UNIVERSITÄT PASSAU

Fakultät für Informatik und Mathematik

Efficient Solutions for Sliding Block Puzzles

Bachelor Thesis by
Till Wübbers

Under the supervision of
Prof. Dr. Martin Kreuzer

Passau, February 2019

Contents

1 Zusammenfassung

In dieser Arbeit werden die Algorithmen *A**, *Iterative Deepening A** und *Near-Optimal Bidirectional Search* zum Lösen von Schiebepuzzles beschrieben. Um die Algorithmen anwenden zu können, wird das Puzzle dabei als Graph von Spielpositionen definiert. Da die drei Algorithmen Heuristiken benötigen, wird die *Manhattan Distance Heuristik* angepasst, um für Schiebepuzzles zu funktionieren. Außerdem wird die *Linear Conflict Heuristik* benutzt und durch eigene Anpassungen verbessert. Die Funktionsweise der Algorithmen wird erklärt und die theoretischen Unterschiede in Geschwindigkeit und Speicherverbrauch werden verglichen.

Für die Arbeit wurden die Algorithmen in der Programmiersprache Rust implementiert. Ein Komandozeilenprogramm zur effizienten Auswertung, sowie eine Webseite für eine visuelle Darstellung wurden erstellt. Während der Implementierung wurden verschiedene Methoden zur Optimierung des Programmcodes angewendet. Die Laufzeit und Zahl der besuchten Knoten der Algorithmen bei verschiedenen Puzzles wird mit dem Programm gemessen. Die Puzzles sind in Kategorien aufgeteilt und die Algorithmen werden für jede Kategorie verglichen.

2 Introduction

Sliding block puzzles are a game of moving puzzle pieces to a goal position on a board. The classic "15-puzzle" is played on a four by four squares grid, with 15 pieces that have a part of an image printed on them. The goal is to move the pieces in such a way that they combine into a bigger image. A more complicated sliding block puzzle is "Rush Hour", where the board consists of differently shaped cars that can only move on one axis. To solve the puzzle, the red coloured car has to be moved to the right edge of the board.

The goal of this thesis is to assess three algorithms that are capable of finding an optimal solution for any sliding puzzle, the *A**, *Iterative Deepening A** and *Near-Optimal Bidirectional Search* algorithms. We will compare the amount of nodes each alorithm expands, as well as their respective memory and time complexity.

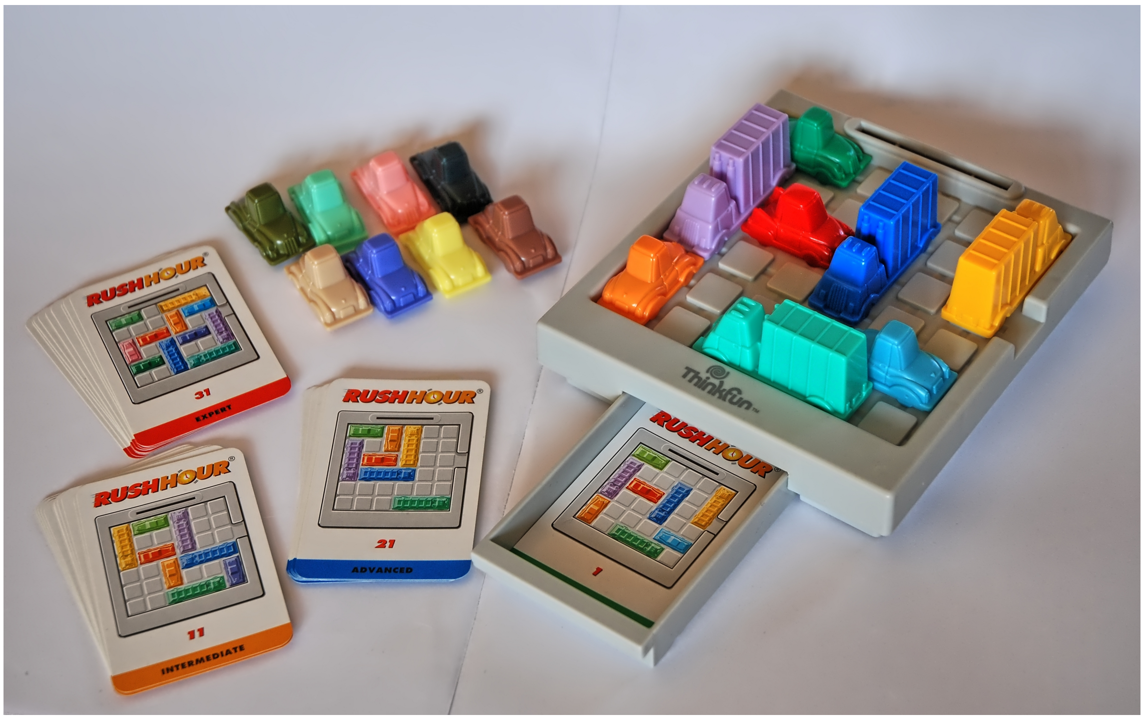


Figure 2.1: The Rush Hour game

2.1 Rules

Sliding block puzzles are played on a two-dimensional grid that is filled with rectangular pieces of varying size. These can be moved around, where one move constitutes a single vertical or horizontal step. Pieces cannot move on top of one another and cannot slide diagonally in one step. The same puzzle will always have the same pieces, a position is a specific arrangement of these pieces on the grid. A puzzle will always have exactly one start position and one or more goal positions. It is considered solved when a series of moves is found that connects beginning to end. It is possible that there exists no connection like this. Therefore some puzzles are unsolvable.

2.2 Pieces

A piece $p = (l_p, g_p)$ has a current location $l_p \in \mathbb{N}^2$ and a goal location $g_p \in \mathbb{N}^2$. Pieces are rectangular and can span multiple fields of the grid, the location of a piece is defined as the x and y coordinates of the top left corner. When every piece on the board is at its goal location, the puzzle is considered to be solved.

2.3 Puzzle graph

The search algorithms in this thesis search for a path between nodes of a graph. To make the problem of solving a sliding puzzle applicable to these algorithms, we need to define a graph describing the different positions and their relationships, as seen in Figure 2.2.

$$G = (N, E)$$

The set of nodes N contains all possible positions of a given puzzle. Every edge in E describes how a single piece of the puzzle gets moved one step, and therefore connects one position to another position in N. The start and goal positions of a puzzle are given by $s \in N$ and $g \in N$. Any sliding puzzle can now be specified by the puzzle graph G and the nodes s and g.

2.4 Paths

Since solving a sliding puzzle requires a path from the start position to the goal position to be found, we require a definition of how such a path would look. Any path on the graph can be described as a tuple of edges $(e_1, ..., e_n) \in E^n$. A solution with n steps on a graph (N, E) is defined as a path $E^* = (e_1, ..., e_n) \in E^n$, with the conditions that $e_1 = s$ and $e_n = g$.

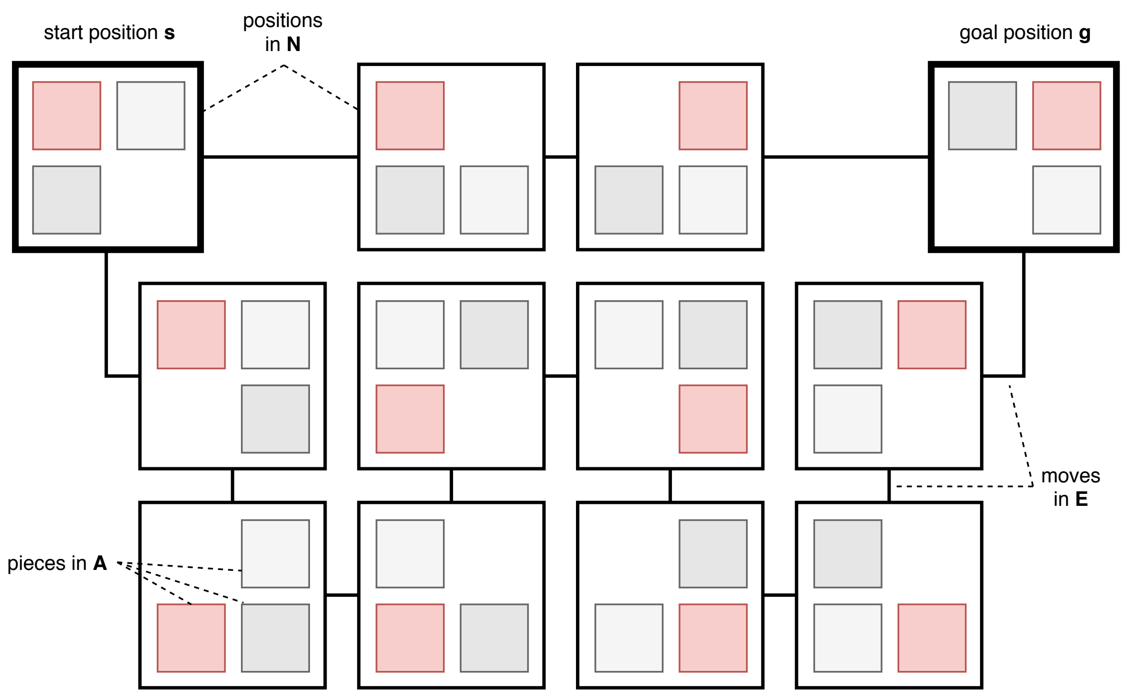


Figure 2.2: Graph of a simple puzzle

3 Heuristics

Most search algorithms require a heuristic function to estimate the smallest number of moves that are needed to reach the goal position. We will call this the *goal distance*. This information is used to direct the search closer towards the goal in each step. In the case of sliding puzzles, the distance of any single piece to its intended position can be estimated as well.

On a graph (N, E) with goal g, any heuristic function $h(n)$ with a board position $n \in N$ will estimate the smallest number of moves from n to g.

3.1 Admissibility

As defined in [1], page 94, "an admissible heuristic is one that never overestimates the cost to reach the goal". Let $h^*(n)$ be the exact distance from n to g. A heuristic $h(n)$ is admissible if $\forall n : h(n) \leq h^*(n)$.

3.2 Consistency

Let s be a board state and s' be any successor of s. A heuristic $h(n)$ is *consistent* or *monotonic*, if $h(s) \leq h(s') + c(s, s')$ where $c(s, s')$ is the cost of moving from s to s'. Additionally, every consistent heuristic is also admissible (see [1], page 95).

When applied to sliding puzzles, the condition can be simplified. Since s' is a direct successor of s and moving along a single edge means moving one step, the cost is always 1. This means the condition will always be $h(s) \leq h(s') + 1$.

3.3 Manhattan Distance

To estimate the smallest amount of moves needed to connect two points $n = (x_1, y_1)$ and $g = (x_2, y_2)$ on the $\mathbb{N} \times \mathbb{N}$ grid, we can use the Manhattan distance heuristic:

$$m : (\mathbb{N}^2, \mathbb{N}^2) \to \mathbb{N}, (n, g) \mapsto |x_2 - x_1| + |y_2 - y_1|$$

This may be used as a guess of the goal distance for a single piece, and will be exactly correct if there are no other pieces blocking its way.

To estimate the goal distance of an entire game position $n \in N$, we can add up the Manhattan distances of all pieces in that position. This yields the map

$$m^* : N \to \mathbb{R}, n \mapsto \sum_{a \in A} m(a)$$

3.4 Linear Conflict

Linear Conflict, as defined in [2], section 4.2, is an additional heuristic that works on top of Manhattan distance. It only applies in the special case of two pieces that are in the same row or column. If they have their goal position in that row or column as well, and if additionally the goal positions of both pieces are behind the other piece so that they need to move around each other, at least two extra moves are required.

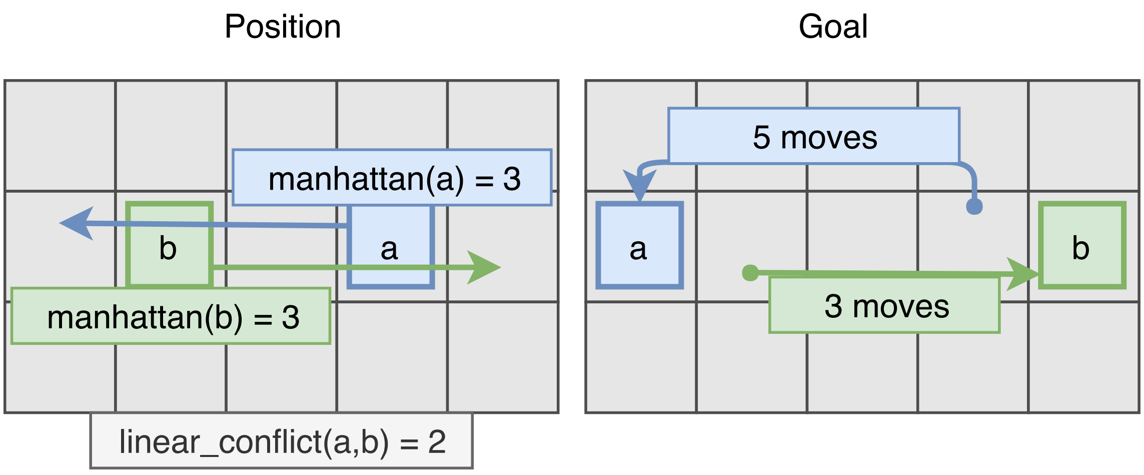


Figure 3.1: Example of a situation where Linear Conflict heuristic applies

3.5 Modified Linear Conflict

For puzzles that have blocks without a goal position, we can create a modified version of Linear Conflict. Let the set of pieces $A = \{a_1, ..., a_i\}$ have goal positions $G = \{g_1, ..., g_i\}$ behind piece b, and let all $a \in A$, $g \in G$ and b be on the same axis. Suppose that piece b does not have a goal. With the definition above, Linear Conflict would not apply in this case, since for any pair (a, b) both pieces need a goal position. However, we can show that at least one extra move is still required in this situation.

For any $a \in A$, let n be the minimal number of moves required to get to the goal, if nothing is blocking the path. There are two possibilities to solve the conflict; either a moves around b, or b moves out of the way.

Let p be the current board state and s be any successor. Let $lc(x)$ be the function of the modified Linear Conflict heuristic. We claim that $lc(p) = 1$ is consistent.

Case 1: a moves around b:
Since both blocks and the goal are on the same axis, a has to make at least two moves on the perpendicular axis. These steps will not move a closer to the goal, meaning that the minimal number of moves is now $n+2$.

This case fulfills the requirement of consistency, namely

$$lc(p) \leq c(p, s) + lc(s)$$

if all $a \in A$ are moved out of the axis, we have $lc(s) = 0$ and $1 \leq c(p, s) + 0 = 1 \leq 1$. Otherwise we have $lc(s) = 1$ and $1 \leq c(p, s) + 1 = 1 \leq 2$.

Case 2, b moves out of the way:
Moving b out of the way does not bring a closer to its goal. This increases the minimal number of moves to $n+1$. Now any other block $a \in A$ can move to the goal without being blocked by b, making this heuristic only applicable once for the set of all $a \in A$ on the same axis.

This case is also consistent, i.e. we have $lc(p) \leq c(p, s) + lc(s)$. The number $lc(s)$ is now 0, since b is not blocking any $a \in A$, and we have $1 \leq c(p, s) + 0 = 1 \leq 1$.

Both cases are consistent and add at least one extra move. Since the Modified Linear Conflict heuristic is consistent, it is also admissible.

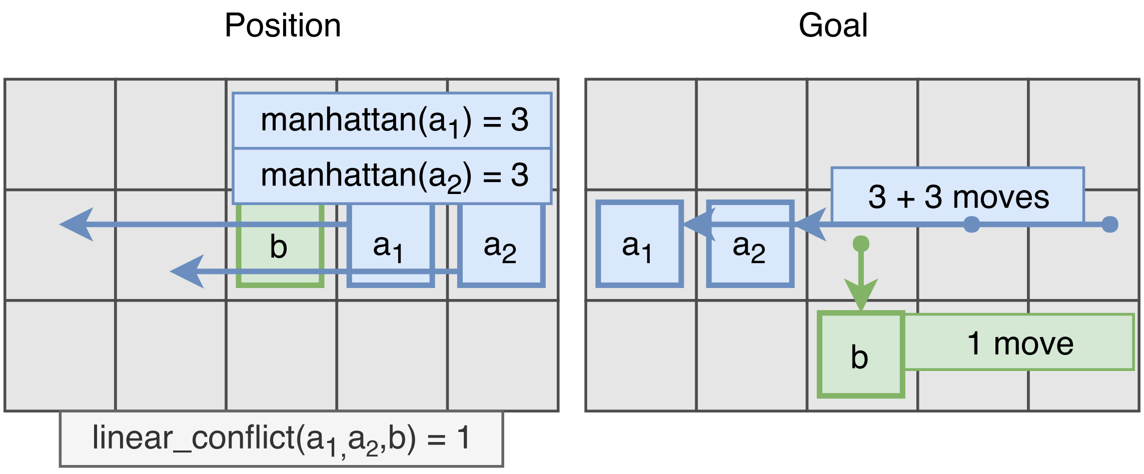


Figure 3.2: Example of a situation where the Modified Linear Conflict heuristic applies

4 Graph search algorithms

Now that we have defined the graph of possible positions and have a heuristic to guide the search, we can use graph traversal algorithms to search for possible solutions.

4.1 Graph shape

A common use-case for the graph search algorithms described in this thesis is pathfinding on a two-dimensional grid. In this kind of problem, the shortest path between a start and a goal point on a grid has to be found. For example the A* algorithm was originally developed to move a robot to it's target. This representation is a useful base for describing the steps an algorithm takes, as every node of the board graph can be described as a simple point on the grid.

Sliding block puzzles can describe the same problem when just having a single block. The solution path is just the shortest list of steps that a block has to take to reach its goal position. Since this representation allows for a more intuitive understanding of how the algorithms traverse the possibility state, we will often use this for the examples in this thesis.

It is to be noted that most state graphs of sliding block puzzles cannot be easily represented by a two-dimensional grid, since multiple blocks cross over the same fields. For example the graph in Figure 2.2 has a board with four fields, but the state graph has more than four states.

4.2 Complexity

The key difficulty of solving sliding puzzles with brute-force methods is the sheer number of possible moves. Some of the example levels that come with the implementation have shortest solution paths of over 100 steps. While this sort of puzzle often only has one or two possible moves at any time, the space requirement of simple algorithms often exceeds physical memory limits. Breadth first search has a space complexity of $O(b^d)$ (see [3], section 2). At a depth of 100 even with a low branching factor b, $O(b^{100})$ will require more than the 16GB of memory the computer which was used for testing had. Therefore more efficient algorithms are needed.

4.3 Depth-first search and iterative deepening

This section is based on the definitions for depth-first search in [4], and iterative deepening in [3], section 4.

On our graph (N, E), we begin at the start position $s \in N$ and move along the first edge starting at this node. The visited node is stored and the process of moving along the first edge of the next node is repeated until either the node has no children, or we reach an already visited node. We then go back and remove all fully explored nodes until there is another node with an unexplored edge. Once we find the goal node, all edges that connect the stored nodes will build the path to the goal. In this way all nodes of the graph are explored eventually, however it is not guaranteed that the solution which is found is an optimal one.

To fix this, *iterative deepening* can be applied. This means that there will be a depth bound $d = 1$, at which the search will reset to the previous node and continue with the next edge that has not been visited. If the entire tree has been searched for a certain depth bound, the number d is increased by one, and the search is restarted.

If there exists a solution at depth x, it will eventually be found when $d = x$, since the entire tree is searched before d is increased. This has the obvious drawback of searching the entire tree for every depth up to the depth of the solution.

4.4 A*

The algorithm uses a *pending* and a *known* set, which are also referred to as *open* and *closed* sets respectively. The pending set contains all nodes that have been found but have not been expanded yet. These nodes are candidates for being expanded in a future step. All nodes that have been expanded in a previous step are stored in the known set. These nodes will never need to be expanded again, since the algorithm already reached this point through an optimal path, and no new knowledge can be gained.

For using the *A** algorithm, we define $c(n)$ as the distance from the start position s to the current node $n \in N$. This will be referred to as the *cost* of n, and will also be used in later algorithms.

We use $h(n)$ as the sum of the Manhattan Distance and Linear Conflict heuristics functions. The function

$$f(n) : N \to \mathbb{R}, n \mapsto c(n) + h(n)$$

therefore describes the estimated distance from the start s via position n to goal g. In every step, the node with the smallest heuristic value is picked for expansion.

When expanding the picked node, we look at all possible child nodes that are not in the known set. If a child node is already in the pending set, the heuristic values of both instances of the node are compared. The version with the smaller heuristic distance is put in the pending set, while the other one is discarded. For each child node added to the pending set, the currently expanding node is stored as it's parent. Once the goal node has been expanded, the path to the goal node can be tracked by recursively following parents, starting from the goal and stopping when reaching the start. Since our heuristic is admissible, the A* algorithm will always find an optimal solution if one exists (see [1], chapter 3.5.2).

Algorithm 1: A*

```
Input  : start node $s$, goal node $g$, graph $(E, N)$
Output: $p$ = $(s,...,g)$, an optimal path from $s$ to $g$,
        or $nopath$ if no path from $s$ to $g$ exists.
// Set of all known (node,parent) tuples
$parents \leftarrow \{(s, \_)\}$
// Set of nodes waiting to be expaned
$pending \leftarrow \{s\}$
// Set of all nodes that have been fully expanded
$known \leftarrow \{\}$

while $pending \neq \varnothing$ do
    // Pick the node in pending with smallest goal distance
    Choose $n^* \in pending$ with $f(n^*) \leq f(n), \forall n \in pending$

    if $n^* = g$ then
        // Generate full path by recursively finding parent
        return $(s, ..., parent(parent(g)), parent(g), g)$
    end

    let $(n^*, n_p^*)$ be the tuple with matching $n^*$ in $parents$
    // Look at all children "o" that are not in known list
    foreach $(n^*, o) \in E$ with $o \in (N \setminus known)$ do
        if $o \in pending$ then
            // Update the parent to current parent, if the current path
               to the node is shorter
            let $(o, o_p)$ be the tuple with matching $o$ in $parents$
            if $d(n^*) + 1 < d(o)$ then
                replace $(o, o_p)$ with $(o, n^*)$ in $parents$
            end
        else
            add $(o, n^*)$ to $parents$
        end
    end
end
return $nopath$
```

4.5 Iterative Deepening A*

The *Iterative Deepening A** (IDA*) algorithm is a graph search algorithm that, like A*, uses a heuristic function to find a path between two nodes on a graph. We can use the same heuristic function as described in the section about A*. One major change in the IDA* algorithm is that there is no *pending* list. The only values that are continuously stored in memory are the depth bound that describes the maximum length a path can have in this step, and the path to the current node.

At first, the depth bound is set to the heuristic value of the start node. Since the heuristic function is required to be admissible, this value is never overestimated and will likely be too small. A depth-first search is used to find the goal node, but every search path is cut off when the heuristic reaches the depth bound. In this case the search will be continued at the next unseen child node. Once the entire tree up to the depth bound is searched, the bound is increased to the smallest known heuristic value, which will be at least $(bound+1)$. When the goal node is found, the current path can be returned immediately as the optimal path.

Algorithm 2: IDA*

```
Input   : start node $s$, goal node $g$, graph $(E, N)$
Output: nopath, if the puzzle is unsolvable.
        $(s,n_1,...,n_i,z)$, if a path from $s$ to $g$ exists.
cutoff ← $h(s)$
path ← $s$
while true do
    $t$ ← search(path, 0, cutoff)
    if $t = true$ then
        return path
    end
    if $t = \infty$ then
        return false
    end
end

Function search(path,depth,cutoff)
    node ← last element from path
    $f$ ← depth + h(node)
    if $f >$ cutoff then
        return false
    end
    if $s = z$ then
        return true
    end
    min ← $\infty$
    for child in children(node) do
        if child not in path then
            Insert child into path
            $t$ ← search(path, depth+1, cutoff)
            if $t = true$ then
                return true
            end
            min = min(min,$t$)
            Remove child from path
        end
    end
end
```

4.6 Comparing A* and IDA*

When applying the A* algorithm on hard sliding block puzzles in practice, memory limitations quickly become apparent. The IDA* algorithm operates in linear space, while the A* algorithm requires exponential amounts of memory (see [5], page 539). The lower space requirement of IDA* comes at the cost of node expansions, as can be seen in Figure 4.1. In this example, the IDA* algorithm did almost four times as many expansions as the A* algorithm.

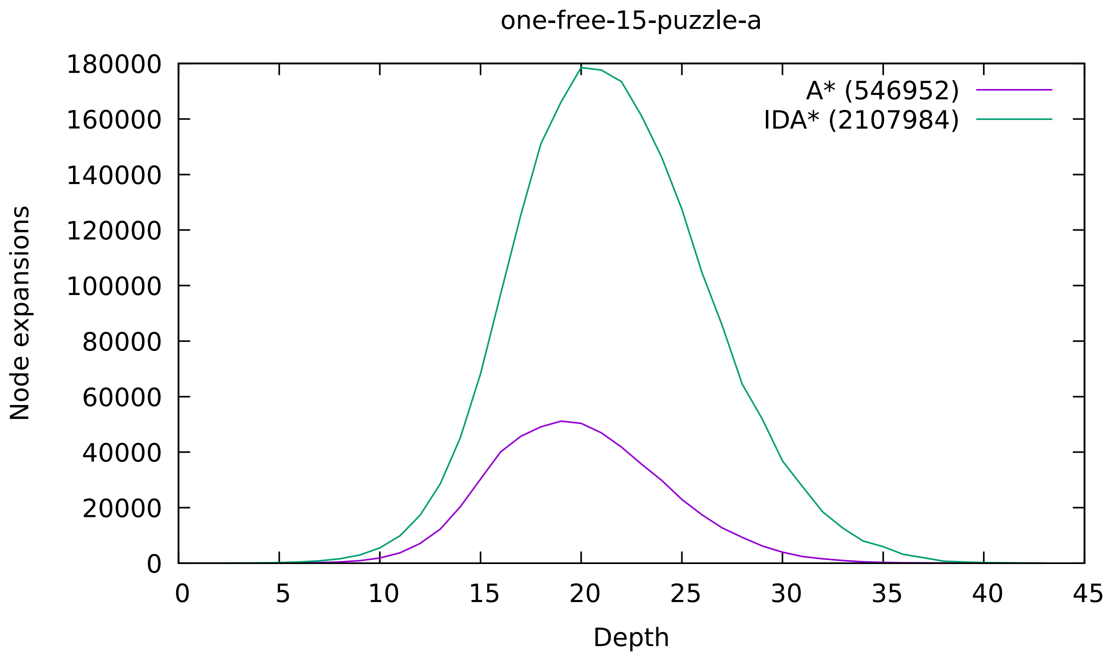


Figure 4.1: The number of node expansions at each depth of A* and IDA* for the puzzle “15-puzzle-a”

5 Bidirectional Search

Bidirectional search is a way to potentially speed up graph traversal by searching from the start and goal position simultaneously. The search can be terminated when the two search spaces intersect. However, some additional conditions are required to make sure the selected path is optimal. The Near-Optimal Bidirectional Search (NBS) algorithm described in [6] is a bidirectional search algorithm that finds optimal paths.

5.1 Goal position

For any kind of bidirectional search, a goal position is needed. This is simple for tasks where there is only a single goal position available. In the case of sliding puzzles this is true for any board where all pieces have a goal position. However, there are also boards with pieces that do not have a goal position and can be in any position when the important pieces reach their goal. This means that on our graph of possible positions, there can be multiple goal nodes.

If we now pick any goal node at random, it is not guaranteed that an optimal path to this node is optimal for all of the goal nodes. Furthermore, there might not even exist any path from the start to this node, making it impossible to find a solution.

For this reason, the following considerations on bidirectional search only apply to the search of boards where every piece has a goal position.

5.2 Near-Optimal Bidirectional Search

The *Near-Optimal Bidirectional Search* (NBS) algorithm has some features that make it useful for solving sliding block puzzles. "As heuristics get weaker, or the problems get harder, the bidirectional approaches improve relative to A*" (see [6], section 8). For some of our puzzles, the heuristic estimate can be a lot lower than the actual shortest path length, making NBS a good approach for solving difficult puzzles.

In Figure 5.1 the step-by-step expansions of the algorithm are displayed. The node expansions themselves work like in any of the previous algorithms, where the surrounding moves are discovered and added to a open list. However, the selection of the nodes which are expanded is very different. Every step of the algorithm a pair of nodes is expanded. This pair is selected from all possible pairs of one forward node U from the forward open list and one backward node V from the backward open list. Each pair (U, V) has a lb-value, which is calculated as follows:

$$lb(U, V) = max\{f_F(U), f_B(V), c(U) + c(V)\}$$

The minimum of all lb-values creates a lower bound on the number of moves required to solve the puzzle.

Out of all pairs, the ones that have the minimal lb-value are selected. Since lb contains the maximum of forward and backward heuristic values, it is possible that on one side of the algorithm a node with non-minimal heuristics is picked. This happens in step 6 of Figure 5.1, where all possible node combinations are selected. Since $min\{f_F(Open_F)\} = 9$ (the minimum heuristic of all open forward nodes) and all nodes in $Open_B$ are smaller or equal to 9, the maximum doesn't exclude any pairs. The condition of $c(U) + c(V)$ doesn't play a role here, since the biggest sum of costs is 8.

From all the pairs that fulfill this condition a random pair is selected, with the following condition: The cost of each node has to be minimal, within its forwards or backwards set of nodes. In step 6 of Figure 5.1 only one node like this exists for both directions, each with a cost of 1. If multiple nodes of a search direction have the same minimal cost, any of these node can be picked.

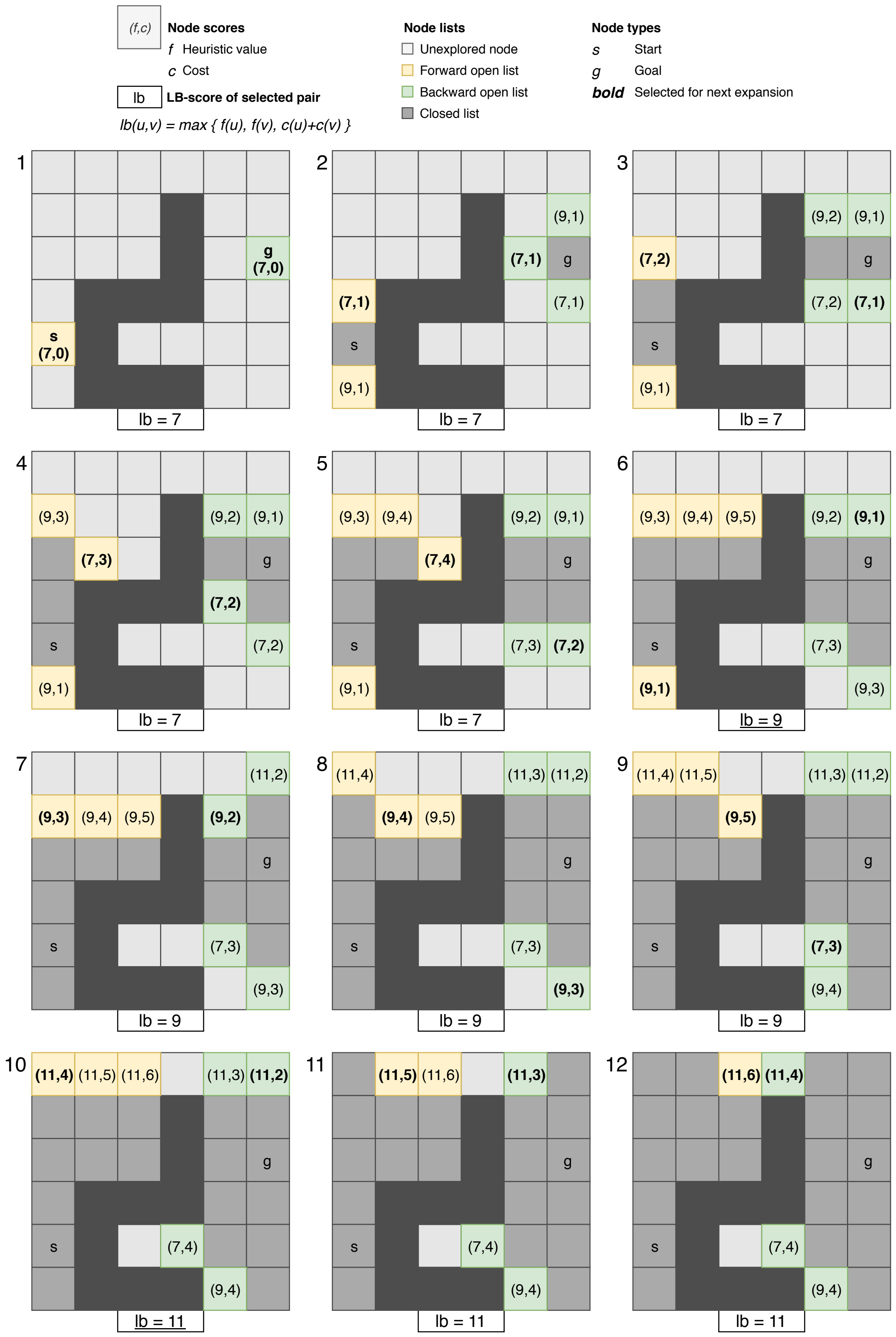


Figure 5.1: Node pair expansions of the NBS algorithm

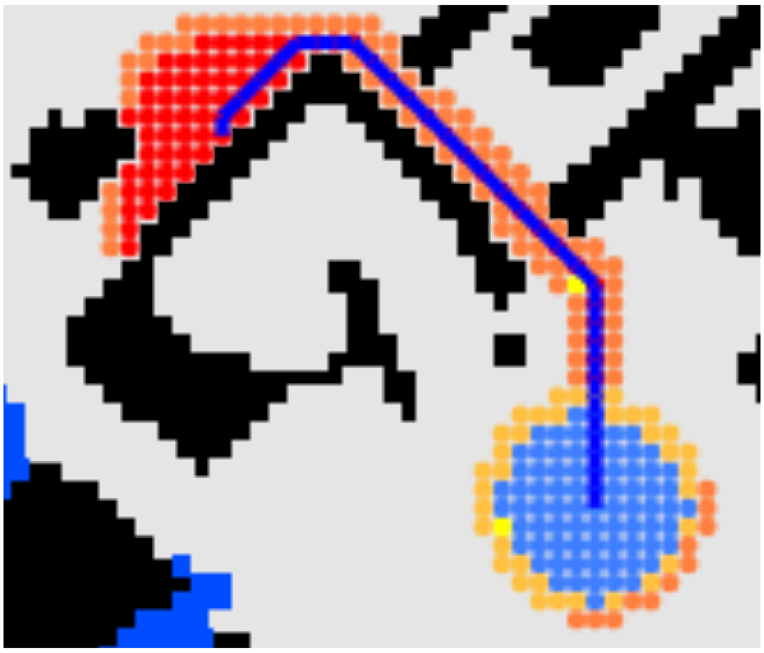


Figure 5.2: Expanded nodes by NBS on an asymmetrical board. Closed forward nodes are blue, closed backward nodes are red. Created with the NBS example application from [7].

In algorithm 3 λ_F and λ_B are empty paths starting from the start s and the goal g respectively. With every iteration of the loop, a pair of one forward and one backward path is selected. Each path is expanded, and the new paths are added to the open set $Open_F$ for forward paths and $Open_B$ for backward paths. If two paths in the open set lead to the same node, the longer path is discarded. Once a forward and a backward path meet, they form a connection from s to g. The length of this combined path is stored in C, and will be lowered for each shorter path that is found. The algorithm continues until there are no more paths in the open sets that could possibly be shorter. This is determined by the smallest heuristic value of all paths in the open set.

Algorithm 3: NBS, based on [6]

```
Input  : start node $s$, goal node $g$
Output: $p$ = $(s,...,g)$, an optimal path from $s$ to $g$,
        or $nopath$ if no path from $s$ to $g$ exists.

// Shortest known path
$P \leftarrow nopath$
// Cost of P
$C \leftarrow \infty$
// Paths that will be expanded
$Open_F \leftarrow \{\lambda_F\}$
$Open_B \leftarrow \{\lambda_B\}$
// Paths that have already been expanded
$Closed_F \leftarrow \{\}$
$Closed_B \leftarrow \{\}$

while $Open_F \neq \varnothing$ and $Open_B \neq \varnothing$ do
    Pairs $\leftarrow Open_F \times Open_B$

    // Lowest bound of all pairs
    lbmin $\leftarrow min\{lb(X,Y) \mid (X,Y) \in Pairs\}$
    // If $lbmin$ is bigger or equal to the cost of the shortest known
       path, that path is already optimal.
    if $lbmin \geq C$ then
        return P
    end

    // Choose pair with minimal $lb(X,Y)$ and minimal cost of the
       individual paths
    minset $\leftarrow \{(X,Y) \in Pairs \mid lb(X,Y) = lbmin\}$
    Uset $\leftarrow \{X \mid \exists Y (X,Y) \in minset\}$
    Umin $\leftarrow min\{c(X) \mid X \in Uset\}$
    Choose any $U \in Uset$ such that $c(U) = Umin$
    Vset $\leftarrow \{Y \mid (U,Y) \in minset\}$
    Vmin $\leftarrow min\{c(Y) \mid Y \in Vset\}$
    Choose any $V \in Vset$ such that $c(V) = Vmin$

    // Expand paths of that pair
    Forward-Expand(U)
    Backward-Expand(V)
end
return P
```

The *Forward-Expand* function adds new paths to the forward open list $Open_F$. In the previous steps of the algorithm, the input path U has been selected to be expanded. The function $expand_F(U)$ creates new paths that add one extra step to the end of U. U is then moved to the closed list $Closed_F$ so that it will never be expanded again.

For every new path W, the opposite open list is checked to see if the end of W and the end of opposing path Y meet. If that is the case a connection from the start to the goal has been found, since W is a forward path that starts from s and Y is a backwards path that starts from g. If this combined path is shorter than the current best distance C, C is updated to the new shortest path length.

W is now added to $Open_F$, unless there already is a known path X that leads from s to the same node as W, and that path X is shorter.

Algorithm 4: NBS - **Function:** Forward-Expand, based on [6]

```
Input: Path to be expanded $U$

Move $U$ from $Open_F$ to $Closed_F$
foreach $W \in expand_F(U)$ do
    // Update best path if $W$ connects to a backwards path and is
       shorter.
    if $\exists Y \in Open_B$ with $end(Y) = end(W)$ then
        if $c(W) + c(Y) < C$ then
            $P = W \cup Y$
            $C = c(W) + c(Y) = c(P)$
        end
    end

    // Add $W$ to $Open_F$, unless a shorter path to the same point
       already exists.
    if $\exists X \in Open_F \cup Closed_F$ with $end(X) = end(W)$ then
        if $c(W) < c(X)$ then
            remove $X$ from $Open_F$ or $Closed_F$
            Add $W$ to $Open_F$
        end
    else
        Add $W$ to $Open_F$
    end
end
```

The implementation of *Backwards-Expand* is analogous to *Forward-Expand.*

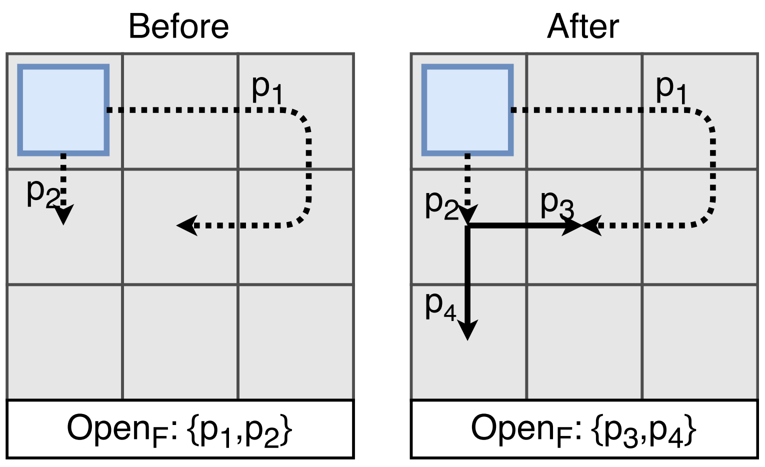


Figure 5.3: Example for the behavior of Forward-Expand. In this situation, p_2 was selected for expansion and is removed from the open list. The children of p_2 are p_3 and p_4. p_4 leads to a new node and is added to the open list. p_3 leads to the same node as p_1, which the open list already contains. The comparison $c(p_1) > c(p_3)$ decides that p_1 gets removed and p_3 gets added to the open list.

5.3 Algorithm comparisons

In Figure 5.4 the steps of the A* and NBS algorithms are compared. One can see how the A* algorithm has to "backtrack" when the heuristic leads it into a dead end. In this case the NBS algorithm is expanding more equally. It still reaches the dead ends, however they are prioritized less due to the additional knowledge gained through the bidirectional search. The forward search eventually depletes all the paths with small estimates. While the backwards search is likely to have such possibilities as well, the lower bound *lb* contains the maximum of both. As the forward estimate is already high, the backwards search explores paths with low cost instead. This can be beneficial in many cases, however for this puzzle it does worse than both A* and IDA*.

For the puzzle in Figure 5.5, one can see how the IDA* algorithm does significantly worse than both other algorithms. It runs into it's depth bound often and therefore has to evaluate a lot more nodes. A* runs into fewer dead ends when compared to the puzzle in Figure 5.4 and reaches the goal quickly.

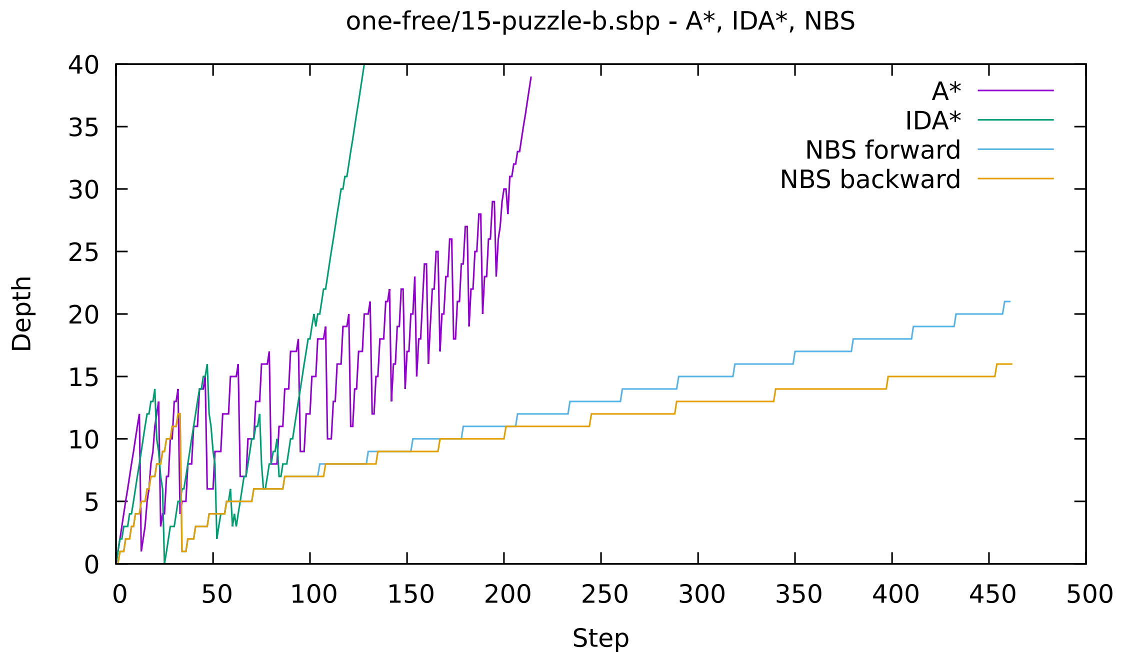


Figure 5.4: Depth for every step of the A*, IDA* and NBS algorithms. The puzzle is displayed in Figure 5.6.

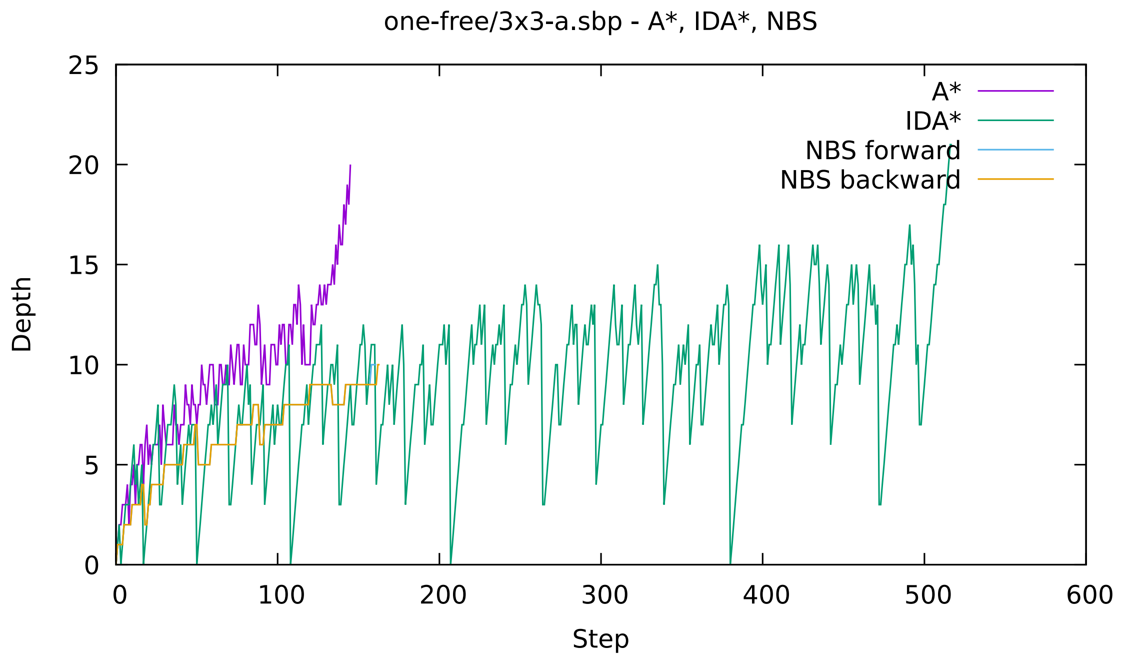


Figure 5.5: Depth for every step of the A*, IDA* and NBS algorithms. The puzzle is displayed in Figure 5.7.

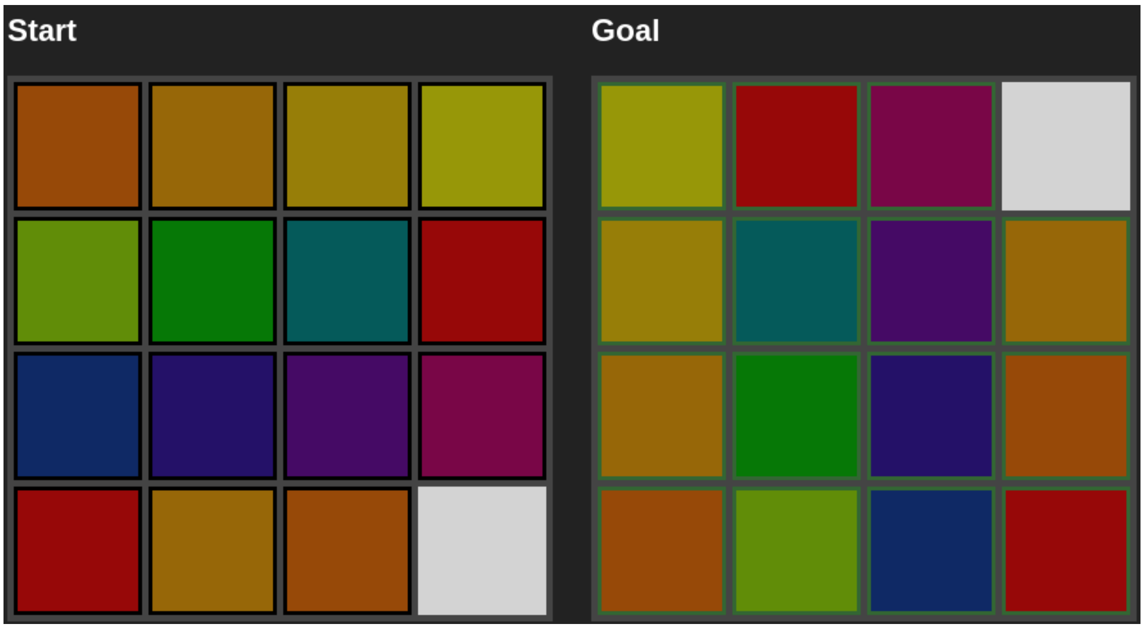


Figure 5.6: Visualization of the one-free/15-puzzle-b puzzle.

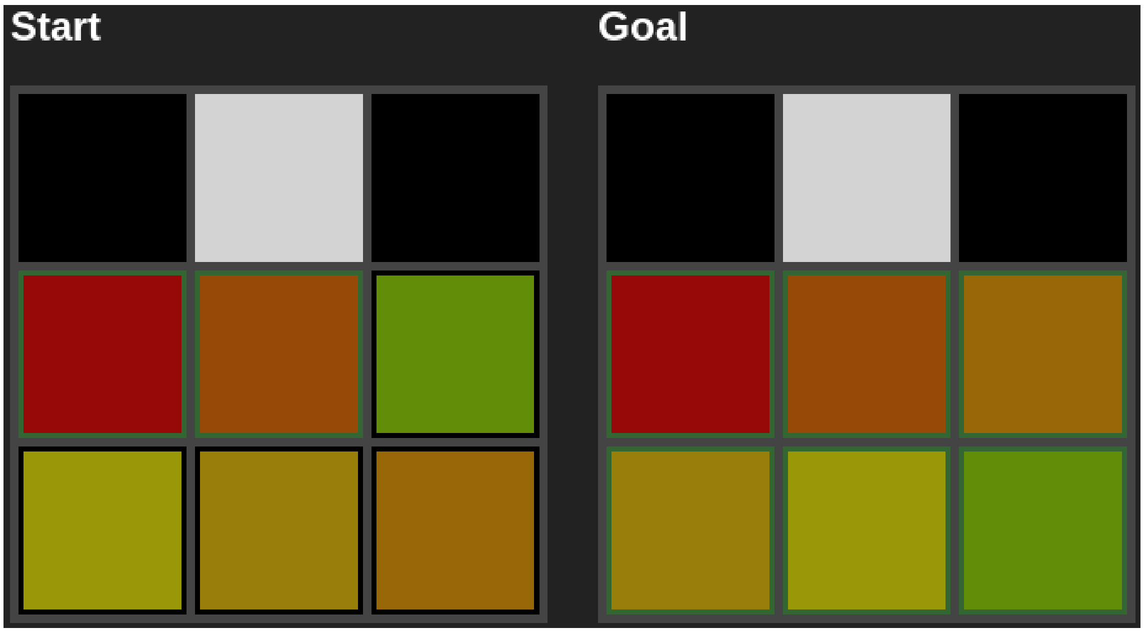


Figure 5.7: Visualization of the one-free/3x3-a puzzle.

6 Implementation

For this thesis, the A*, IDA*, and NBS algorithms were implemented. Additionally, a website and a command-line tool are provided for displaying and solving puzzles. The algorithms were implemented using the *Rust* programming language, and compiled as a library ("crate") so they can be integrated into multiple projects. The command-line interface was implemented separately, using *Rust* as well.

6.1 Command-line Tool

The provided command-line binaries can be run on most Windows and Linux operating systems, without any additional dependencies. They can solve sliding block puzzles that are provided in the ".sbp" file format, which can be created by tools such as the *SBP Solver* program by Pierre-Francois Culand. Extended logging is available and has been used to create the evaluation depth graphs used in this thesis.

The tool can be configured with the following command-line options:

`--solver` *`algorithm_name`*	Chooses the algorithm for solving the puzzle. Options for *algorithm_name*: *AStar*, *IDA*, *NBS*. If the solver option is not specified, the A* algorithm is used by default.
`--duration`	Prints out the time that the program took to solve the puzzle.
`--evaluation`	Option *Map* prints out the number of nodes that were expanded. For each depth value, the number of nodes expanded is printed out in one line. For the NBS algorithm, forward and backward expansions are printed separately. Option *List* prints out the current depth of each iteration of the algorithm. Also prints two numbers when NBS is selected.
`--moves`	Prints out the length of the shortest path that was found.
`--result`	Prints out the steps of the path that was found.

Example usage:

```
./console --result --solver AStar puzzles/sbp-solver/Trivial.sbp
```

The *evaluation* option has been used to create the depth graphs in Figure 4.1 and Figure 5.4.

6.2 Locating performance problems

While this thesis mostly focuses on the number of node expansions as a performance metric of an algorithm, the implementation had to have a minimum level of optimization for the algorithm to terminate after a reasonable time. A variety of techniques have been used to increase performance of the program. However, finding the slow sections of the code in the first place is an important task. Otherwise the actions taken to speed up the program might only have little effect.

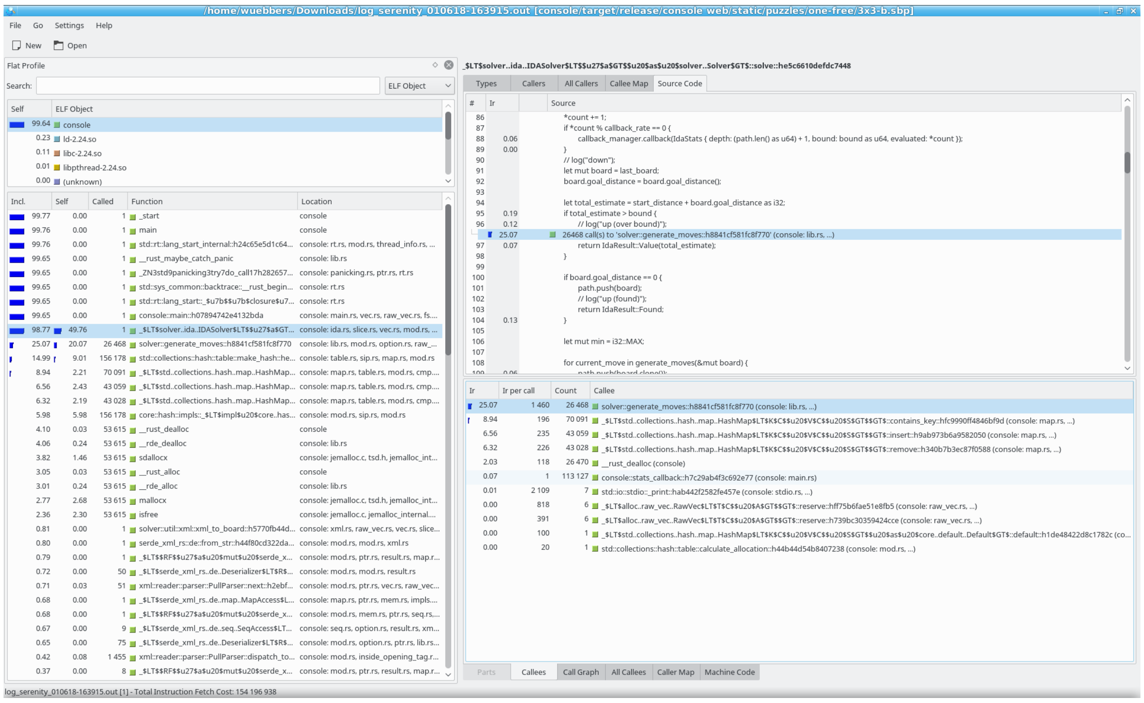
The main tool used for this is the *callgrind* option included in the Valgrind tool for the Linux operating system. In combination with the visualization tool KCachegrind, as seen in Figure 6.1, the functions that take up a lot of runtime can be traced.

6.3 Transposition Tables

Since the graph of possible moves is cyclic, none of the algorithms described in this thesis allow their search paths to loop. To avoid searching the same node twice, transposition tables are used to store which nodes have already been visited. Since storing the entire board position leads to high memory usage when we execute the algorithms, a hash of the position is stored instead.

The hash value of any board is calculated by taking the pieces of the board and combining their hashes. The piece hashes are made up by their position as well as their type. The type of a piece is a unique number, if the piece has a goal position. If the piece has no goal position, it shares a type with all other goal-less pieces that have the same width and height. This results in a hash value that changes when any piece is moved. However, if two pieces that have the same size and no goal position are swapped, the hash value stays the same.

A set of these hashes, the transposition table, is kept in a *HashSet*. The Rust *HashSet* and *HashMap* structs have an expected time complexity of $O(1)$ for getting an element (see [8]), which means the transposition table can be quickly checked to see if any board position has been visited before.



/home/wuebbers/Downloads/log_serenity_010618-163915.out [console/target/release/console web/static/puzzles/one-free/3x3-b.sbp]

Self	ELF Object
99.64	console
0.23	ld-2.24.so
0.11	libc-2.24.so
0.01	libpthread-2.24.so
0.00	(unknown)

Incl.	Self	Called	Function	Location
99.77	0.00	1	_start	console
99.76	0.00	1	main	console
99.76	0.00	1	std::rt::lang_start_internal::h24c65e5d1c64...	console: rt.rs, mod.rs, thread_info.rs, ...
99.65	0.00	1	__rust_maybe_catch_panic	console: lib.rs
99.65	0.00	1	_ZN3std9panicking3try7do_call17h282657...	console: panicking.rs, ptr.rs, rt.rs
99.65	0.00	1	std::sys_common::backtrace::__rust_begin...	console: rt.rs
99.65	0.00	1	std::rt::lang_start::_$u7b$$u7b$closure$u7...	console: rt.rs
99.65	0.00	1	console::main::h07894742e4132bda	console: main.rs, vec.rs, raw_vec.rs, fs....
98.77	49.76	1	_LTsolver..ida..IDASolver$LT$$u27aGT...	console: ida.rs, slice.rs, vec.rs, mod.rs, ...
25.07	20.07	26 468	solver::generate_moves::h8841cf581fc8f770	console: lib.rs, mod.rs, option.rs, raw_...
14.99	9.01	156 178	std::collections::hash::table::make_hash::he...	console: table.rs, sip.rs, map.rs, mod.rs
8.94	2.21	70 091	_LTstd..collections..hash..map..HashMap...	console: map.rs, table.rs, mod.rs, cmp....
6.56	2.43	43 059	_LTstd..collections..hash..map..HashMap...	console: map.rs, table.rs, mod.rs, cmp....
6.32	2.19	43 028	_LTstd..collections..hash..map..HashMap...	console: map.rs, table.rs, mod.rs, cmp....
5.98	5.98	156 178	core::hash::impls::_$LT$impl$u20$core..has...	console: mod.rs, sip.rs, mod.rs
4.10	0.03	53 615	__rust_dealloc	console
4.06	0.24	53 615	__rde_dealloc	console: lib.rs
3.82	1.46	53 615	sdallocx	console: jemalloc.c, tsd.h, jemalloc_int...
3.05	0.03	53 615	__rust_alloc	console
3.01	0.24	53 615	__rde_alloc	console: lib.rs
2.77	2.68	53 615	mallocx	console: jemalloc.c, tsd.h, jemalloc_int...
2.36	2.30	53 615	isfree	console: jemalloc.c, jemalloc_internal....
0.81	0.00	1	solver::util::xml::xml_to_board::h5770fb44d...	console: xml.rs, raw_vec.rs, vec.rs, slice...
0.80	0.00	1	serde_xml_rs::de::from_str::h44f80cd322da...	console: mod.rs, mod.rs, xml.rs
0.79	0.00	1	_$LT$$RF$$u27au20mutu20$serde_x...	console: mod.rs, ptr.rs, result.rs, map.r...
0.72	0.00	50	_LTserde_xml_rs..de..DeserializerLTR$...	console: mod.rs, mod.rs, result.rs
0.71	0.03	51	xml::reader::parser::PullParser::next::h2ebf...	console: mod.rs, ptr.rs, vec.rs, raw_vec...
0.68	0.00	1	_LTserde_xml_rs..de..map..MapAccess$L...	console: map.rs, ptr.rs, mem.rs, impls....
0.68	0.00	1	_$LT$$RF$$u27au20mutu20$serde_x...	console: mod.rs, mem.rs, ptr.rs, seq.rs,...
0.67	0.00	9	_LTserde_xml_rs..de..seq..SeqAccess$LT...	console: seq.rs, option.rs, result.rs, xm...
0.65	0.00	75	_LTserde_xml_rs..de..DeserializerLTR$...	console: mod.rs, option.rs, ptr.rs, lib.rs...
0.42	0.08	1 455	xml::reader::parser::PullParser::dispatch_to...	console: mod.rs, inside_opening_tag.r...
0.37	0.00	8	_$LT$$RF$$u27au20mutu20$serde_x...	console: mod.rs, ptr.rs, result.rs, map.r...

log_serenity_010618-163915.out [1] - Total Instruction Fetch Cost: 154 196 938

_LTsolver..ida..IDASolver$LT$$u27aGT$$u20asu20$solver..Solver$GT$::solve::he5c6610defdc7448

```
Ir      Source
              *count += 1;
              if *count % callback_rate == 0 {
0.06              callback_manager.callback(IdaStats { depth: (path.len() as u64) + 1, bound: bound as u64, evaluated: *count });
0.00          }
              // log("down");
              let mut board = last_board;
              board.goal_distance = board.goal_distance();

              let total_estimate = start_distance + board.goal_distance as i32;
0.19          if total_estimate > bound {
0.12              // log("up (over bound)");
    25.07     26468 call(s) to 'solver::generate_moves::h8841cf581fc8f770' (console: lib.rs, ...)
0.07              return IdaResult::Value(total_estimate);
              }

              if board.goal_distance == 0 {
                  path.push(board);
                  // log("up (found)");
                  return IdaResult::Found;
0.13          }

              let mut min = i32::MAX;

              for current_move in generate_moves(&mut board) {
```

Ir	Ir per call	Count	Callee
25.07	1 460	26 468	solver::generate_moves::h8841cf581fc8f770 (console: lib.rs, ...)
8.94	196	70 091	_LTstd..collections..hash..map..HashMapLTK$C$$u20VC$$u20SGT$$GT$::contains_key::hfc9990ff4846bf9d (console: map.rs, ...)
6.56	235	43 059	_LTstd..collections..hash..map..HashMapLTK$C$$u20VC$$u20SGT$$GT$::insert::h9ab973b6a9582050 (console: map.rs, ...)
6.32	226	43 028	_LTstd..collections..hash..map..HashMapLTK$C$$u20VC$$u20SGT$$GT$::remove::h340b7b3ec87f0588 (console: map.rs, ...)
2.03	118	26 470	__rust_dealloc (console)
0.07	1	113 127	console::stats_callback::h7c29ab4f3c692e77 (console: main.rs)
0.01	2 109	7	std::io::stdio::_print::hab442f2582fe457e (console: stdio.rs, ...)
0.00	818	6	_LTalloc..raw_vec..RawVecLTT$C$$u20AGT$$GT$::reserve::hff75b6fae51e8fb5 (console: raw_vec.rs, ...)
0.00	391	6	_LTalloc..raw_vec..RawVecLTT$C$$u20AGT$$GT$::reserve::h739bc30359424cce (console: raw_vec.rs, ...)
0.00	100	1	_LTstd..collections..hash..map..HashMapLTK$C$$u20VC$$u20SGT$$u20asu20$core..default..Default$GT$::default::h1de48422d8c1782c (co...
0.00	20	1	std::collections::hash::table::calculate_allocation::h44b44d54b8407238 (console: mod.rs, ...)

Figure 6.1: KCachegrind application window

6.4 Efficient Computation of Possible Moves

When implementing the different path finding algorithms, a universal source of slowdown became apparent. For any of the algorithms a lot of nodes have to be expanded, meaning that all possible moves of a given board position have to be calculated.

The original implementation used a simple list of pieces. Every piece was moved in every possible direction and was checked for intersections with other pieces or walls. If no intersections were found, the move was marked as valid. This method of comparing list elements with each other turned out to be a major factor in the runtime of the program. The assumed reason for this slowdown is that every piece has to be checked for intersection with every other piece, for every possible move direction. This does not seem to scale well for puzzles with many pieces.

To solve this problem, the *bitboard* technique as described in [9] was implemented. While a list of pieces is still required, a representation of blocked fields of the board is created as well. This is done by mapping every bit of a unsigned 64-bit number to a position of the board. If a piece occupies a certain position the bit is set to one, otherwise it stays at zero.

If we want to check if a certain piece can move to any position, we only have to remove the bits of the piece we want to move from the board. This is necessary since pieces can occupy many spaces, and could potentially intersect with their old position after a move. When all target positions are zero, the move is valid. This reduces the amount of checks that have to be done from comparing every piece with every other piece to just checking a few bits. After implementing this technique, the overall amount of time for calculating moves was reduced to an insignificant amount.

6.5 Implementing the Linear Conflict Heuristic

While the concept of the *Linear Conflict* heuristic is relatively simple, the actual implementation details prove difficult to get right. When there are multiple conflicts in the same row, it is simply not possible to sum up the conflicts, since they may interlock and could be solved in a faster way. Failing to fix this would result in a heuristic that would not be admissible. The solution used for solving this first lists all the pairs for every row and column. For each of these rows or columns, the piece with the biggest number of pairs containing it is selected. All those pairs are removed and two moves are added to the heuristic value. This happens because we assume the piece will be moved out of the way in the solution, incurring an extra cost of two and solving all conflicts regarding this piece. Then the next piece is selected, repeating the process until all conflicts in this row or column are solved.

6.6 Priority queue for A*

The A* algorithm requires the selection of a node with the smallest goal distance out of all nodes in the pending list. An easy way to implement this is to use a priority queue for storing the pending list, which will sort the nodes by their goal distance. To enforce a consistent behavior, the queue will sort entries by the first-in-first-out principle if their goal distances are equal. This is not required for the correctness of the algorithm, but makes the algorithm steps independent of the actual implementation of the queue.

7 Algorithm timings

Timings have been generated using the command-line tool, solving each puzzle sequentially with each of the three algorithms. For puzzles where the NBS algorithm is not guaranteed to find optimal paths, the entries are marked with the word *skip* in the table. The timings were measured on a computer with a Intel® Core^{TM} i7-4790 CPU with a 3.6 GHz clock rate. The application runs single-threaded.

The *length* column shows the lowest number of moves to solve the puzzle. Since all used algorithms find optimal paths, their length will always be the same. The actual moves of the path might differ between algorithms.

The *time* columns show the measured time it took for each algorithm to find the optimal path. The timing starts after the puzzle has been loaded, right before the first step of the selected algorithm. It ends when the algorithm has finished, but before the results are printed out to the console. When the time to solve the puzzle exceeds thirty seconds, the solver is terminated and marked by the entry "-" in the table.

The *expanded* columns are displaying the total number of nodes that have been visited and expanded by the selected algorithm.

Our puzzle selection is sorted by different categories. The first category *one-free* contains different variants of the *15-puzzle* on a four by four grid, as well as similar puzzles in smaller sizes. All blocks on the grid are spanning only a single field, and there is only one free space on the board. The *gauntlet* category has only a single free space as well. However, the blocks vary in size and the solution paths are considerably longer than the previous puzzles. The last category *sbp-solver* has more varied puzzles, containing the *Century* puzzle by John Conway and some levels from the *Rush Hour* game.

puzzle	length	time			expansions		
		A*	IDA*	NBS	A*	IDA*	NBS
one-free/15-puzzle-a	44	5s, 78ms	5s, 17ms	-	546952	2107984	-
one-free/15-puzzle-b	39	1ms	<1ms	7ms	214	127	462
one-free/15-puzzle-c	-	-	-	-	-	-	-
one-free/15-puzzle-d	-	-	27s, 969ms	-	-	12522519	-
one-free/3x3-a	20	<1ms	<1ms	<1ms	145	515	162
one-free/3x3-b	30	30ms	25ms	215ms	10549	23665	6202
one-free/3x3-c	23	1ms	2ms	5ms	511	1887	712
one-free/4x3-a	-	-	-	-	-	-	-
one-free/4x3-b	30	48ms	32ms	754ms	10149	20814	8123
gauntlet/Gauntlet	243	2ms	325ms	skip	1137	214147	skip
gauntlet/Gauntlet1	235	4ms	1s, 223ms	skip	2240	905564	skip
gauntlet/Gauntlet2	311	1ms	1s, 494ms	skip	945	1012689	skip
gauntlet/Gauntlet484	484	18ms	-	skip	7398	-	skip
gauntlet/Gauntlet5	415	13ms	7s, 274ms	skip	5008	3706943	skip
gauntlet/Gauntlet6	183	1ms	99ms	skip	766	67628	skip
sbp-solver/BlockAdo	103	9ms	-	skip	4444	-	skip
sbp-solver/BlockAdo+	152	12s, 338ms	-	-	1919344	-	-
sbp-solver/Century	131	83ms	-	skip	41700	-	skip
sbp-solver/Donkey	116	52ms	-	skip	23912	-	skip
sbp-solver/RushHour-39	82	9ms	-	skip	3781	-	skip
sbp-solver/RushHour-40	81	8ms	-	skip	2962	-	skip

7.1 Analyzing the timings

When looking at the timings for the A* and IDA* algorithms in the *one-free* category, A* is slightly slower than IDA*. In most cases the A* algorithm only expands half as many boards. Due to the simplicity of the IDA* algorithm, it seems to expand boards at a slightly quicker pace. Compared to the other two algorithms, then NBS algorithm runs significantly slower. In most cases it expands more boards than A* and less boards than IDA*.

In the *gauntlet* category, A* significantly outperforms IDA* in both time and expansion count. In most cases, IDA* expands about one hundred times more nodes than A*. This seems to happen due to the long solution paths of the puzzles. The NBS algorithm is not applicable in this set of puzzles, since they do not have defined goal positions.

The *sbp-solver* puzzles were only able to be solved by the A* algorithm. Comparing the node expansions to the previous category, the puzzles seem to be slightly harder. This seems to be fatal for the IDA* algorithm's runtime, as it exceeds the thirty second limit for every puzzle. In the one puzzle where the NBS algorithm was applicable, it exceeded this limit as well.

When looking at the overall results, the A* algorithm seems to be the best choice for solving sliding block puzzles. However, the thirty second time limit has been chosen for a reason. After experimenting with the runtime of the program, the approximately 13 GB of free RAM were completely filled up. If the algorithm ran for around 35 seconds, it was likely to crash the application with an out-of-memory error. Of course, runtime isn't a good measurement of how much memory the A* algorithm consumes. Most of the memory is consumed by the open list, which can grow at different speeds depending on the shape of the puzzle.

8 Conclusion

In this thesis three different algorithms for solving sliding block puzzles were described. These algorithms were originally designed for general graph search and have been adapted to fit the specific requirements.

Out of all the algorithms, the A* algorithm has shown to be the most efficient algorithm for many of the tested puzzles. Its runtime was consistently faster or at least similar to the other algorithms. The downside of using this algorithm is the increased memory consumption. While this was not explicitly measured in this thesis, it often lead to problems during testing.

The IDA* algorithm seems best suited for the puzzles from the *one-free* category, where it outperformed A* by a small margin. The performance difference is mostly due to the faster speed at which IDA* expands the nodes. This is likely the result of better optimized code when compared to the implementation of the other algorithms.

Since the NBS algorithm is bidirectional, it is not applicable to many puzzles. On the puzzles that it can be applied on, the node expansion count is in a similar range as the count of the A* algorithm, while occasionally outperforming it. However, the implementation was the most complex by far. This resulted in slower execution times, as optimizing this code was out of scope for this thesis.

8.1 Future work

While this thesis has compared the overall speed of solving puzzles for each algorithm, an unequal amount of time has been spent on implementing and optimizing the code for each algorithm. For a more fair comparison, the amount of node expansions have been examined. Future work could examine the actual performance of certain implementations. Additionally, the memory usage of the algorithms could be measured as well.

A Instructions for accompanying CD

The CD provided with this thesis contains the source code for the three algorithm implementations, as well as a website for interactive play. For compiling the code, the Rust compiler in version 1.32.0 or higher is required. It can be acquired through the Rustup installer program at `https://rustup.rs/`.

The *solver* project folder contains the actual algorithms that were implemented. The code can be found in the `a_star.rs`, `ida.rs` and `nbs.rs` files in the `solver/src` directory.

The *console* project contains code for the command-line application. In the `console` directory, the `cargo build` command can be run to compile the code. The compiled application will be located at `target/debug/console` or `target/debug/console.exe` depending on your operating system.

The *web* project contains the code for the website. It also contains the collection of puzzles, which is located at `web/static/puzzles`. For running the website, the *cargo-web* subcommand in version 0.6.23 or higher is required, which can be installed with `cargo install cargo-web`. In the `web` directory, the command `cargo web start` will run a local website at `http://localhost:8000`. To build the website code for hosting, the command `cargo web deploy` can be used.

Bibliography

[1] S. J. Russell and P. Norvig, Artificial intelligence: a modern approach, pp. 94–95, Malaysia; Pearson Education Limited, 2010.

[2] O. Hansson, A. Mayer, and M. Yung, Criticizing solutions to relaxed models yields powerful admissible heuristics, *Information Sciences*, vol. 63, no. 3, pp. 207–227, 1992.

[3] R. E. Korf, Depth-first iterative-deepening: An optimal admissible tree search, *Artificial Intelligence*, vol. 27, no. 1, pp. 97–109, 1985.

[4] R. Tarjan, Depth-first search and linear graph algorithms, *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, 1972.

[5] A. Mahanti, S. Ghosh, D. S. Nau, A. K. Pal, and L. N. Kanal, Performance of ida on trees and graphs, in *AAAI*, pp. 539–544, 1992.

[6] J. Chen, R. C. Holte, S. Zilles, and N. R. Sturtevant, Front-to-end bidirectional heuristic search with near-optimal node expansions, *arXiv preprint arXiv:1703.03868 [cs.AI]*, 2017.

[7] Nathan R. Sturtevant, Bidirectional Search - GDC 2018 AI Summit. `https://movingai.com/GDC18/index.html`, 2018. [Online; accessed 30-January-2019].

[8] The Rust Project Developers, Rust Documentation, Module std::collections. `https://doc.rust-lang.org/1.32.0/std/collections/`, 2013. [Online; accessed 16-December-2018].

[9] R. Hyatt, Chess program board representations. `https://web.archive.org/web/20140205165421/http://www.cis.uab.edu:80/info/faculty/hyatt/boardrep.html`, 2014. [Online; accessed 21-January-2019].

Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass ich diese Bachelorarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Passau, Februar 2019

Till Wübbers