped closer together. We can then prune the search tree for positions where this number is larger than a threshold value. One possible function q is given in equation 4.1, where the position contains n spaces with coordinates $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$. This function returns the sum of Manhattan distances of all pairs of spaces.

$$q(\text{position}) = \sum_{i=2}^{n} \sum_{j=1}^{i-1} \left(|x_i - x_j| + |y_i - y_j| \right) . \tag{4.1}$$

Another suggestion for such a function is the minimum number of squares needed in order to connect all spaces in a position.

Care must be taken so that a too low threshold isn't selected. A too low threshold can have two consequences: Either one loses the optimal solution, or one loses the ability to find any solution at all. We are more concerned with the last issue in this project.

One possible algorithm is to begin with a low threshold (for instance, the value returned from the starting position), and if no solution is found, increase the threshold and repeat the search.

Some other metrics could also be considered when determining the goodness of a position:

- General mobility: The number of possible moves in a position.
- Master block mobility: The shapes and sizes of the blocks near the master block.
- The number of fault lines in a position. A fault line is a vertical or horizontal line which doesn't intersect any blocks.
- In addition to measuring how far away the spaces are from each other, the positions of the spaces relative to the master block are interesting as well. It's desirable to have them close to the master block.

Instead of cutting these positions off the search tree completely, the features described above can instead be incorporated into the heuristic evaluation function for A*/IDA* in order to penalize such positions. It would be very hard to make sure such a function is admissible, though.

4.4.2 Don't allow certain moves

One feature that can characterize moves done by a human player in such a puzzle, is that blocks are moved according to a plan. Each block moved will have a designated destination.

Often, in the initial position of a puzzle, the master block is at the opposite side of the grid from the destination square. In addition, the blocks in middle are shuffled in an unfavourable way. The solution often involves repacking of the pieces so that they are easier to move around. When a block is parked, it will rest against other blocks on at least two sides.

One heuristic attempting to capture this way of parking blocks, is to disallow moves that don't leave blocks in a position good for packing.

A block is said to be *resting* if at least two of its non-opposite sides are resting against other blocks, or the frame. A block is *hanging* if it isn't resting.

When a block is in transit from its previous position to its new position, it will sometimes necessarily enter the hanging state. What we wish is to avoid having several blocks at this state. If at least one hanging block exists in a position, we don't want to move other blocks (except possibly the master block).

4.4.3 Don't allow the master block to move away from the goal

This is an idea that might not work in many cases, but can lead to huge reductions in the search space when it does work.

The idea is simple enough: Don't consider moves that increases the distance of the master block to the goal.

Other variants are possible, for instance only allowing up to a certain number of such moves.

4.4.4 Assume that some blocks are never used

This is an idea that is likely to work on some larger puzzles and/or puzzles with multiple solution paths. It is already known to work on the puzzle *Turtle*. One can conjecture that some of the blocks need not be moved in order to solve the puzzle. For instance, it is trivial to see that one column of blocks need not be moved on the puzzle Easy.

However, the difficulty of automatically finding blocks that never need to be moved is unknown and remains to be investigated. The simplest solution would be for a human to select the blocks. One could also imagine an algorithm that starts with the entire board locked, and then iteratively unlocks blocks for movement. Finding the next block to unlock is a potentially difficult problem. Another way of unlocking pieces is to make a graph of the blocks, where there is an edge from block a to block b if b must be moved in order for a to move. Then, do a topological sort on the graph. If we wish to unlock a block c, we also unlock every block corresponding to a node that can be reached from c.

4.4.5 Cleanup after running out of memory

When doing a full search in a huge search space, we are expected to run out of memory at some point.

When this happens, we can throw something out of memory and continue the search. For instance, when the search queue exceeds p elements, we can go through the queue and drop every element except the q best ones, according to an evaluation function.

To keep the search going even longer, we also need to throw out elements from the set containing all positions we have seen so far. When the set is full (according to some limit we have set) and we need to insert a new element, there are two main ways to drop an element: Either drop the least frequently used one, or the least recently used one. Either way, the set structure would need to be augmented to keep track of this information. One could also analyse the usage pattern of this set, and come up with hybrid strategies that work better for this domain.

4.4.6 Subboard (local) solving

Consider to only move blocks inside a subboard surrounding the master block. For instance, don't allow moves where the block to be moved is farther away from the master block than d unit squares. The distance d could for instance be the smallest Manhattan distance from a subsquare of the master block to a subsquare of the block to be moved.

Finding a good value parameter d is hard. If it is too small, the puzzle would be unsolvable. An iterative algorithm could be run, such that d is increased and the search is restarted if no solution was found.

Also, in some puzzles the spaces are far away from the master block, making it harder to restrict the d value. Though that could be decreased as the spaces are moved closer to the master block.

4.4.7 Plan with subgoals

Support intermediate goals. When an intermediate goal is reached, clear the search queue, and start a new search where the new start node is the node that reached the intermediate goal. In addition, not clearing the history of previously visited positions will prevent the search from going backwards.

Typical intermediate goals are moving the master block to certain positions (along the path from the start position to the final goal), move a key block out of the way, or move the spaces to a beneficial position, like in front of the master block or a key block.

Another improvement would be to automatically find such subgoals. This is expected to be a very hard task.

4.4.8 Post-processing non-optimal solutions

Many of the techniques we have mentioned will lead to non-optimal solutions. If we want to shorten the solution sequence, post-processing can be applied. For instance, take a short segment of the solution found, and use a small search to find the shortest path between the start and the end of the segment.

Another alternative is to improve a non-optimal solution by human inspection.

4.5 Parallelism

Modern CPUs typically come with several processing cores on the same chip. This is another capability that can be exploited, and can result in vast speedups. The speedup will enable us to search a bit farther into a search space, given a constant amount of time.

However, there are some disadvantages. Designing parallel algorithms is more difficult than designing serial algorithms. The resulting implementation is likely to be more complex than the corresponding sequential algorithm.

Due to the limited time at disposal in this project, we will not pursue this idea further.

Korf [12] used parallel breadth-first search in order to do a complete search for the 15-puzzle. They used POSIX threads to parallelize the search. All threads shared the same data space, and mutual exclusion was used to temporarily lock data structures.

4.6 Memory hierarchy

Some algorithms like BFS and A*, the memory limit is often reached before we run out of time or patience. Hence, using disk drives to enable us to search much farther is an interesting possibility.

The algorithms need to be designed carefully, however, so that the disk access is sequential whenever possible, as random disk access is several orders of magnitude slower than memory access.

Our previous sliding-block puzzle solving program used disk storage with sequential disk access to allow us to search farther where we normally would run out of memory.

We will not pursue this idea further in this project, apart from using our old sliding-block solver when we find it useful.

When using disk storage, care must be taken to avoid corrupt data. Korf [12] used disk storage in their complete breadth-first search of the 15-puzzle, in addition to using a parallel algorithm. They had to use several disks in RAID in order to guarantee correctness of the data that was written to disk, as the large amounts of data written (10^{15} bits) resulted in frequent disk errors when run on a disk system without redundancy mechanisms.

4.7 What we will implement

We have decided to implement BFS, A* and IDA*.

BFS was chosen because it is easy to implement, and it is expected that we can examine more positions in a given time frame than the other algorithms. Also, we have some puzzles in our test suite where we expect A* and IDA* to perform badly.

IDA* was chosen because it has performed well on all the similar domains mentioned in section 3. However, the performance of IDA* is highly dependent on the quality of the heuristic function. If the $h(x)$ estimate is too low, it can lead to an excessive number of iterations being performed. We will implement a simple transposition table with no other purpose than marking previously visited positions in an iteration.

A* was chosen as an alternative to IDA*, since it is very easy to implement both IDA* and A* once we have written the necessary subroutines. Also, A* is a nice reserve in case IDA* performs poorly. Another reason is that we wish to test non-admissible heuristics.

We have decided to include the following enhancements and ideas:

- **Transposition table**: Added for IDA*, because IDA* does not perform well on graphs with cycles unless we mark visited positions.
- **Weighted heuristics**: This is easy to implement, and we believe it will result in an increased performance for the A* algorithm, since we can achieve a better fit to the actual cost function.
- **Pruning based on the closeness of spaces**: We believe this is a key property for describing a good position in our domain, and this can lead to large reductions in the search space size. We will also use a space closeness function as a penalty add-on to the heuristic function.
- **Position of spaces relative to the master block**: This can be useful for giving the heuristic function increased granularity in measuring

the distance to a goal position.

- **Pruning: Don't allow to move other blocks in the presence of hanging blocks**: We think this is an important concept which emulates they way humans play.
- **Pruning: Don't allow to move the master block away from the goal**: This enhancement will probably only work on the easier levels, but we include it since it is easy to implement.
- **Assume that some blocks are never used**: It is known that this idea will result in enabling Turtle to be solvable. In addition, it is easy to implement.
- **Cleanup after running out of memory**: This idea has been used with success by Pflug [15], so we include it as well.
- **Subgoals**: This will break up a puzzle into smaller parts, and we believe this enhancement can reduce the complexity of a puzzle.

The following enhancements and ideas will not be included:

- **Move ordering**: We cannot see how this can be beneficial for our domain.
- **Pattern databases**: Preliminary tests shows that constructing pattern databases will use more resources than solving the puzzle itself.
- **Macro moves**: We cannot find a way to make use of this idea in our domain.
- **Endgame databases**: These cannot be used in our domain, because the goal positions are rarely fully specified, resulting in an exponential number of goal positions in the search space.
- **Relevance cuts**: This enhancement assumes that there exist moves in a position that don't influence each other. This does not hold for our domain.
- **Pattern search**: This is an enhancement similar to pattern databases, and will not be included for the same reason as for pattern databases.
- **General mobility**: We think this enhancement will be made redundant by other enhancements that we are including (pruning and penalty based on the closeness of spaces).
- **Master block mobility**: This could have been a good enhancement, but we have no clear idea of how to implement this idea.
- **Number of fault lines**: We believe this concept is captured by another concept (general mobility), which we also chose to not include.

- **Sub-board (local) solving**: The idea is interesting, but we cannot find a good way to implement it.
- **Post-processing non-optimal solutions**: In this project, we focus on finding any solution to a puzzle rather than finding the optimal solution.

Chapter 5

The implementation

This chapter will describe the implementation that was created during this project. Our previous effort, an efficient BFS solver using disk swapping, is described in section 2.9.2.

Its code base is rather messy, making it hard to expand the program with new features. Therefore, a new program was made from scratch, attempting to make it easier to expand with new algorithms and improvements than the previous version. Some ideas were reused, like the way a position is represented in memory using Huffman coding.

5.1 Description of the program

The program is operated from the command line prompt. The program accepts several parameters. The only mandatory parameter is a file name containing the puzzle to be solved. This file can contain intermediate goal positions (partial goals). The following list contains the optional parameters.

- Choice of algorithm (BFS, A* or IDA*).
- Set initial *pathlimit* value for IDA*.
- When reaching a subgoal, clear the history of previously visited positions.
- Force efficient duplicate check for BFS (correct when moves are reversible), even when it is wrong to use it.
- Turn on or off all the different prunings, as well as adjusting their parameters.

- Select which heuristic function to use, as well as adjusting their parameters.
- Don't terminate when solution is found.

Figure 5.1 shows a screenshot of our program, showing the usage screen with the available parameters. Figure 5.2 shows the output after solving the Triathlon puzzle with out best settings.

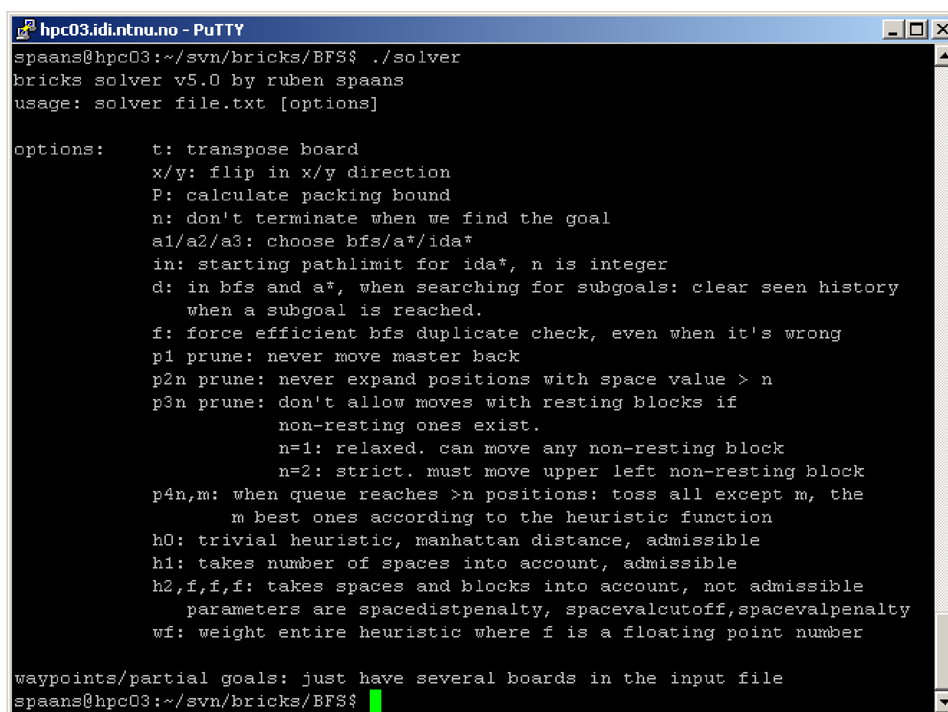


Figure 5.1: Screenshot of our solver, options screen

The same program is used to find the packing bound, and hence it contains a parameter for invoking the bound calculation, and some additional parameters for transposing and flipping the board in order to help speed up the bound calculation.

Implementation was done with the purpose of being easy to maintain and expand rather than being fast. In fact, the new program ended up being a factor of 6 slower than our old program. Subroutines was made to be used for all the implemented algorithms, and some do slightly more work since they are general for all algorithms. Also, structures are passed via parameters, instead of storing them globally as in the old program. In some cases, structures are copied instead of passing a reference. The latter is probably what accounts for most of the difference in execution time compared to the old version.

```
hpc03.idi.ntnu.no - PuTTY
0 (1): (0 1) (1 0) (1 1)
1 (2):
2 (4): (0 1)
3 (8):
4 (3): (0 1) (0 2)
5 (1): (-1 1) (0 1)
6 (7): (1 0)
code takes 63 bits, 8 bytes
 0: 4 0010 (4)
 1: 3 000 (0)
 2: 3 101 (5)
 3: 2 11 (3)
 4: 3 100 (1)
 5: 4 0011 (12)
 6: 2 01 (2)
heuristic evaluation function #2
heuristic weight: 8.000000
a* invoked.
checked 2000 pos, 583 in q, g,h=67,336.0, bd=8, maxg=109, 00:00
checked 4000 pos, 1147 in q, g,h=37,376.0, bd=8, maxg=126, 00:00
checked 6000 pos, 1652 in q, g,h=21,384.0, bd=8, maxg=128, 00:00
checked 8000 pos, 2342 in q, g,h=71,304.0, bd=8, maxg=128, 00:00
checked 10000 pos, 2882 in q, g,h=89,264.0, bd=8, maxg=128, 00:00
checked 12000 pos, 3383 in q, g,h=86,328.0, bd=8, maxg=128, 00:00
checked 14000 pos, 3779 in q, g,h=108,320.0, bd=7, maxg=132, 00:00
checked 16000 pos, 4345 in q, g,h=161,256.0, bd=7, maxg=165, 00:00
checked 18000 pos, 5196 in q, g,h=157,232.0, bd=7, maxg=181, 00:00
checked 20000 pos, 5763 in q, g,h=87,288.0, bd=7, maxg=184, 00:00
checked 22000 pos, 6286 in q, g,h=88,272.0, bd=7, maxg=187, 00:00
checked 24000 pos, 6684 in q, g,h=152,256.0, bd=7, maxg=188, 00:00
checked 26000 pos, 7280 in q, g,h=113,328.0, bd=6, maxg=188, 00:01
checked 28000 pos, 7981 in q, g,h=190,232.0, bd=6, maxg=193, 00:01
checked 30000 pos, 8605 in q, g,h=157,224.0, bd=6, maxg=198, 00:01
checked 32000 pos, 9418 in q, g,h=77,368.0, bd=6, maxg=199, 00:01
checked 34000 pos, 9970 in q, g,h=101,328.0, bd=6, maxg=199, 00:01
checked 36000 pos, 10414 in q, g,h=143,304.0, bd=6, maxg=203, 00:01
checked 38000 pos, 11092 in q, g,h=88,296.0, bd=6, maxg=203, 00:01
checked 40000 pos, 11617 in q, g,h=73,376.0, bd=6, maxg=203, 00:01
checked 42000 pos, 12794 in q, g,h=210,208.0, bd=4, maxg=227, 00:01
checked 44000 pos, 13492 in q, g,h=255,120.0, bd=1, maxg=271, 00:01
checked 46000 pos, 13978 in q, g,h=279,56.0, bd=1, maxg=285, 00:01
checked 48000 pos, 14518 in q, g,h=196,216.0, bd=1, maxg=300, 00:01
checked 50000 pos, 14925 in q, g,h=248,104.0, bd=1, maxg=307, 00:01
checked 52000 pos, 15559 in q, g,h=221,192.0, bd=1, maxg=314, 00:01
reached subgoal 1 in 264 steps, processed 52376 positions in 0:00:01
spaans@hpc03:~/svn/bricks/BFS$
```

Figure 5.2: Screenshot of our solver, solving the Triathlon puzzle

The program does not support swapping to disk when the memory is used up. For excessively large BFS runs we still have the option of using the old program.

5.2 Implementation of heuristics and improvements

This section describes how the heuristics are implemented. Each heuristic can be activated manually by the user.

5.2.1 The A*/IDA* heuristic function

The program includes three slightly different heuristic functions. These are explained in the following subsections.

The Manhattan distance heuristic, $h_0(x)$

The first heuristic function is rather trivial: The Manhattan distance from the position of the master block to its goal position. This heuristic is clearly admissible, because it's not possible to move a block a distance of d in less than d steps.

Slightly improved heuristic, $h_1(x)$

This heuristic attempts to capture the fact that there is a limited amount of free space in a puzzle, and extra steps are needed in order to bring the free space in front of the master block each time it advances.

The following formula was implemented:

$$h(x) = x_m + y_m + 2\frac{x_m x_r}{s} + 2\frac{y_m y_r}{s},$$

where x_m, y_m is the distance of the master block to goal in the x-axis and y-axis, respectively, x_r, y_r is the number of free spaces required for the master to move horizontally and vertically, respectively, and s is the number of free cells in the puzzle.

The intention was to design this to be an admissible heuristic. However, we didn't attempt to prove it and it turns out the heuristic is not admissible after all. There exist positions in the puzzle Isolation where the heuristic function overestimates the distance to the goal. We decided not to change it, since it was discovered after the majority of the tests were done. In addition, almost

all the runs that used this heuristic also used added penalties which clearly make the resulting heuristic function non-admissible.

Non-admissible heuristic, $h_2(x)$

With basis in the above heuristic, two penalties can be added.

- A penalty based on the average position of the spaces, and its distance from the master block.
- A penalty based on how scattered the spaces are.

The formula for the first penalty p_1 is:

$$\begin{aligned} v_x &= \max(0, |x_w - (x_p + x_r)| - x_r) \\ v_y &= \max(0, |y_w - (y_p + y_r)| - y_r) \\ p_1(x) &= \alpha_1(v_x + v_y) \end{aligned}$$

where x_w, y_w is the average position of the spaces, x_p, y_p is the position of the upper left portion of the master block, x_r, y_r is the centroid of the master block, α_1 is a scalar giving the weight of this penalty and x is the position we want to evaluate. v_x and v_y represent the distance of the centroid of the spaces to the centroid of the master block, after subtracting the "radius" of the master block. We use the difference between the upper left corner of the master block and the centroid of the master block as the radius.

The formula for the second penalty p_2 is:

$$p_2(x) = \alpha_2 \max(0, s - t)$$

where s is the position's *space value* (equation 5.1, t is a lower threshold for what space values we want to give a penalty, α_2 is a scalar giving the weight of this penalty and x is the position we want to evaluate.

Weighted heuristic

This modification was originally explained in section 4.3.3. To repeat, the solution cost used in A* and IDA* when using a weighted heuristic function is $f(x) = g(x) + wh(x)$ for some constant $w > 1$, where $g(x)$ is the actual cost from the start to the current position, and $h(x)$ is the estimated cost to goal. w can be set by the user.

5.2.2 Pruning the search space

Several enhancements are made in an attempt to reduce the search space, while hopefully not cutting off positions vital for finding a solution.

Never move master block away from goal

When this pruning is activated, the search will not expand nodes corresponding to positions where the master block attempts to move away from its goal position. The master block is said to move away if the move increases the Manhattan distance of the block to the goal.

This enhancement removes *edges* from the search space graph rather than *nodes*. However, some nodes might also be removed as well, due to them becoming unreachable. This can be seen in some of the easier puzzles where the master block never needs to diverge in order to reach goal.

However, with this enhancement, master block moves become irreversible. Hence, we risk accepting duplicates in our search queue if we only check for duplicates against the two previous search depths (rather than all previous search depths) in the BFS.

Remove scattered space positions

When a move is performed, a *space value* is calculated from the resulting position. Let $x_1, x_2, \ldots, x_n$ and $y_1, y_2, \ldots, y_n$ be the x and y coordinates of all the spaces in the position. Equation 5.1 shows how this value is calculated. The value is the sum of the Manhattan distances of all pairs of spaces.

$$\sum_{i=2}^{n} \sum_{j=1}^{i-1} \left(|x_i - x_j| + |y_i - y_j| \right). \tag{5.1}$$

When this pruning is activated, only positions with a space value $v \leq t$ will be expanded by the search, where t is a user-specified threshold.

Pruning: Drop most of the queue when it reaches a certain size

When the queue size at any time reaches a given size t in A* or IDA*, or the queue size at the end of a search iteration in BFS exceeds t, the following action is performed:

The q best positions (as measured by the heuristic evaluation function) is kept in the queue, while the rest is dropped. In the case where several positions have the same heuristic value and this would cause more than q positions to be kept, the positions occurring earlier in the queue will be preferred.

Pruning: Movement restriction in the presence of "hanging" blocks

A block is said to be *hanging* if it isn't resting against other pieces in two non-opposing sides. If a block isn't hanging, it's said to be *resting*. When this heuristic is activated and at least one hanging block exists in a position, no moves will be allowed apart from moves involving the hanging block or the master block. There are two modes for this heuristic, *strict* and *relaxed*.

- In relaxed mode, any hanging block can be moved if several exist in a position.
- In strict mode, only the first hanging block found during the row-major column-minor scan of the position. This strict condition only matters if there are at least two hanging blocks in a position.

This pruning cannot be combined with the duplicate check in the BFS search that only checks for duplicate positions in the current and the two previous search depths. The reason is that some moves are disallowed in just one direction. Bidirectionality is a necessary condition for this type of duplicate check. Therefore, when this heuristic is turned on in a BFS search, a less efficient type of duplicate check (against all previous search depths) will be performed instead.

This pruning has no negative effects on the duplicate checking in A* and IDA*.

Chapter 6

Results and analysis

In this chapter we will present our results of running the program on the puzzles in the test suite. We will report whether we solved them, the amount of search work needed, which algorithms/improvements we used to solve them and the optimality of the solutions found.

We are especially interested to see the performance difference between an efficient standard breadth-first search against the best settings we could find, on a per-puzzle basis.

We will give an in-depth analysis of the efficiency and usefulness of the heuristics and enhancements we used.

In the "goals" section (section 1.1), we mentioned that we wished to solve a subset of puzzles which were found to be solvable after the literature study which is documented in chapter 2. We will discuss to which degree these goals were reached.

Table 6.1 shows a quick summary of all the puzzles and whether they were solved by the three algorithms we used.

6.1 How the tests were run

With such a huge parameter space that the program allowed and the limited time available for testing, we could not do systematic runs for all puzzles, while covering the parameter space in any sensible way. We performed the tests using the following steps:

(i) Run standard BFS on all puzzles.

(ii) Run standard A* on all puzzles.

Table 6.1: Summary of solved puzzles by algorithm

Puzzle	BFS	A*	IDA*	Puzzle	BFS	A*	IDA*
Easy	✓	✓	✓	Triathlon		✓	✓
Forget-me-not	✓	✓	✓	Magnolia	✓		
Ithaca				Turtle	✓		
American pie		✓	✓	Apple			
Still easy		✓	✓	Schnappi			
Rose		✓	✓	Salambo			
Little sunshine				Sunshine			
Chair				Paragon 1FG			
Isolation	✓	✓	✓	Warmup	✓	✓	✓
Hyperion	✓	✓	✓	Get ready	✓	✓	✓
San	✓			Climb game 15D	✓	✓	
Corona				Climb pro 24			
Thunder				The Devil's nightcap	✓	✓	✓

(iii) Run standard IDA* on all puzzles.

(iv) For each puzzle, consider them separately and attempt to find settings that will either solve the puzzle with the least possible search work, or failing that, make as much progress as possible into solving the puzzle.

Possibilities for the last option include tweaking the algorithm parameters, conjecturing that some blocks in a puzzle won't be used and lock them, and making a plan with subgoals for solving the puzzle.

The tests were run on two computers in NTNU's high performance laboratory, Tesla (12 GB RAM) and HPC03 (8 GB RAM) running Ubuntu. A few tests were also run on one of our personal computers, an AMD64 X2 3800+, 2.0 GHz with 2 GB RAM running Windows XP.

Usually, we ran out of memory after a while, so individual tests weren't running for very long, typically between half an hour and two hours.

6.2 The puzzles we solved

In this section we list the puzzles we managed to solve, and how we solved them, as well as data like solution lengths, search effort spent and settings used.

Table 6.2 shows the puzzles we solved, the solution length, search work needed and which algorithm we used to achieve the least amount of work.

Table 6.2: The puzzles we solved, number of nodes expanded.

Puzzle	BFS	A*	IDA*
Easy	361	**321**	997
Forget-me-not	24,030	**23,688**	1,180,833
American pie	-	**3,246,310**	3,357,760
Still easy	-	**8,468**	572,964
Rose	-	**1,731,647**	6,744,600
Isolation	2,635,906	**2,605,176**	344,389,161
Hyperion	631,845	**599,941**	22,552,539
San	**609,603,451**	-	-
Triathlon	-	**52,376**	52,397
Magnolia	**487,497,593**	-	-
Turtle	**1,112,032,359**	-	-
Warmup	51,312,128	**53,396**	271,341
Get ready	239,208,397	**63,329**	93,585
Climb game 15D	748,286,813	**29,735,389**	-
The Devil's nightcap	142,667	**138,678**	26,050,814

The work needed is defined by the number of nodes examined during the search.

Table 6.3 shows the puzzles we solved, this time we have listed the shortest solution we managed to find, along with the search work needed in order to find this solution, and what algorithm we used in order to achieve this result.

6.2.1 Analysis of the solved puzzles

As expected, we solved the easy puzzles with every algorithm we had at our disposal. The easy puzzles include Easy, Forget-me-not, Isolation, Hyperion and The Devil's nightcap. They were all solvable with examining less than 3 million positions, and in no more than one minute. The difference in number of nodes examined was quite similar for BFS and A*, with A* needing from 1.2% to 11% less work. IDA* needed a factor or 38 to 188 times more work, except on Easy, where IDA* only needed 3 times more work. The very small search space of Easy limits the extra work that can possibly be done. The entire search space is feasible to enumerate for each of these puzzles.

Three more puzzles were solved by BFS and A*: Warmup, Get ready and Climb game 15D. Warmup has an estimated search space size of $6.2 \cdot 10^{11}$, and a solution happens to exist close to the start position in the search space graph. This puzzle was not one we expected to solve with standard BFS. In order to solve Get ready with BFS, we needed to lock 11 of the blocks,

Table 6.3: The puzzles we solved, solution length in steps

Puzzle	BFS	A*	IDA*
Easy	**27**	**27**	**27**
Forget-me-not	**116**	**116**	**116**
American pie	-	**186**	**186**
Still easy	-	**79**	**79**
Rose	-	**85**	**85**
Isolation	**260**	**260**	**260**
Hyperion	**159**	**159**	**159**
San	**227**	-	-
Triathlon	-	**264**	**264**
Magnolia	**401**	-	-
Turtle	**817**	-	-
Warmup	**144**	187	235
Get ready	**177**	228	228
Climb game 15D	**260**	1137	-
The Devil's nightcap	**888**	**888**	**888**

mainly the leftmost and rightmost columns of blocks. Climb game 15D was solvable with standard BFS, but with a search effort of 748 million positions.

With A*, we managed to solve Warmup and Get ready with significantly less search effort than with BFS. BFS needed 960 times more work on Warmup and 2863 times more work on Get ready. These numbers demonstrate the strength of A* combined with a heuristic, in combination with puzzles where it isn't too hard to move the master block around. We used a non-admissible heuristic which gives a huge penalty to the distance of the master block from the goal, so A* will almost always favour positions where the master block is closer to the goal.

With A* we also managed to solve Climb game 15D with less search effort than BFS. BFS needed to examine 25 times more nodes. The reason is the same as for Warmup and Get ready, we used a heuristic which gave a huge penalty for the distance of the master block distance to its destination. However, the blocks in this puzzle are slightly harder to rearrange and move around. While the two previous puzzles should be pretty easy for a human to solve, Climb game 15D is considered to be very hard, because of the irregular shapes of many of the blocks, and the small amount of space.

With BFS we managed to solve three puzzles which we couldn't have to solve using A*.

We had to lock 8 blocks on Turtle and let BFS run for 10 hours. We found a solution after examining $1.1 \cdot 10^9$ positions, making this the hardest puzzle for

us to solve in our test suite. The puzzle contains large blocks and irregularly shaped blocks, which makes it hard to pack the blocks. Our A* heuristic only takes the master block's position and the position of the spaces into consideration. In order to reach a position with the blocks packed in a different way and enabling further progress towards solving the puzzle, it might be needed to go through some intermediate, more messy positions with slightly scattered spaces. Our algorithm won't consider these positions until it has considered all positions with the spaces scattered with similar , but shorter distances, and with the same distance of the master block to the goal. In addition, the memory requirements for our A* algorithm is much higher, since it needs to keep the entire set of previously examined positions in memory. A* managed to move the master block 2 cells away from the goal.

The reason why we managed to solve San and Magnolia with BFS and not with A* is essentially the same as above. BFS needed a lot of search effort in order to find a solution, too much for our A* implementation to handle. Also, the blocks in these puzzles are awkward to pack and move around, which is problematic for our heuristic. In order to solve magnolia with BFS, we had to cut off all positions with a space value higher than 19. 18 didn't work (we cut off all paths leading to a solution), neither did 20 (the search space became too large, we ran out of memory before finding a solution).

We managed to solve 4 puzzles with A* and IDA* that we couldn't solve with BFS. We can't solve these with BFS, because of the branching factor and exponential growth of the search tree.

Still easy is a puzzle in a medium sized grid, containing mostly 1×1 blocks. The puzzle is trivial to solve for a human. The amount of 1×1 blocks make it easy to move the master block around, so the A* algorithm is able to solve this puzzle with very little work: 8468 positions examined.

Rose is a slightly more constrained version of Still easy. Most of the 1×1 blocks are replaced with 1×2, 2×1 and 2×2. This is still considered an easy puzzle for humans. A combination of easiness of moving the master block and a short solution sequence ensure that we can solve it with A*, though we need to examine 1.7 million nodes.

American pie has more space than the two aforementioned puzzles, but has slightly more awkward blocks - sizes like 4×1 and 2×3, in combination with a master block shaped like a Z-tetromino. Because of the amount of space, it's not difficult to rearrange the blocks. This enables us to solve the puzzle with A* using the heuristic with penalty for scattered spaces.

Triathlon takes place in a tall grid with little space. There are enough 1×1 blocks to make it easy enough to rearrange the blocks. In fact, with A* we can solve the puzzle with as little as 52376 positions examined. With BFS

Table 6.4: The puzzles we didn't solve

Puzzle	Search depth (BFS)	Closest to goal (A*)
Ithaca	16	12
Little sunshine	9	10
Chair	41	3
Corona	98	-
Thunder	27	11
Apple	73	4
Schnappi	397	18
Salambo	12	23
Sunshine	29	9
Paragon 1FG	144	2
Climb pro 24	151	2

we ran out of memory after examining 412 million positions. Again, the ease of maneuvering the master block is the key to the success of A* on this puzzle, it fits our heuristic function well.

To sum it up, BFS succeeds on puzzles with a small search space, or puzzles where a solution occur early in the search tree (Warmup is a good example of the latter). The small memory overhead, plus the fact that we only need to keep 3 search depths in memory at any one time, enables us to search much deeper, making it possible to solve puzzles like San and Turtle.

Puzzles containing a small amount of space (2-6) and where it is easy to move the master block around are handled well by A* and our heuristic. It is also beneficial for A* that the master block moves steadily towards the goal, and that there are no points where major rearranging and repacking is needed while the master block is static or even moves back.

6.3 The puzzles we didn't solve

Table 6.4 shows the puzzles we didn't solve, along with the search depth we reached with BFS and closest distance to the goal A* managed to get (Manhattan distance). We did not run A* and IDA* on Corona, because of its unique nature - interior walls and that the master must initially move away from the goal.

As we can see from the table, the master block got close to its goal in several puzzles. It was 2 cells away on Paragon 1FG and Climb pro 24, 3 cells away on Chair and 4 cells away on Apple. However, it is possible to

reach a position where the master block is close to the goal (measured by the Manhattan distance), but requires a long solution sequence to reach the goal.

It might very well be possible that our current program is capable of solving more puzzles - the most likely candidates in our eyes are Chair, Apple, Paragon 1FG and Climb pro 24. Of these, Paragon 1FG has the smallest upper bound on the search space, $3.7 \cdot 10^{12}$. Small changes in the parameters can have a large impact on the outcome. For instance, Magnolia was solved using a space value of $s = 19$, while we couldn't solve it with $s = 18$ or $s = 20$. We cannot rule out that similar small changes in search parameters for the mentioned puzzles could have resulted in us solving more of them.

6.3.1 Analysis of the unsolved puzzles

The main reason why we didn't solve these particular puzzles is the size of the search space in combination with irregularly shaped blocks, which makes it hard to pack the blocks and keep the spaces together. In all cases, we ran out of memory both when using BFS and A*. (IDA* has slightly lower memory requirements than A*, but is much slower.) The puzzle we solved with the largest search space upper bound was Get ready ($3.4 \cdot 10^{17}$), and the unsolved puzzle with the smallest upper bound was Paragon 1FG ($3.7 \cdot 10^{12}$). In a problem domain with exponential growth for each search depth, solution length matters a lot. The shortest known solution for Paragon 1FG is 333 steps, while Get ready can be solved in no more than 177 steps. In addition Paragon 1FG has the higher estimated branching factor of these two puzzles (5.45 versus 3.69).

The progress we made on some of the puzzles were disappointing. We didn't manage to move the master block at all on Ithaca or Thunder when running A*. Both these puzzles feature awkwardly shaped master blocks, and an initial position containing packed blocks, with the master block as the innermost block (or close to being so), requiring tens of carefully planned steps before the master block can be moved at all.

On Sunshine, we managed to move the master block one step down, but not further. The element which makes this puzzle very difficult to solve even for humans is the low number of spaces (only 5, just enough to move the master block one step), two 5×5 blocks and a master block whose bounding box is also 5×5. Each time the master block moves, the blocks behind it must be repacked carefully in order to create new space in front of the master block so it can move again. This puzzle has one of the longest solution sequences in our test suite, the shortest known solution is 713 steps.

On Little sunshine, which is an easier version of Sunshine (for humans, at

least), our program was again not able to move the master block. The starting position is especially unforgiving, the spaces are spread across the puzzle. Our program attempts to quickly gather the spaces in order to keep them close, but it has no concept of good packing. In our best manual attempt we manage to move the master block at the 61th step. Also, this puzzle has the largest estimated search space size and branching factor of our test suite.

We knew that our program couldn't handle Corona in its current form, so we made a plan for this puzzle with 6 subgoals. The subgoals roughly followed the correct path for the master block from start to goal. With BFS we managed to reach the second subgoal, and with A* we managed to reach only the first. Our heuristics and enhancements are ineffective on this puzzle. All non-master blocks are U-shaped and they are impossible to pack tightly, rendering our improvements based on the closeness of spaces useless.

Salambo and Schnappi are, along with Sunshine and Little sunshine, the hardest puzzles in our test suite. Schnappi is in fact almost identical to Turtle (apart from being flipped in the Y-axis); the main difference is that the chair-formed block is reversed, and in Schnappi the master block is trapped inside it. Hence, we cannot do what we did on Turtle and lock the chair-formed block and every block above it. Schnappi is the most difficult puzzle so far to appear in the Bricks series, if we use number of people who have solved as the metric. It has been solved only by four people. The reason it is difficult for our program is because of the size of the puzzle (leading to a large search space), the solution length (best known is 1069) and blocks that are difficult to pack.

Salambo is a quite unique puzzle in our test suite. All blocks are highly irregularly shaped, and there is a lot of space in the puzzle which results in a large amount of possible moves, and hence a high branching factor. The solution involves reordering many of the blocks in the corridor, as the puzzle starts out with the blocks appearing in the wrong order for the final packing. The blocks in the main chamber must be packed in order to swap some of the blocks in the corridor, and then everything must be repacked in order to give room for the master block. For a program to be capable of solving this puzzle, it should be able to find a way to pack the blocks in the main chamber, and find a way to achieve this packing. This includes detecting that the blocks in the corridor need to be reordered, and performing the reordering as well. This puzzle could benefit from other areas of artificial intelligence, such as planning, which could be combined with search. Not surprisingly, our program made little progress on Salambo - it managed to move the master block two steps.

Table 6.5: Number of nodes expanded, by algorithm

Puzzle	BFS	A*	IDA*
Easy	361	**321**	997
Forget-me-not	24,030	**23,688**	1,180,833
Isolation	2,635,906	**2,605,176**	344,389,161
Hyperion	631,845	**599,941**	22,552,539
Warmup	51,312,128	**53,396**	271,341
Get ready	181,324,037	**63,329**	93,585
Climb game 15D	748,286,813	**29,735,389**	-
The Devil's nightcap	142,667	**138,678**	26,050,814

6.4 The efficiency of the algorithms, heuristics and improvements

In this section we will first look the different algorithms and what kind of puzzles they did well at. We will also look at each improvement and see how well they performed.

6.4.1 Comparison of algorithms

Table 6.5 shows the number of number of nodes expanded for each algorithm for the puzzles that we managed to solve with both BFS and A*.

A* is consistent at solving puzzles with less work than BFS and IDA*, though A* in several cases just barely beats BFS. Also, A* beats the other algorithms in solving the most puzzles.

Because of the lower overhead for running BFS and the savings achieved by the efficient duplicate check, our implementation of BFS is able to search deeper than A*, thus being able to find some solutions that A* can't find. The solution for puzzles like San and Turtle is currently out of range for our A* implementation.

As pointed out in section 6.3.1, A* performs well on puzzles where it is easy to move the master block forward, and where there is very little risk of reaching a jammed position when doing so. The number of spaces in the puzzle does not matter too much as long as the first property holds.

IDA* is dominated by A*. This is even though IDA* can solve every puzzle that A* can solve (with one exception). IDA* just needs more resources to achieve the same result. We make a special note of the resource usage on puzzles such at Isolation and The Devil's nightcap. The pathlimit is raised a lot of times for these puzzles, causing IDA* to run a lot of iterations. Due

to the low effective branching factor, the last iteration does not dominate the previous ones in execution time, which results in a lot of extra work. IDA* was designed to exploit that search trees grow exponentially with a high branching factor, but that does not happen in all the puzzles in our test suite.

One possible way to improve IDA* is to improve the pathlimit exponentially, instead of increasing it with the lowest total cost that exceeds the pathlimit. For a chosen $p > 1$, the ith iteration could have a pathlimit of $l \cdot p^i$ where l is the initial pathlimit. The p should be chosen sufficiently large in order to avoid running too many iterations. A scheme where the pathlimit grows dynamically could work as well, as long as it grows exponentially.

6.4.2 The heuristic function

We achieved poor results when we used the admissible heuristic, which only uses the Manhattan distance of the master block to the goal. Our improved admissible heuristic (which turned out to return non-admissible values for Isolation) didn't work much better, they were a poor fit to the actual cost. Even if we weight it in an attempt to match the actual cost, the fact that it doesn't take the spaces into account still makes it poor.

The h_2 heuristic with penalties worked much better than the admissible heuristic. Adding a penalty for the distance of the spaces to the master block enabled the search to distinguish between positions where the master block was in the same position. Without this penalty, we could not solve Climb game 15D with A*. However, the solutions found using this heuristic can be far away from the optimal solution length. For example, using the h_2 heuristic we found a solution 75% longer than the optimal solution.

There are still concepts that our heuristic function doesn't consider. Our heuristic cannot detect if the blocks immediately in front of the master block are easy or hard to move. In some cases the master block can't progress, because it is stuck in front of large blocks or awkwardly shaped blocks. We believe that a heuristic function able to detect these situations can perform much better.

6.4.3 Pruning: Space value

The *space value* pruning reduces the search space by eliminating all positions where the sum of Manhattan distances of all pairs of spaces exceed a given threshold. This can affect the solvability of a puzzle if the threshold is chosen too low.

Table 6.6: The space value effect on Forget-me-not

Forget-me-not			
Solution length	Space value cutoff	Work needed	Search space size
-	2	184	184
118	3	14,756	15,673
118	4	19,599	20,831
116	5	22,579	24,129
116	∞	24,030	25,955

Table 6.7: The space value effect on Isolation

Isolation			
Solution length	Space value cutoff	Work needed	Search space size
-	26	976,881	976,881
351	27	1,462,558	2,097,238
346	28	1,556,674	2,267,778
324	29	2,334,005	3,951,586
316	30	2,456,681	4,170,850
315	31	2,584,478	4,401,008
266	32	2,447,171	4,686,394
262	33	2,496,235	4,833,786
260	34	2,548,059	4,956,468
260	∞	2,635,906	5,129,684

Since we have several puzzles in our test suite where the entire search space can be examined, we can take a look on how this pruning affects the search space.

In tables 6.6, 6.7 and 6.8 we have listed the search space size, search effort needed to solve and solution length for various space value cutoffs for the puzzles Forget-me-not, Isolation and The Devil's nightcap. As we can see, we are not guaranteed to find optimal solutions any longer. Lowering the cutoff sufficiently results in a longer solution length or no solution at all if the cutoff is too low.

Unfortunately, it was not feasible to investigate larger puzzles because of the time required to do these runs. Given more time to run our program, we could have investigated Hyperion, Climb game 15D and San, among others. All puzzles listed in the tables take place in small grids and might not give a correct picture of the savings we can achieve on larger puzzles.

Table 6.9 shows the ratio of the search work (positions examined) in the

Table 6.8: The space value effect on The Devil's nightcap

The Devil's nightcap			
Solution length	Space value cutoff	Work needed	Search space size
-	4	9234	9234
894	5	132,616	447,186
890	6	138,742	482,216
890	7	141,729	493,635
888	8	141,531	497,872
888	∞	142,667	497,872

Table 6.9: Search space and search work savings

Puzzle	Ratio search work	Ratio search space
Forget-me-not	0.614	0.604
Isolation	0.554	0.408
The Devil's nightcap	0.929	0.898

reduced search space compared to the search work in the full search space, as well as the ratio of the reduced search space compared to the full search space. To determine the reduced search space, we used the lowest space value cutoff such that the puzzle was still solvable.

For these three puzzles, the best reduction we achieved was a near half reduction in the search effort for Isolation. For The Devil's nightcap, the search effort was only reduced by around 7%. However, these three puzzles are small, contain little space and the solution sequences range from medium to extremely long.

For Isolation, we notice that decreasing the cutoff can lead to more search work before finding a solution. When reducing the cutoff from 32 to 31, the shortest solution length increased from 266 to 315, and the search work increased by 4.5%.

We suspect that the savings in search effort are much larger for puzzles with in larger grids with larger search spaces. The only real indication we have for this suspicion is the fact that we solved Magnolia using BFS, with space value pruning as the only enhancement. We chose the lowest cutoff that enabled us to solve the puzzle, the search examined 487 million positions. Hence, when we decreased the cutoff by one, we couldn't find any solution after searching through the whole reduced search space. When we increased the cutoff by one, we ran out of memory after examining 525 million positions. In fact, Magnolia was the only puzzle in our test suite where this pruning

Table 6.10: The effect of the resting blocks pruning for Hyperion

	None	Relaxed	Strict
Solution length	159	164	164
Work needed	631,845	643,796	642,024

was crucial for solving.

As mentioned in section 6.4.2, the space value function was incorporated into our A* heuristic function as a penalty. Combined with another penalty for the position of the spaces, we managed to solve additional puzzles like Triathlon and Climb game 15D with A* (even though we solved the latter puzzle with standard BFS).

We consider this a worthwhile enhancement to both the BFS and A* algorithms.

6.4.4 Pruning: Resting blocks

This improvement comes in two variants, relaxed and strict. The difference is that in the strict mode, the first hanging block found during the scan of the position has to be moved, if any hanging blocks exist. In relaxed mode, any of the hanging blocks can be moved. If no hanging blocks exist in a position, every legal move is permitted and no pruning occurs.

It is hard to measure the usefulness of this improvement. When we tested it on large puzzles, we saw huge reductions in the search tree. However, as we didn't solve these large puzzles, we cannot know if we these savings would be lost by increasing the length of the solution that would have been found it we could run the search to its conclusion.

On small puzzles with little space, we found that this improvement had little effect. With two spaces, there is no difference in the relaxed and strict modes, as there can be at most one hanging block. Tests performed on Hyperion show that the search work is increased, due to an increase in the solution length. Table 6.10 shows the data. Tests performed on Easy, Forget-me-not, Isolation and The Devil's nightcap show that almost every position in the search space can eventually be reached, but the sequence of moves required to reach a given position might increase. Hence, the savings in the search space for small puzzles are negligible. We don't know if this is the case for larger puzzles.

Table 6.11 shows the size of the portion of the search space examined after searching to depth n for the puzzle Ithaca, for some selected search depths. We see that the savings in the search tree size are huge, but as mentioned

Table 6.11: Search space sizes after depth n on Ithaca

n	None	Relaxed	Strict
5	4743	753	623
10	563804	52871	40609
15	28196019	1707573	1205741
16	58264526	3225476	2231504
17	-	6012024	4075861
18	-	10972739	7366767
19	-	20326280	13217385

previously, we don't know if we lose the savings because we might find a solution at a later search depth.

However, we believe that this enhancement could have been useful if we were able to search further into the puzzles with lots of space, like Ithaca. If there are hanging blocks in a position, more hanging blocks cannot be introduced, except as a consequence of moving an existing hanging block.

However, another disadvantage is that this is a rather expensive enhancement, it increases runtime by a bit since the position has to be scanned for hanging blocks before performing child generation.

6.4.5 Pruning: Never move the master block away from its goal

When this option was turned on, a move that increased the Manhattan distance of the master block to the goal was disallowed.

We did not use this option on many puzzles. When we used it and it worked, the savings in search work were not significant, and it didn't enable us to solve more puzzles. The only positions removed from the search tree are those positions where the master block is not contained in the bounding rectangle containing the start position and the goal position of the master block. In some puzzles, this results in very small savings. One such case is Ithaca, where the master block never needs to move back, but the only positions removed from the search space are those there the master block is in the bottom row, which accounts for around 16.7% of the positions.

There aren't many puzzles where there exists a solution such that each move of the master block decreases the Manhattan distance to the goal. Eligible puzzles for this improvement are Easy, Still easy, Rose, Ithaca, Sunshine, Little sunshine. It is unknown whether it is possible on other puzzles. For Sunshine and Little sunshine, we manage to reduce the search space by

90.9%. However, this is a part of the search space that is likely to not be examined by an A* algorithm, so these savings are not useful.

6.4.6 Plan with multiple subgoals

We ended up not using this improvement very much. In most puzzles, it is not clear what route the master block is going to take. However, it is easier to identify problematic blocks and figure out that it would be a good strategy to place them in a corner where they won't be in the way of the master block.

We used this feature on three puzzles only - and it was not essential in solving more puzzles. For Magnolia, we made plan with 7 subgoals where the master block followed the same path as in our manual solution to this puzzle. We didn't specify any of the other blocks in the subgoals. Our program (using BFS) was able to reach the second subgoal, but not able to find the third subgoal. It means the master block managed to move around a third of its total length to the final goal. We eventually managed to solve Magnolia without subgoals.

For Corona, we made a plan with 6 subgoals where the master block take the shortest way around the walls to the goal. Our program managed to reach the second subgoal, but not the third. It managed to move the master block to the top center of the board, but not the next intermediate position exactly in the middle of the board. The master block was moved one third of the total path length to the goal.

For Paragon 1FG we made a slightly different plan than for the two puzzles above. We made one intermediate subgoal (in addition to the actual goal), where the master block had moved past the 3×2 block. We didn't mange to solve the puzzle with this plan. In fact, the first subgoal wasn't reached.

One weakness with this enhancement is that when we reach a subgoal, the first position reaching the subgoal is used at the starting position for the search towards the next subgoal. This position might not be a very good one, and in some cases there might not exist a path from this position to the goal without backtracking the steps that was made in order to reach this position.

We suggest two ways of improving the performance of this enhancement:

- Let the user specify a more detailed plan. To avoid reaching a subgoal with a position where it is difficult to make further progress, the user can specify additional constraints that must hold in order to reach the subgoal. Additional constraints can include the position of problematic blocks.

- Change the algorithm. Instead of taking the first position, we can search until we have n positions satisfying the subgoal. Some checks can also be made to avoid having positions that are too similar.

We did not benefit from this enhancement in the tests we did on the puzzles in our test suite. For any given puzzle, we can always start with a plan and make it more detailed until we solve it, but this way of solving a puzzle is less than satisfying. A more interesting problem is the automatic construction of such a plan, based on analysis on the starting position.

6.4.7 Queue management

When this option is turned on and the size of the queue exceeds a given threshold, it is emptied and only the q best positions are kept. The positions are ranked according using our h_0 heuristic - Manhattan distance of master block to goal. In the case of a tie, the positions that occur earlier in the original queue are picked.

This is another improvement we ended up using very little. We weren't able to solve any new puzzles using this improvement.

The main problem is that we chose to use a simple heuristic when picking the best position. After a few queue flushes, the queue often ends up containing only positions where the master block is at the same place. In this case, the first positions are always picked from the queue and we eventually enter an infinite loop.

This can be improved by refining the criteria for picking the best positions to keep. We could instead come up with a better heuristic, or we could even define some criteria specific to the puzzle we are trying to solve. Instead of keeping exactly q positions, we could instead drop all positions satisfying some criteria describing a bad position, for instance if the master block hasn't moved at all.

6.5 Summary

With BFS, we managed to solve all the puzzles with small search spaces, and the puzzles where a solution occurs early in the search tree. We also managed to solve the puzzles where a solution occurs after examining as much as 10^9 positions. We managed to search this far because of the lower memory requirement for our BFS algorithm, compared to our A* algorithm.

With A*, we managed to solve all with puzzles containing a small amount of spaces (2-6) and where it is easy to rearrange the blocks. In these puzzles,

it is possible to move the master block steadily towards the goal, and these puzzles don't require major reshuffling of the other blocks at any stage.

Of the 26 puzzles in our test suite, 11 of them were not solved by our program. The following list contains characteristics that occurred in many of the puzzles we didn't manage to solve.

- The search space sizes are typically larger than 10^{16}.
- The puzzles feature large blocks, or many irregularly shaped blocks.
- The blocks are hard to move around, many moves lead to positions where no progress is eventually possible.
- It is hard to advance the master block towards its goal, it may require a high number of intermediate moves between each move where the master block advances.
- Long solution sequences.
- Excessive amounts of space.
- The search spaces can have dead ends. Moves can lead to positions where no further progress is possible. This includes positions where the master block is close to its goal.

Our program managed to move the master block as close as a Manhattan distance of 2 to its goal on some of the puzzles. Having such a position does not mean that we are close to a solution. In some cases, our program managed to move the master block close to the goal early, but still didn't manage to solve the puzzle.

Even with a memory-efficient representation of a position, running out of memory was our biggest problem. This was a larger obstacle for us than the execution time of our program, even though our program was not very efficient in terms of speed.

Here is a summary of the various heuristics and enhancements we used.

Admissible heuristic

Our admissible heuristic function, returning the Manhattan distance between the master block and its destination, was very far from being a good fit to the actual cost function. In the worst cases, it returned values several orders of magnitude lower than then actual cost. This heuristic didn't contribute towards solving new puzzles.

Non-admissible heuristic

Our non-admissible heuristic function was a very valuable addition to our program. The key was that in addition to using the original admissible heuristic and scaling it in order to making it match the actual cost function better, we added penalties based on the positions of the spaces. This heuristic ensured that the A* algorithm was given much better guidance towards goal states, reducing the search work by a factor of up to 3000. Non-admissible heuristics enabled us to solve 4 puzzles which we couldn't solve with other improvements. The disadvantage of this heuristic is that it involves a lot of experimentation to find good parameters to use. Using this heuristic with the wrong parameters results in much more search work, and in the worst cases more work than the BFS algorithm needs.

Pruning the search space: space value

The purpose of this improvement was to cut off positions with too scattered spaces from the search space. This improvement worked very well in some cases. For puzzles taking place in a large grid containing few spaces (2-4), we believe this improvement results in a large reduction in the search space. This reduction was sufficient to make one new puzzle solvable that we couldn't solve without this improvement. The disadvantage is that experimentation is required in order to find a good cutoff threshold. If the threshold is too low, the puzzle becomes unsolvable and if the threshold is set too high, the search space savings are no longer significant.

Asserting that some blocks never need to be moved

This is not an algorithmic improvement as such, but rather a modification that could be applied to a puzzle before we ran our program. The purpose of this improvement is to analyse the starting configuration of a puzzle before attempting to run our program on it, and conjecture that some blocks don't need to be moved in order to solve the puzzle. Then, the input file to the program is modified such that these blocks are locked into place. This improvement doesn't apply to many puzzles, because in most cases, every block need to be moved at some point. Still, we gained one new puzzle solved with this improvement. On the few puzzles that this improvement works, massive reductions in search space size are achieved.

Table 6.12: The contribution of the various improvements

Improvement	New puzzles solved
Non-admissible heuristic	4 (American pie, Still easy, Rose, Triathlon)
Pruning based on space value	1 (Magnolia)
Locking of blocks	1 (Turtle)

Pruning moves in the presence of hanging blocks

The purpose of this improvement was to reduce the amount of *hanging blocks* in a position at any one time. This improvement didn't work as well for us. It resulted in a slightly lower branching factor, but at the same time increased the solution length. These two effects had the tendency to cancel each other out, resulting in no savings in search work needed. We didn't manage to solve any more puzzles using this improvement.

Subgoals

The purpose of this improvement was to allow a puzzle to have multiple subgoals before the final goal position. Unfortunately, this did not work very well for us. We believe that one implementation detail made this improvement worse: When reaching a subgoal, a new search with the next subgoal as its goal was started, using the first position that reached the previous subgoal as the only element in the queue. This one position could be a dead end in the search space. We didn't manage to solve new puzzles using this improvement.

Pruning: Don't allow the master block to move away from its goal

This improvement didn't work well for us, as in most puzzles, the master block needs to make moves that increase its distance to the goal.

Emptying the queue when it becomes too large

This is another improvement that didn't work well for us. In order to find the best positions to keep, we ranked them according to the distance of the master block to its goal. We believe this criterion was too coarse. We did not manage to solve any new puzzles using this improvement.

Table 6.12 shows how much each improvement contributed. Each improvement is listed, along with the puzzles we wouldn't be able to solve without this improvement.

To wrap up this chapter, we present the main lessons we learned.

- The complexity of this domain is formidable, resulting in large search spaces which represent a significant challenge. As a consequence, the bottleneck of our program was memory limitations rather than execution time.
- Domain-specific improvements are needed in order to overcome the complexity. All of the 6 new puzzles solved can be attributed to domain-specific enhancements.
- Designing a good heuristic function is of utmost importance. Our heuristic function, which reduced the search work by up to a factor of 3000 compared to BFS, was not able to do much progress on the harder puzzles. Our heuristic function was not armed with enough knowledge about the domain. It didn't know about the concept of having small and easily movable blocks near the master block, which we identified as a reason for not being to solve more puzzles.
- Small adjustments in the parameters for our enhancements can have a large impact on the result. We solved one extra puzzle because we found a working parameter value. Values in the neighbourhood of the value we used didn't work.
- We learned that one cannot use the same algorithm as others have successfully used, and expect this to be equally successful. We implemented IDA*, but its performance was less than satisfactory. There are at least two reasons for this: The growth of our search trees don't fit the IDA* algorithm. A solution can occur at the end of the search tree when we have searched through most of the search space, and at this stage, there can be less nodes at depth $i+1$ than at depth i. Another reason is that our admissible heuristic gave too low estimates to the solution length, resulting in many iterations. Our non-admissible heuristic caused IDA* to behave more erratic. In either case, IDA* needed much more resources to achieve the same results as A*.

Chapter 7

Future work

In this section we will identify some of the problems remaining in our program. We will also suggest some ways to enhance the program.

7.1 Improved heuristic function

The A* and IDA* algorithm is currently suffering because of a non-tight admissible heuristic. In puzzles like Isolation, our admissible heuristic estimate is as low as 1.1% of the actual cost. A much better admissible heuristic will be of great benefit for A* and IDA* and could lead to more puzzles being solved.

In addition, the non-admissible heuristic we used could still be improved. We concluded that we couldn't solve a couple of puzzles because our heuristic function doesn't know about the blocks between the goal and the master block. If they are small and easy to move, it can lead to a much easier time finding a solution. If we can add a penalty for less maneuverable blocks in front of the master block in the direction of the goal, we believe that our program could perform better and possibly solve some more puzzles.

7.1.1 Pattern database

In section 4.3.4, we did a little test to determine the usefulness of pattern databases for the puzzle *The Devil's nightcap*. In this case, each database became larger than the search space of the puzzle itself, rendering it useless. It remains to investigate the usefulness of pattern databases on other instances of sliding-block puzzles.

Here are two suggestions on how to make pattern databases.

- Remove blocks until state space in reduced puzzle is feasible for a full breadth-first search. Then, do a full search and store the distance to goal for every position. One problem is to determine which blocks to remove, and another weakness is that the bound might be a very bad fit to $h*$, the actual cost function.

- Replace larger blocks with 1x1 blocks until the search space in the reduced puzzle is feasible for a full BFS. Then do as above. However, if all steps have cost 1, we risk overestimating since we filled our puzzle with many 1x1 blocks. One possible solution is to give each 1x1 block a new cost: $\frac{1}{k}$ where k is the size of the largest block replaced with 1x1 blocks. This can result in a tighter bound than the above suggestion, since we are not adding more space from the puzzle. Some disadvantages are that it might not be possible to bring the search space down to a feasible size for large puzzles, and we cannot do a standard BFS in the search space because the moves now have different costs.

7.2 Transposition table for IDA* with more features

When a good heuristic function is in place, one possible next step is to improve the transposition table.

Currently, the implementation of IDA* has a very simple transposition table which is a set containing positions we have seen so far in one iteration. This enables us to avoid cycles, but no information is carried on between iterations.

The IDA* algorithm will benefit from a transposition table with the following properties:

- Fast lookup - check if a position is in this table.

- For each item in table, maintain a counter which increases for each access.

- Ability to throw out least recently used or least frequently used items.

- Ability to update upper bound for solution length when same position is checked more than once.

7.3 Store the solution

The current implementation doesn't support displaying the solution, nor is the solution stored. Depending on the options used in our program, older search depths are dropped, which makes reconstruction of the solution impossible. Also, no links are stored between parent and child positions.

If we don't drop old positions, we can reconstruct the path by doing a backward search. From the goal positions we found, we can try all moves and try to reach a position in the previous search depth. Repeat until we've found the start position.

7.4 Post-processing

We have found many non-optimal solutions using non-admissible heuristics and pruning that don't guarantee optimal solutions. Especially the space value pruning can lead to contrived solutions.

Given that we have one solution, we can post-process it to make it shorter. For any given pair of positions, one could attempt to find a shorter sequence of moves that connect them.

7.5 Considering moves instead of steps

Change the state transition from steps to moves, and attempt to find heuristics based on this kind of move. The Manhattan distance heuristic cannot be used any more, but it would be interesting to see whether a good heuristic based on pattern databases could possibly perform better using moves instead of steps. Using moves instead shortens the solution length, albeit at the cost of increasing the branching factor.

7.6 Optimisations

This section will list optimisations that can be applied to the existing algorithms, rather than suggestions for new algorithmic enhancements.

7.6.1 Speed

The current program isn't very fast, it is a factor of 6 slower than our old BFS program. It is advisable to write a new program and drop the enhancements

Table 7.1: Number of bits needed to represent positions in some selected puzzles

Puzzle name	Number of bits Huffman coding	Number of bits permutation rank
Easy	19	11
Forget-me-not	26	19
Ithaca	119	100
Still easy	51	39
Isolation	39	31
Paragon 1FG	70	52
The Devil's nightcap	37	28

which were found to be of less use.

7.6.2 State representation using permutation rank

Consider a string containing each block in a puzzle, including spaces. A position can be constructed from a permutation of this string, as mentioned in section 2.6.1.

We can achieve a more compact state representation by using the permutation rank: an integer between 0 and $p-1$, where p is the number of distinct permutations of the string of blocks, which is identical to the *packing bound* we introduced in section 2.6.1. This representation will lead to savings in memory usage in most cases. See the examples of savings below.

The existing state representation is also very memory compact, so this improvement isn't essential, it's just rather nice.

For performance and ease of implementation, a position is stored at the beginning of a byte in memory. The size of the representation is then effectively rounded up to the nearest 8 bits.

Table 7.1 shows the number of bits needed to represent a position using Huffman coding, and the number of bits needed to represent a position using permutation rank, without rounding up.

This representation automatically suggests another way for doing duplicate checks - have a boolean vector of size n, where n equals the packing bound of the puzzle, assuming this vector is possible to store. With such an array, duplicate checks can be done in constant time, simply by looking up the position in the vector, using its permutation rank as the index.

7.6.3 State representation using a trie

Wang [24] suggests another representation, based on the initial permutation of the string which contains each block and space.

Positions can then be inserted into a trie. If two positions share the same prefix for the first p positions, their subtrees will diverse at depth p in the trie. The amount of memory needed to store the trie is dependent on the order of the items in the permutation.

7.7 Macro moves

This improvement can potentially be very beneficial. One can imagine macro moves taking the master block from one corner to another in a subboard of a puzzle.

However, it is not clear how to define a macro move and when it should be applied in order to save search space size. Nevertheless, macro moves have been used with success in other domains like Sokoban, and it should be investigated further whether it is useful for sliding-block puzzles.

7.8 Parallelism

One can't overlook the potential savings of dividing the work between tens or hundreds of processor cores. However, care must be taken in the design of the algorithm and memory layout. In general, converting serial programs to parallel programs is hard.

Most of the steps in a BFS algorithm with delayed duplicate checking (like our old solver) can be parallelised. Each position in the queue is independent of the others, and the children positions of each position can therefore be generated in parallel. The duplicate check itself can also be parallelized, since parts of the queue can be checked for duplicates independently. The last step of doing a mergesort on the files that together comprise the complete search depth is harder to parallelise, but there exist efficient parallel algorithms for mergesort that run faster than a sequential mergesort.

7.9 Further potential for improvements

In this project, one person has been working for a limited time doing research and implementing a program that solves sliding-block puzzles.

We believe there is a vast potential for applying existing methods and discovering new improvements that can push the best results in this domain even further. We have not yet looked outside the field of search algorithms.

We believe that a sliding-block puzzle solving program can benefit from other areas of artificial intelligence, like planning.

Chapter 8

Conclusion and summary

In this project, we have looked at how we can construct a computer program that solves sliding-block puzzles.

We looked at the available published literature and looked at how others have solved sliding-block puzzles and similar problems before, and learned about some algorithms and methods that could be used.

Then, we did an analysis of our domain, and found some properties that could be used in order to make our implementation more efficient. We also suggested some domain-specific ideas to implement. We selected 26 puzzles from different sources, which became our test suite, the puzzles on which we would try out our new methods.

During the domain analysis stage, we discovered an efficient method of calculating the exact number of possible configurations for a given puzzle. This enabled us to get a upper bound of the search space size that was two to five orders of magnitude lower than a simple bound using combinatorics. We managed to calculate this upper bound for two thirds of the puzzles in our test.

We then implemented a sliding-block puzzle solving program using the algorithms and methods we found during the literature study, and the domain-specific ideas we developed during our analysis of the problem domain. We implemented BFS, A* and IDA* with several improvements that should reduce the search space size.

We then measured the performance of our program against a very efficient BFS implementation, as well as measuring the progress of our program on the difficult puzzles it didn't manage to solve.

With some of the enhancements we implemented in our program, we were able to significantly reduce the computational work needed in order to solve

easier puzzles. We also managed to reduce the search space size by eliminating non-promising positions.

Our program managed to solve 6 new puzzles that the efficient BFS program wasn't able to solve. The biggest contribution came from our domain-specific improvements. The non-admissible heuristic used in the A* algorithm had the largest impact on the number of new puzzles solved. The heuristic function estimates the distance to a solved position based on the position of the master block and its distance to its destination, the positions of the spaces relative to the master block and how scattered the spaces are in the grid. Other valuable domain-specific improvements include pruning positions with scattered spaces from the search space, and not allowing certain blocks in a puzzle to move, which also greatly reduces the search space size.

Despite the numerous reports of using IDA* with success in other similar single-agent domains, IDA* performed poorly at solving sliding-block puzzles. The two main reasons we could identify were the nature of our non-admissible heuristic function and the behaviour of the search trees generated from the search space graphs. In our experience, our heuristic function gave the best results when we allowed it to overestimate the cost to a goal node. Overestimation causes IDA* to wrongly increase its limit for the maximal solution length (pathlimit), resulting in extra work. The other reason is that the search tree grows exponentially at first, but this growth is often reduced later in the search tree, and sometimes the size of a given search depth shrinks compared to the previous search depth. The efficiency of IDA* is based on the assumption that the work done in the last iteration will dominate the sum of the work done in all previous iterations. This assumption does not hold for our domain.

Sliding-block puzzles turned out to be a very complex domain, which resisted the use of traditional methods. Traditional algorithms with broadly applicable heuristics (like the Manhattan distance) can at most give small reductions in search effort. We were dependent on the use of domain-specific enhancements to overcome the complexity.

There are still a lot of interesting ideas to explore in this domain. We only got the chance to explore a few of the ideas we had. We refer to chapter 7 for some areas where additional research can be beneficial.

Chapter 9

Acknowledgements

Thanks to Tor Gunnar Høst Houeland for being my supervisor in practice. He provided valuable feedback and pushed me to be more productive so that I could actually finish in time. Thanks to Helge Langseth for being my formal supervisor and for accepting my own idea as a project.

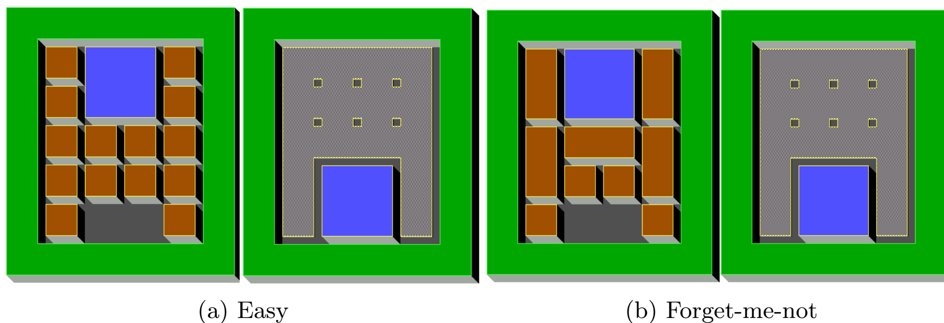
Thanks to Anne C. Elster for providing access to the powerful computers at the HPC laboratorium.

Appendix A

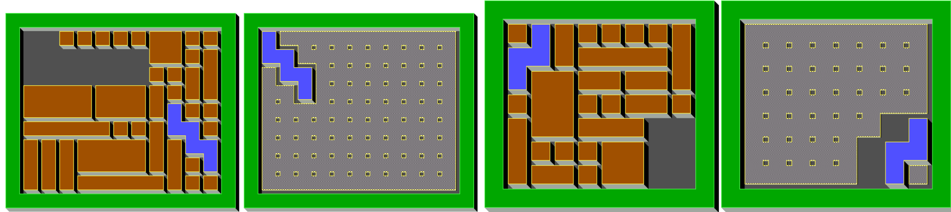
Test suite

Each figure consists of two positions: The one to the left shows the starting position, and the one to the right shows the goal condition. The shaded gray area is "don't care" - only the blocks and spaces specified are required to be at the designated positions.

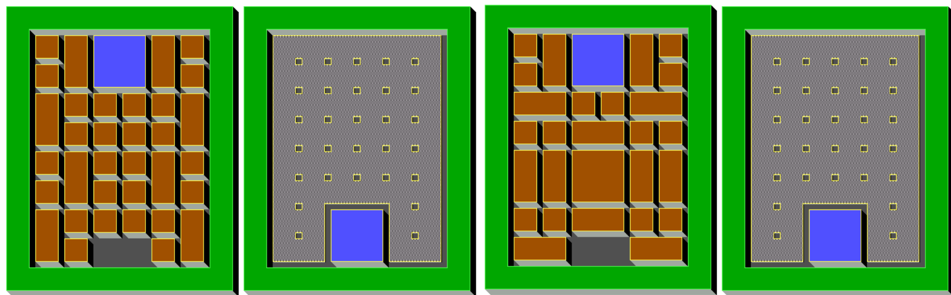
Many of these puzzles are slightly modified. Some of the puzzles originally contained special elements not included in the domain as we have defined it. In almost every case, this resulted in a slight relaxation in the constraints of the original puzzle. There is one exception, our version of Turtle is much harder than the original version.



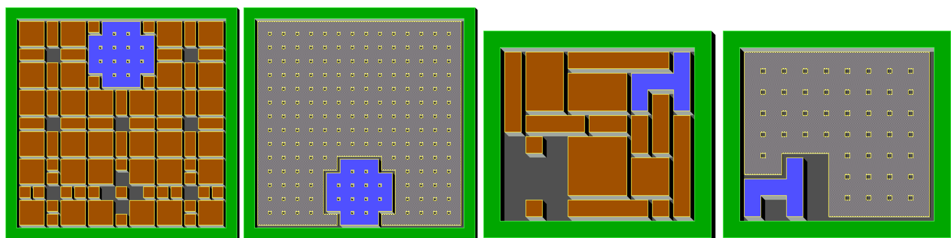
(a) Easy (b) Forget-me-not



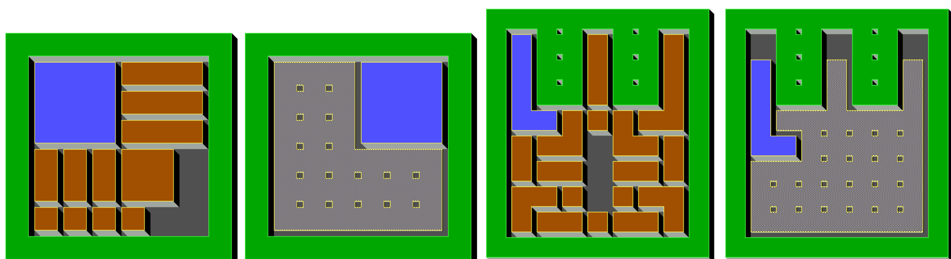
(c) Ithaca (d) American pie



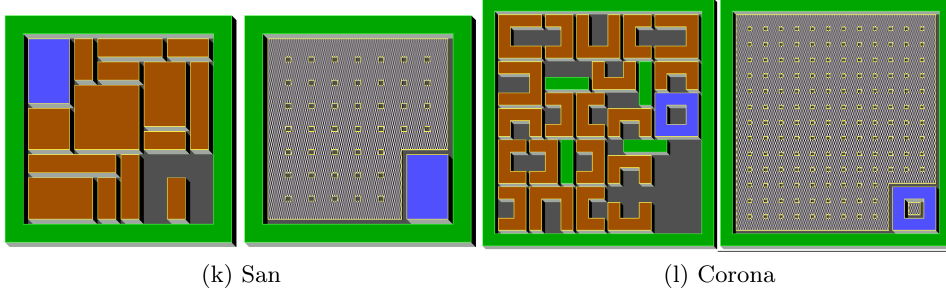
(e) Still easy (f) Rose



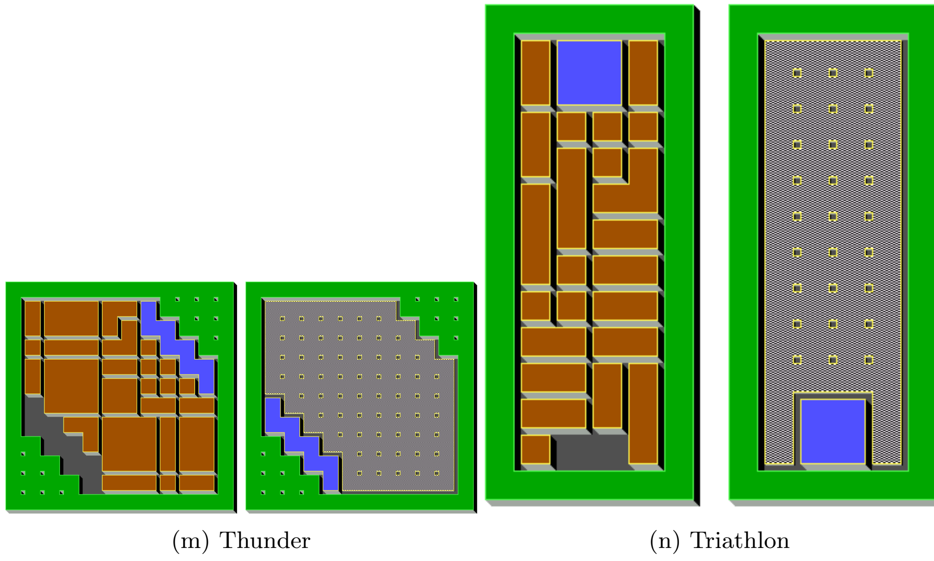
(g) Little sunshine (h) Chair



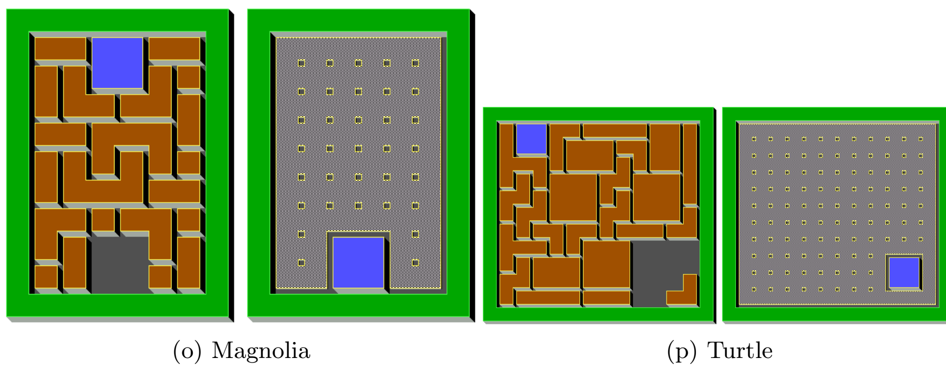
(i) Isolation (j) Hyperion



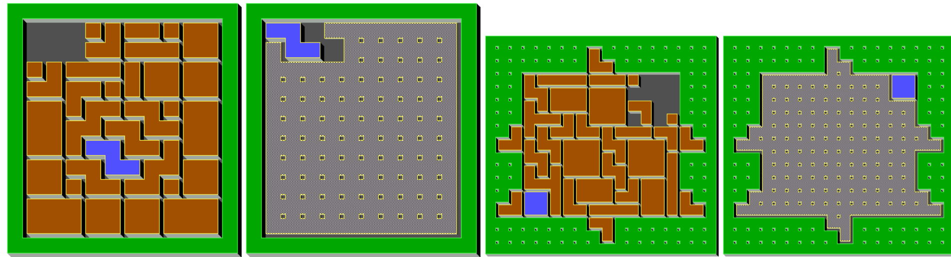
(k) San (l) Corona



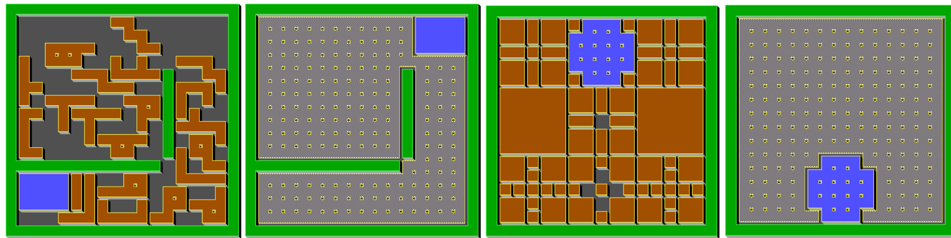
(m) Thunder (n) Triathlon



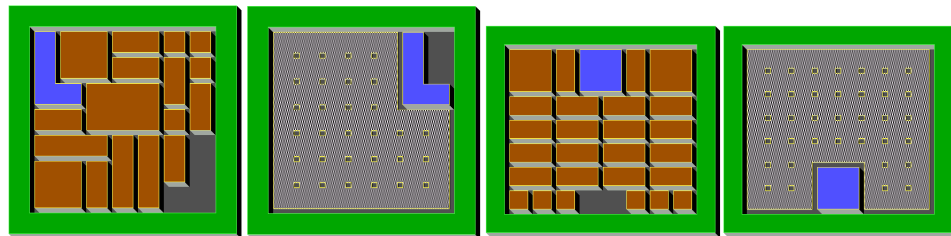
(o) Magnolia (p) Turtle



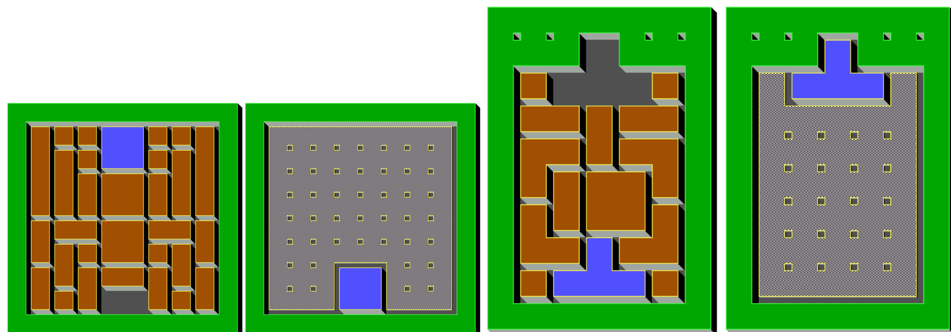
(q) Apple (r) Schnappi



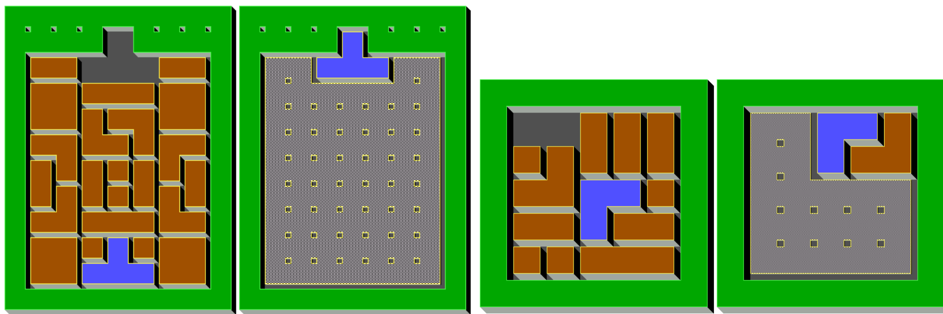
(s) Salambo (t) Sunshine



(u) Paragon 1FG (v) Warmup



(w) Get ready (x) Climb game 15D



(y) Climb pro 24

(z) The Devil's nightcap

Bibliography

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

[2] Joseph C. Culberson, *Sokoban is PSPACE-complete*. Technical Report TR97-02, Department of Computing Science, University of Alberta, 1997.

[3] Joseph C. Culberson, Jonathan Schaeffer, *Pattern databases*. Computational intelligence, 14(4), pp.318-334, 1998.

[4] Erik D. Demaine, Robert A. Hearn, *Playing games with algorithms: Algorithmic combinatorial game theory*. Games Of No Chance 3, MSRI Publications, pp.3-56, 2009.

[5] Robert A. Hearn, Erik D. Demaine, *Games, Puzzles & Computation*. A K Peters, 2009.

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

[7] Andreas Junghanns, *Pushing the limits - new developments in single-agent search* (Ph.D. Thesis). University of Alberta, Department of Computing Science, 1999.

[8] Jon Kleinberg, Éva Tardos, *Algorithm design*. Addison Wesley, 2005.

[9] Richard E. Korf, 1985, *Depth-first iterative-deepening: An optimal admissible tree search*. Artificial Intelligence 27(1):pp.97-109, 1985.

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

[11] Richard E. Korf, *Finding optimal solutions to Rubik's cube using pattern databases*. Proceedings of the national conference on artificial intelligence, pp.700-705, 1997.

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

[13] Richard E. Korf, Peter Schultze, *Large-scale parallel breadth-first search.* Proceedings of the national conference on artificial intelligence, 2005.

[14] Jim Leonard, "JimSlide", http://xuth.net/jimslide/, retrieved on 12.12.2009.

[15] Nolan Pflug, Post on internet forum, http://www.forumromanum.com/member/forum/forum.php?action=ubb_show&entryid=1074412595&mainid=1074412595&threadid=2&USER=user_89109, retrieved on 10.12.2009.

[16] Tomas Rokicki, *Twenty-five moves suffice for Rubik's Cube.* http://arxiv.org/abs/0803.3435, retrieved on 22.09.2009.

[17] Tomas Rokicki, *Twenty-two moves suffice.* http://cubezzz.homelinux.org/drupal/?q=node/view/121, retrieved on 22.09.2009.

[18] Karl Rottmann, *Matematisk formelsamling.* Spektrum forlag, 2003.

[19] Andreas Rottler, "Bricks Game Home Page", http://www.bricks-game.de/, retrieved on 12.12.2009.

[20] Stuart Russell, Peter Norvig, *Artificial intelligence - a modern approach.* Prentice Hall, 1995.

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

[22] "The Sliding Block Puzzle Page", http://www.puzzleworld.org/slidingblockpuzzles/, retrieved on 12.12.2009.

[23] Jared Weinberger, "New & Old Sliding Block Puzzles", http://www.brainyday.com/jared/flash/sliders.html, retrieved on 12.12.2009.

[24] Posts on the Bricks website discussion forum, http://www.forumromanum.com/member/forum/forum.php?q=bricks_solver_software-bricks_game&action=ubb_show&entryid=1103203051&mainid=1103203051& USER=user_89109&threadid=1088594388, retrieved on 10.12.2009.