Solving Sokoban efficiently: Search tree pruning techniques and other enhancements

bachelor thesis

Niklas Sandhu Peters
409947

20.06.2023

Supervisor: Prof. Dr. Benjamin Blankertz
Prof. Dr. Marc Alexa

Technische Universität Berlin
School of Electrical Engineering and Computer Science
Institute of Software Engineering and Theoretical Computer Science
Special Field Neurotechnology

Abstract

Artificial Intelligence is present in many fields nowadays. The game Sokoban offers the opportunity to test techniques and algorithms that are also used in methods of Artificial Intelligence and game theory.

In this thesis we present different graph search algorithms, as well as different pruning techniques and specific improvements to solve levels in the game Sokoban faster and more efficient. For this we provide a Sokoban solver that can be configured and started via a Graphical-User-Interface (GUI) as well as over a Command-Line-Interface (CLI).

We will show with which improvements we get a clear, visible improvement to solve Sokoban levels more efficiently. In addition, the techniques shown can be applied to other fields of Artificial Intelligence and game theory.

Zusammenfassung

Künstliche Intelligenz ist heutzutage in vielen Bereichen präsent. Das Computerspiel Sokoban bietet eine Plattform, um Techniken und Algorithmen zu testen, die auch in Methoden der künstlichen Intelligenz und Spieltheorie angewandt werden.

In dieser Bachelorarbeit stellen wir verschiedene Graph-Suchalgorithmen, sowie verschiedene Pruning-Techniken und spezifische Verbesserungen vor, um Level im Spiel Sokoban schneller und effizienter zu lösen. Dazu stellen wir einen Sokoban-Solver zur Verfügung, der sowohl über eine grafische Benutzeroberfläche (GUI) verfügt, als auch über ein Command-Line-Interface (CLI) konfiguriert und gestartet werden kann.

Wir werden zeigen, mit welchen Methoden wir sichtbare Verbesserung erreichen, um Sokoban-Level effizienter lösen zu können. Darüber hinaus können die gezeigten Techniken auf andere Bereiche der künstlichen Intelligenz und Spieltheorie übertragen werden.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 About our work . . . 1
1.3 Structure of the thesis . . . 1

2 Preliminary concepts **2**
2.1 Artificial Intelligence and Game Theory . . . 2
2.2 Graphs . . . 2
2.3 Search Algorithms . . . 3
2.3.1 Breadth First Search (BFS) . . . 3
2.3.2 Dijkstra's search algorithm . . . 5
2.3.3 A* search algorithm . . . 8
2.3.4 IDA* search algorithm . . . 9
2.3.5 Heuristics for search algorithms . . . 12
2.3.6 Bounded Relaxation . . . 12
2.4 Transposition Tables . . . 13
2.5 Sokoban . . . 14
2.5.1 Rules of Sokoban . . . 14
2.5.2 Why solving Sokoban levels is interesting . . . 17

3 Related Work **21**
3.1 Complexity . . . 21
3.2 Rolling Stone Solver . . . 21
3.3 Festival Solver . . . 22
3.4 Pre-calculation approach . . . 23

4 Implementation - Conception and Realization **24**
4.1 Python . . . 24
4.1.1 Used Python modules . . . 24
4.2 Solver Development . . . 25
4.2.1 General program sequence . . . 25
4.2.2 Search Spaces . . . 26
4.2.3 Move Generators . . . 29
4.2.4 Board Representation . . . 30
4.2.5 Transposition Table . . . 30
4.2.6 Search Algorithms . . . 31
4.2.7 Heuristics . . . 31
4.2.8 Static and Dynamic Weighted A* . . . 32

4.2.9 Deadlock Detection . . . 32
4.2.9.1 Simple Deadlocks . . . 32
4.2.9.2 Freeze Deadlocks . . . 33
4.3 Solver Optimization Level Configurations . . . 33
4.4 Graphical User Interface and Game Mode . . . 34

5 Experimental results 38
5.1 Test Environment . . . 38
5.2 Test Set . . . 38
5.3 Comparing search algorithms . . . 39
5.4 Relaxing the move space . . . 40
5.5 Speedup with a Transposition Table . . . 41
5.6 Efficient solving with domain knowledge enhancements . . . 42
5.6.1 Simple Deadlocks . . . 42
5.6.2 Freeze Deadlocks . . . 43
5.7 Experiments with bound relax . . . 44
5.8 Comparison to existing solvers . . . 47

6 Discussion 49
6.1 Search algorithm comparison . . . 49
6.2 Move space relaxation . . . 49
6.3 Transposition Table speedup . . . 50
6.4 Deadlock detection . . . 50
6.5 Bound relax . . . 50
6.6 Results in comparison to existing work . . . 51

7 Conclusion 52
7.1 Constraints and limitations of our work . . . 52
7.2 Future improvements and application areas . . . 52

Bibliography 54

List of Figures

2.1 BFS walk through a graph 5
2.2 Dijkstra's algorithm search shortest distances to all nodes in a graph 7
2.3 A* searches for a goal node 9
2.4 Transposition Table used to reuse computed node values 14
2.5 Objects in a Sokoban level 15
2.6 The Sokoban $n \times m$ level grid 16
2.7 Sokoban level in ASCII notation 16
2.8 Solution example of the 5th level from the *Microban I* level library 17
2.9 Simple Deadlocks in a Sokoban level 18
2.10 Freeze Deadlock in a Sokoban level 19
2.11 Recursive Freeze Deadlock in a Sokoban level 19
2.12 Deadlock due to a frozen box 20

3.1 The FESS algorithm 22
3.2 Connectivity Feature example 23

4.1 The general program sequence to solve a level 26
4.2 Move Space Graph Example 27
4.3 State Space Graph Example 28
4.4 The Manhattan Distance 31
4.5 GUI Overview 34
4.6 The GUI solves a level 35
4.7 GUI Level has been solved 35
4.8 GUI solver configuration 36
4.9 Game Mode 37

5.1 L0 Solver tested on the easy level set 39
5.2 L0 Solver tested on the hard level set 39
5.3 Results from L1 Solver with Extended Move Generator 40
5.4 Solved Levels of L2 Solver 41
5.5 Average processing speed of L2 Solver 42
5.6 Results from L3 Solver with Simple Deadlock Detection 42
5.7 Results from L4 Solver with Freeze Deadlock Detection 43
5.8 Processing speed comparison of Solver L3 and Solver L4 44
5.9 Results from L5 Static Solver with different ε values 45
5.10 Results from L5 Dynamic Solver with different ε values 46
5.11 Comparison between Solvers L4 and L5 Dynamic 10.0 47
5.12 Comparison of a *L5* solver variant with other solvers 48

List of Tables

4.1 Possible solver configurations 34

List of Algorithms

1 Breadth-First-Search (BFS) 4
2 Dijkstra's algorithm with priority queue 6
3 A* algorithm 10
4 IDA* in a recursive variant 11

5 Classic Move Generator 29

1 Introduction

1.1 Motivation

Sokoban is an easily comprehensible game with simple rules and has a steady fan base that is constantly creating new levels for Sokoban. Nevertheless, even easy Sokoban levels require a lot of computational effort to solve.

Therefore the single-player game Sokoban provides a good platform to study graph search algorithms and use methods to prune the search and make it more efficient.

Sokoban has also been proven to be NP-Hard [1] and even PSPACE-complete [2] which underlines the relevance of Sokoban for the scientific use and the need of steady improvements of graph search algorithms.

1.2 About our work

Since solving Sokoban requires a high computational effort, it seems reasonable to make this needed computational effort more efficient.

In our work we present pruning techniques and other improvements that make solving Sokoban levels more efficient. We also offer a GUI to provide a playground for our solver to easily test the improvements.
Furthermore, we will measure the improvements of the solver and show the respective benefit of these improvements in relation to a self-created Sokoban test set.

1.3 Structure of the thesis

We start in chapter 2 by introducing the basic terms and methods to have a basic prior knowledge and understanding, necessary for this thesis. Then, in chapter 3, we will present existing work on the topic of Sokoban and its efficient solving.

In chapter 4 we will introduce our solver and its GUI and describe how we planned and implemented the program architecture. Then, in chapter 5 and chapter 6, we evaluate the experimental results and effects of the improvements of our solver and briefly compare our solver with existing solvers.

Our work ends with chapter 7 where we will draw a conclusion and give an outlook for future application purposes and possible improvements.

2 Preliminary concepts

2.1 Artificial Intelligence and Game Theory

Artificial Intelligence (AI) plays an important role in the context of game theory. The field aims to develop strategies and algorithms that can make intelligent decisions in games whether they are competitive or cooperative scenarios. The game Sokoban is one of the games that is intensively used for research on game theory and Artificial Intelligence.

Artificial Intelligence techniques have been applied to the Sokoban game to create and develop intelligent search agents, which are capable of solving Sokoban levels more efficiently. One of the standard techniques is to use search algorithms, such as A* to explore the search space of the game and find optimal and semi-optimal solutions.

Due to the hard problems of Sokoban, the game also serves as a test ground for developing and improving advanced AI techniques such as heuristic search algorithms and machine learning approaches. Other researchers have explored domain-specific heuristics to guide search algorithms towards a more promising path in the search space. For more information about related work, see chapter 3.

Furthermore, reinforcement learning has also been used to train search agents for Sokoban. Reinforcement learning algorithms, such as Q-learning or deep reinforcement learning methods like Deep Q-Networks, have been utilized to train agents capable of solving Sokoban puzzles with good effectiveness.

Overall, research on the Sokoban game contributes to the field of AI and game theory by pushing the boundaries of intelligent and efficient decision making and problem solving capabilities.

2.2 Graphs

- **Directed graph**
  A directed graph G is a pair $G = (V, E)$ where:

  V is a set of vertices with a vertice $v \in V$, we will call a vertice also a node, and

  E is a set of edges with an edge $e \in E$ with $E \subseteq V \times V$, the pairs are ordered.

  A graph is ordered if E is a set of ordered pairs of distinct vertices.

- **Weighted graph**
  A weighted graph G is a tuple with $G = (V, E, g)$ where (V, E) are the sets of vertices and edges of a graph and $g : E \longrightarrow \mathbb{R}$ is a weighting function.
  The weight of an edge (v, w) is defined as $g(v, w)$.

2.3 Search Algorithms

Graph search algorithms are used to traverse and analyze graphs like we have introduced in 2.2.

They are used to find paths between two nodes of a given graph or to traverse the whole graph in a particular way. We use two types of graph search algorithms in our work: informed and uninformed search algorithms.

Uninformed search algorithms, also known as blind search algorithms, are those that do not have any information about the graph beyond its topology. These algorithms explore the graph by systematically visiting each node or edge until a goal node is found.

Informed search algorithms, also known as heuristic search algorithms, use additional information about the graph to guide their search towards the goal. This information is typically provided by a heuristic function that estimates the distance from the current node to a given target node. Informed search algorithms can be more efficient than uninformed search algorithms because they can avoid exploring nodes that are unlikely to lead to a solution.

There are two important characteristics a graph search algorithm can have:

- **Completeness:**
  A search algorithm is called complete if it is guaranteed to always find a solution if one exists. In other words, a complete search algorithm will always finish the search without getting stuck in an infinite loop.
- **Optimality:**
  If a search algorithm will always find the shortest path between two given nodes, which means the path with the lowest cost in a weighted graph or simply the shortest path in an unweighted graph, the algorithm is called optimal.

2.3.1 Breadth First Search (BFS)

Breadth First Search (BFS) explores all the nodes in a graph starting from a specified node (i.e. the root vertex) and visits the neighbours of the current node and adds them to a FIFO queue while continuing with the top node in the queue in the same way. Hence the nodes in the queue have to be kept in memory until they will be explored. The nodes in the queue are also called the *search frontier*.

Due to the fact that BFS does not take a heuristic into account the algorithm is classified as an uninformed search algorithm.

When searching for a specific goal node inside the graph the algorithm has to visit in the worst case every path to every node, hence the time complexity is $O(|V| + |E|)$.
The space complexity of BFS is $O(|V|)$. Since the algorithm has to store all unvisited nodes in the *search frontier*, it may not be suitable for problems with a very large $|V|$ as well as graph search problems where the nodes are generated during the search since we do not know how large $|V|$ will become, assuming that we do not have any domain knowledge about the problem.

BFS is complete and, in unweighted graphs[1], optimal. The optimality can be explained by the fact that BFS systematically visits the nodes of the first graph layer with respect to the starting or root vertex and continues with the next layer after having fully searched the previous layer.

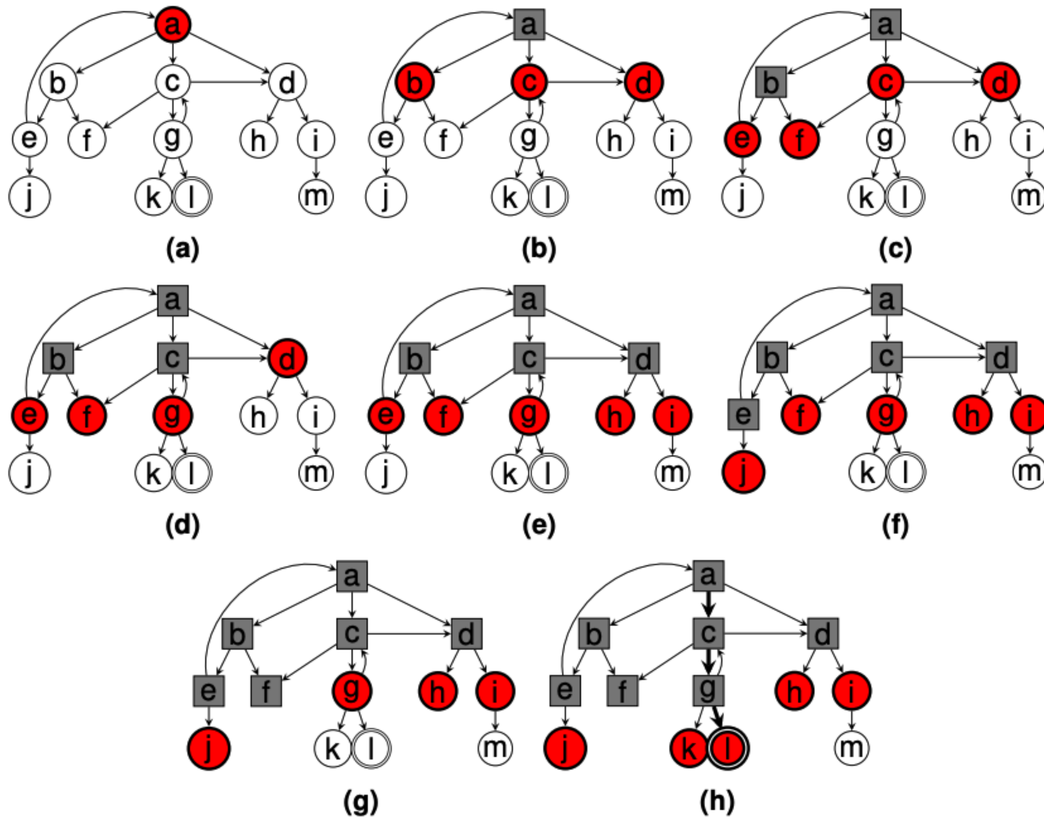
Figure 2.1 shows an example of how BFS would search a graph and in Algorithm 1 there is a pseudocode of BFS.

Algorithm 1: Breadth-First-Search (BFS)

```
Input  : graph G, start node s
Output: path to a goal node g, assuming it exists in G
let Q be a FIFO queue;
let visited ⟵ [s];
Q.push(s);
while Q ≠ ∅ do
    node := Q.pop()
    if node == g then
        return path to node;
    end
    foreach neighbour e of node do
        if node ∉ visited then
            Q.push(e);
            visited.add(e);
        end
    end
end
source: [4]
```

[1]if the graph is weighted and all edge weights have the same values, BFS is optimal too

Figure 2.1: BFS walk through a graph



a is the start node and *l* the goal node. White nodes are unexplored, while grey nodes are fully explored with all its neighbours marked in red, which is the *search frontier*. The algorithm ends in figure (h) as the goal node *l* has been found and marked.
source: [3]

2.3.2 Dijkstra's search algorithm

Dijkstra's search algorithm is a common and widely used algorithm to find the shortest path between a given start node and all other nodes of a weighted graph $G = (V, E, g)$ where the cost function g represents the costs of traversing an edge $e \in E$. The algorithm belongs to the group of uninformed search algorithms, because the algorithm is not using any heuristic.

Dijkstra's algorithm first initializes a distance array which symbolizes the costs to travel from a given start node to all other nodes in a graph. The initialized values inside the distance array are all set to ∞. The algorithm visits the nodes that are close to the current start node iteratively and updates the cost of reaching each neighboured node. Dijkstra's algorithm does this until all nodes in the graph have been visited. An example of how Dijkstra's algorithm find the shortest paths from a start node to all other nodes in a graph is shown in Figure 2.

The algorithm is known to be optimal and correct when using a graph without negative edge weightings, given that, when the algorithm determines, it will return the shortest path in G from the start node to every node $n \in V$ [5].

Dijkstra's algorithm has a time complexity of $O(|V|^2)$ in its original version. While the algorithms time complexity depends also on the data structure used to store the nodes that should be visited or already have been visited the time complexity can be optimized by using a priority queue (binary heap) to store those nodes. By doing that the algorithms time complexity would be $O(|E| \log |V|)$. If using a more complex data structure called Fibonacci heap the time complexity can be decreased to $O(|E| + |V| \log |V|)$[6].

Using a priority queue, the space complexity of Dijkstra's algorithm is $O(|V|)$. This is because the algorithm needs to maintain the queue to store maximum $|V|$ nodes and it needs to store a set of already visited nodes as well, which also has a maximum size of $|V|$. Since these are the only two data structures needed to maintain all information, the algorithm needs, and both structures have a space complexity of $|V|$ the algorithms space complexity is also $|V|$.

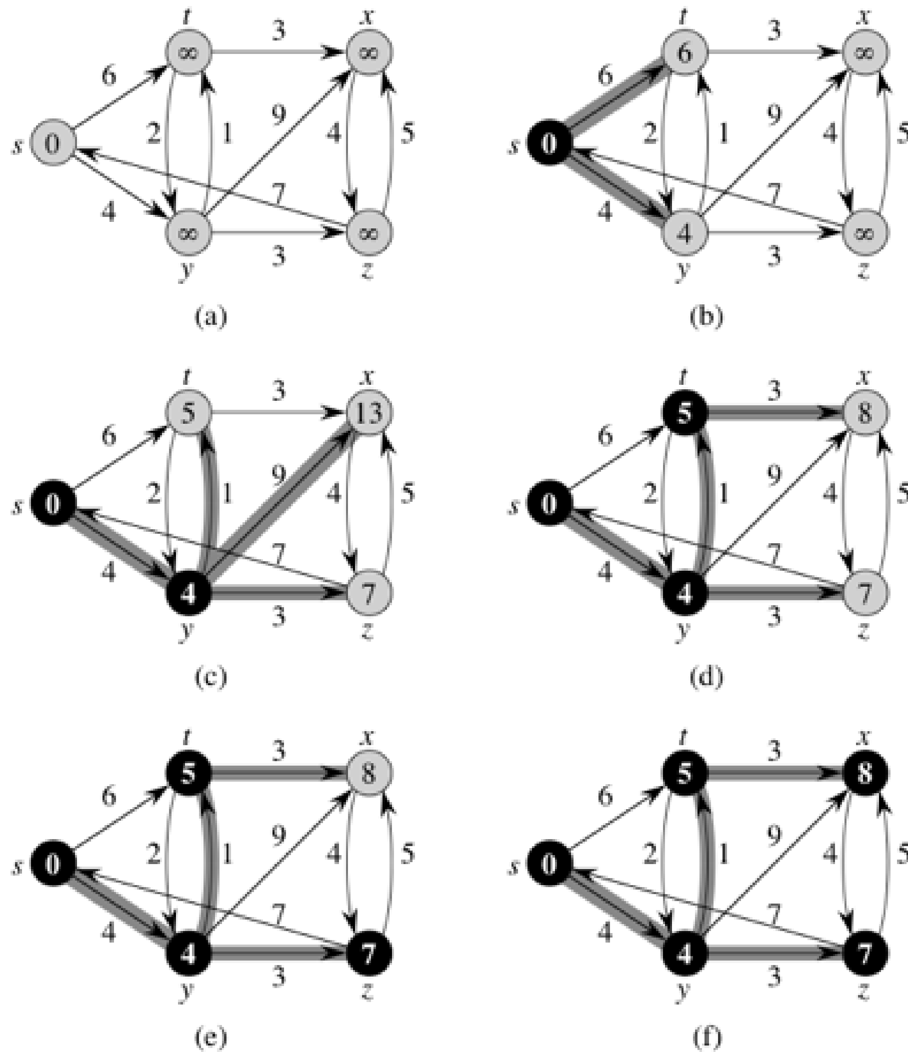
Algorithm 2 shows a possible pseudo code realization of Dijkstra's algorithm using a priority queue.

Algorithm 2: Dijkstra's algorithm with priority queue

```
Input  : graph G = (V, E), start node s, goal node g
Output: shortest path from start node s to a goal node g, assuming it exists in G
let dist[v] = ∞ for each v ∈ V;
let dist[s] = 0;
let Q be a priority queue containing for all v ∈ V = dist[v];
while Q ≠ ∅ do
    node := Q.get() // returns the item with min(dist[node])
    if node == g then
        return path to node;
    end
    if node ∉ visited then
        visited.add(node);
        foreach neighbour e of node do
            alt ⟵ dist[e] + distance;
            if alt < dist[e] then
                dist[e] ⟵ alt;
                Q.add(e);
            end
        end
    end
end
source: [7], https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
```

As we just want to know the shortest path between the start node s and a given goal node g we stop the algorithm when we get g from Q. The *distance* value is the costs from s to a current *node* the algorithm is visiting.

Figure 2.2: Dijkstra's algorithm search shortest distances to all nodes in a graph



In a) the initial graph $G = (V, E)$ and the start node $s \in V$ is shown. The algorithm starts iteratively by visiting the neighbours of s and updates the shortest paths to a node if there is one found. In the final picture f) the algorithm has determined. The grey marked arrows are the shortest paths to the nodes starting from s.

source: `https://www.cs.dartmouth.edu/~thc/cs10/lectures/0509/0509.html` *(visited: 06.05.23)*

2.3.3 A* search algorithm

The A* algorithm is a generalisation of the Dijkstra's algorithm and was first described in 1968 by Peter Hart, Nils J. Nilsson and Bertram Raphael. It uses a heuristic to guide the search towards the goal, meaning it uses some domain knowledge in the search space of a graph $G = (V, E, g, h)$. Here g is again a cost function where $g(n)$ is the cost from the start node to a node $n \in V$ and additionally a function h which is the algorithms heuristic function with $h(n)$ describes the estimated cost from a node $n \in V$ to a goal node.

The algorithm maintains two lists: an open list and a closed list. While the open list contains all nodes that are being considered for exploration and on the other hand the closed list contains all nodes that have been explored. Usually the open list is implemented as a priority queue data structure.
Exploration describes a process of traversing the search space to find a given goal node. The process does this by visiting the start node and iteratively visiting the neighbours of that node and adding them to the open list until a goal node is found.

In every iteration A* selects the node with the lowest $f(n)$ value from the open list. Where the function f is defined as $f(n) = g(n) + h(n)$. After selecting the node n the algorithm expands the selecting node by visiting its neighboured nodes and adding them to the open list if they are not already have been added to the closed list. For every neighboured node the f value is calculated and then node n will be updated if f through the selected node is less than any previous value for that node, meaning that a cheaper path to that node was found.
The algorithm stops if the goal node was found or there are no more nodes in the open list, meaning that there is no path in G from the start node to the goal node. A visualization of how A* searches for a goal node can be found in Figure 2.3. And a pseudo-algorithm in Algorithm 3.

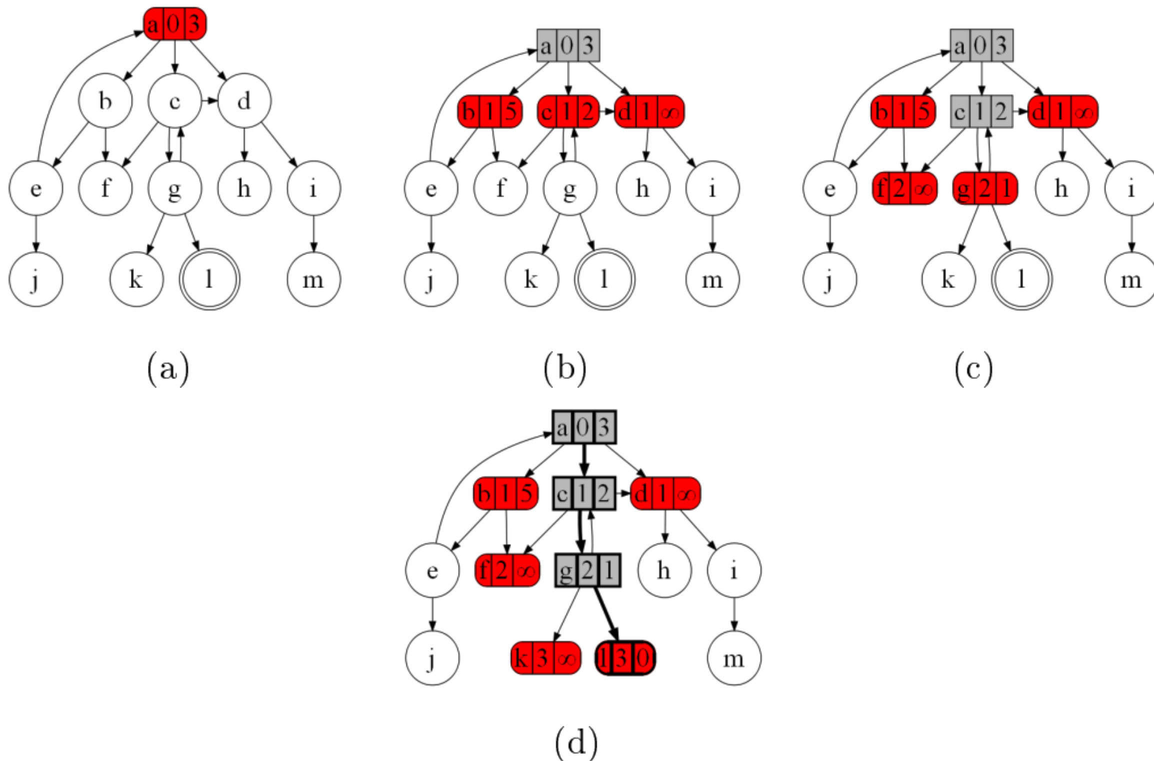
A* is known to be optimal[2] and complete [8].

The time complexity of A* is strongly connected to the used heuristic and depends on how many nodes the algorithm actually visits with the used heuristic. Speaking about the worst case time complexity this upper limit must not have much significance. However the time complexity is $O(|E_0| \log |V_0|)$, where E_0 are the visited edges from A* and V_0 the visited nodes [9].

In the worst case the algorithm has to store every node of G in the open and closed list which leads to the problem that for large search spaces A* can run into a large memory overhead.

[2] assuming that the given heuristic function h is admissible, see chapter 2.3.5

Figure 2.3: A* searches for a goal node



A* starts the search at the start node a. The goal node in this example is node l. The second value of each node shows the cost value from $g(n)$ and the third value the heuristic value $h(n)$. In picture d) all nodes were explored, except the nodes e, j, h, i, m. In this example the heuristic is *perfect*, meaning that the assumed value $h(n)$ is indeed the real length of the remaining path from the actual node to the goal node.
source: [3]

2.3.4 IDA* search algorithm

The IDA* algorithm is a variant of the A* search algorithm using iterative deepening to traverse the search space, meaning that it will start searching with a depth limit of one and increasing this bound step by step. In each iteration step a new upper bound is computed for a current node (the f value, defined in 2.3.3). If the computed f value is greater than the current depth limit the node will be pruned and the search will backtrack to the parent node. Algorithm 4 shows IDA* in an recursive variant.

Due to the fact that A* can lead to a memory overhead for large search space domains, IDA* has the advantage that it stores only the current path being explored and not every node, hence it uses less memory than A*. However IDA* might be revisit multiple nodes due to the depth limit, which means that it could be less time efficient than A*.

Algorithm 3: A* algorithm

```
Input  : graph G = (V, E), start node s_V, goal node g_V
Output: shortest path from start node s_V to goal node g_V, assuming it exists in G
let visited ⟵ ∅ be a set of visited vertices;
visited.add(s_V);
s_V.distance = 0;
openList.push(s_V, 0) // the nodes are ordered by the priority (second argument: estimated
  total distance);
while openList ≠ ∅ do
    node = openList.pop();
    visited.add(node);
    foreach neighbour succ of node do
        if visited.contains(succ) then
            continue
        end
        if succ == g_V then
            return pathTo(succ);
        end
        f ⟵ g(node) + cost(vertex, succ) + h(succ);
        if succ ∈ openList then
            if succ.distance < node.distance + cost(vertex, succ) then
                succ.distance = node.distance + cost(vertex, succ);
                succ.predecessor = node;
                openList.update(succ, f);
            end
        else
            succ.distance = node.distance + cost(vertex, succ);
            succ.predecessor = node;
            openList.add(succ, f);
    end
end
source: [8], [10]
```

Algorithm 4: IDA* in a recursive variant

```
Input  : graph G = (V, E), start node s_V, goal node g_V
Output: shortest path from start node s_V to goal node g_V, assuming it exists in G
let path be the current search path (data structure: stack) and node the current node;
let g the cost to reach the current node and cost(node, succ) a step cost function;
Function IDA*(s_V) is
    bound ⟵ h(s_V);
    path ⟵ [s_v];
    while true do
        t ⟵ search(path, 0, bound);
        if t == FOUND then
            return (path, bound);
        end
        if t == ∞ then
            return NOT-FOUND;
        end
        bound = t;
    end
end
Function search(path, g, bound) is
    node ⟵ path.last;
    f = g + h(node);
    if f > bound then
        return f;
    end
    if node == g_V then
        return FOUND;
    end
    min ⟵ ∞;
    foreach neighbour succ of node do
        if succ ∉ path then
            path.push(succ);
            t ⟵ search(path, g + cost(node, succ), bound);
            if t == FOUND then
                return FOUND
            end
            if t < min then
                min = t
            end
            path.pop()
        end
    end
    return min
end
```

Source: `https://en.wikipedia.org/wiki/Iterative_deepening_A*`*, visited: 07.05.23*

2.3.5 Heuristics for search algorithms

The search algorithms A* and IDA* use heuristics. A heuristic ensures that the search in the search space is enhanced by its heuristic values and thus that the search for a goal node is directed more efficiently.

For our purpose we can denote a heuristic function $h : V \longrightarrow \mathbb{R}; n \mapsto h(n)$, where n is a node of a Graph $G = (V, E), n \in V$. The output value $\mathbb{R}$ of $h(n)$ is the approximated cost from the node n to a goal node. For our needs we must consider two different characteristics of $h(n)$:

- **admissible:**
  Let a function $g(n)$ be the actual cost from a node n to a goal node. A heuristic $h(n)$ is called admissible if:
  $$h(n) \leq g(n)$$
  meaning that the heuristic approximation is always less than or equal to the actual cost from n to the goal node, so it provides a lower bound on the actual cost to reach the goal. An admissible heuristic guarantees an optimal solution for an optimal search algorithm using that heuristic. Note that the use of an admissible heuristic does not mean that the search itself will find the solution quicker or in a more efficient way only that the found solution is optimal.

- **monotone:**
  Let us define the function $cost(n, succ(n))$ or short denoted $c(n, n')$, where $c(n, n')$ is the cost travelling from node n to node n'.

  Our heuristic $h(n)$ is called monotone if:
  $$h(n) \leq c(n, n') + h(n')$$
  $$\text{and } h(goal\ node) = 0$$
  meaning that the approximated cost of reaching a goal node from node n is always less than or equal to the cost of reaching the goal through the successor of node n plus $h(n')$. Monotony implies also efficiency, because every node is visited only once. This means also that all monotone heuristics are also admissible.

2.3.6 Bounded Relaxation

Bounded relaxation is a technique which we will use for A*. In chapter 2.3.5 we explained the admissibility criterion. Assume that we have an admissible heuristic $h(n)$ used in an A* algorithm, this will also mean that A* has to check all equivalent ways (due to $h(n)$) to find the optimal way to a goal node which could lead, especially in large search spaces, to a high amount of computing time.

Fortunately it is possible to speed up the search process by relaxing the admissibility criterion, which leads to the loss of the guaranteed optimality of A* with $h(n)$. By losing the optimality to find a goal node faster, we want to at least give a guarantee that the found solution is no worse than $(1 + \varepsilon)$ times the optimal solution path, meaning that we have a bounded relaxation called *ε-admissible*.

There are different *ε-admissible* bounded relaxation techniques. We want to introduce two of them[3]:

- **Static Weightings / Weighted A*:**
  Let $h(n)$ be still our admissible heuristic used by A*. The weighted version of A* uses a modified heuristic with a static value of ε, the function $h_s(n) = \varepsilon h(n), \varepsilon > 1$. Using h_s, A* can find a path with a cost of at most ε times of the optimal path cost of the search space.

- **Dynamic Weighting:**
  The dynamic weighting uses a modified heuristic $h_d = h(n)(1 + \varepsilon\omega(n))$, with

  $$\omega(n) = \begin{cases} 1 - \frac{d(n)}{N} & \text{if } d(n) < N \\ 0 & \text{otherwise} \end{cases}$$

  Here $d(n)$ is the depth of the search for the current node n and N is the anticipated length of the complete solution path.

2.4 Transposition Tables

With a Transposition Table we can reuse previously evaluated states found by a search algorithm like A* in a search space.

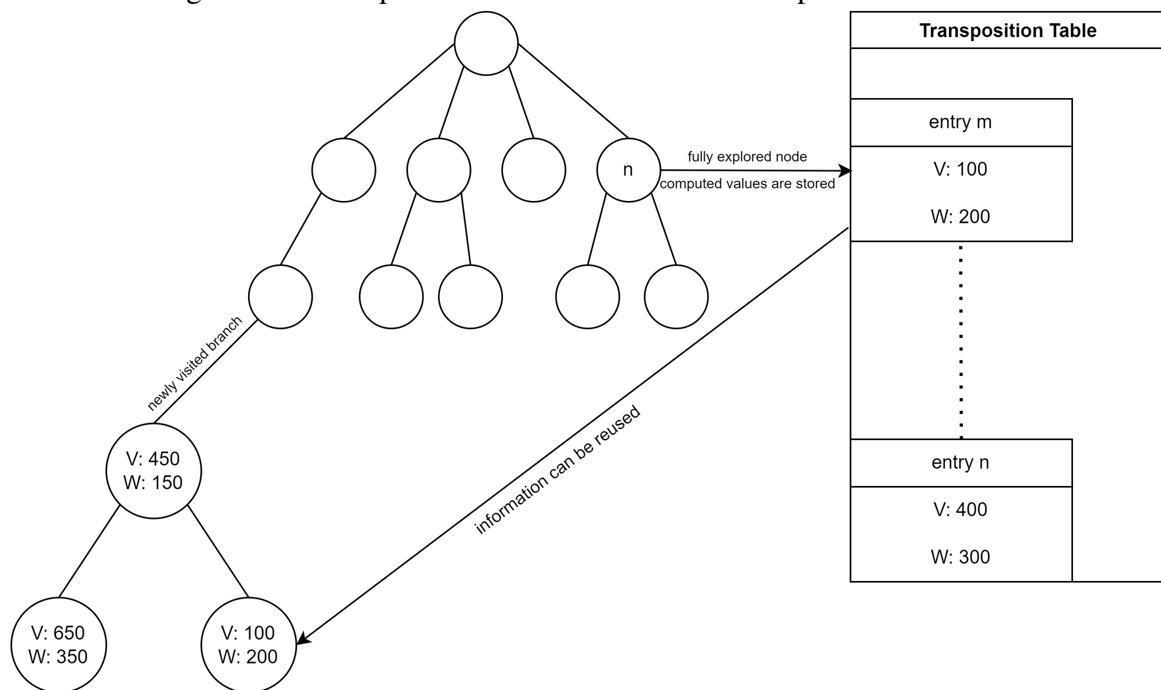
While the algorithm is exploring nodes it might expand already explored nodes in the graph, this will usually happen more often in search spaces with a large branching factors. To decrease the computing time a Transposition Table can be used to store the mapping between a specific state and the evaluated results, which will be for example the computed heuristic value of a node.

In context of game theory the Transposition Table can furthermore store additional information about the actual board situation of a node. For example by storing information about the best move found so far or the depth of the search, which can be assisting information to prune the search at the actual node or at expanding nodes from the actual node. A visual example is given in Figure 2.4.

The data structure of the Transposition Table is usually build as a key-value pair structure where the key is a hashable information one can get from the explored node and the value is all the information one would store about the actual node. A concept of a Transposition Table hash function is also described by Zobrist [11].

[3] both techniques are taken from: `https://en.wikipedia.org/wiki/A*_search_algorithm` (10.06.23)

Figure 2.4: Transposition Table used to reuse computed node values



The Transposition Table stores two pseudo values for a node V, W. While a new node is expanded the algorithm recognizes the node and can find the corresponding values inside the Transposition Table instead of recompute the values V and W.
source: own creation, idea from: `https://i.stack.imgur.com/yWs8n.jpg`, *visited: 16.05.23*

2.5 Sokoban

Sokoban (Japanese for warehouse keeper) is a single-player puzzle game originally published in December, 1982 by Hiroyuki Imabayashi as a PC-game[4]. Today, there exist several versions and modernization's of the game and a bunch of self made levels build by the Sokoban community.

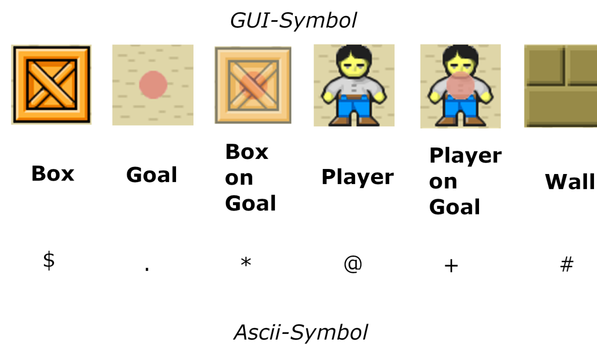
A Sokoban level is a two-dimensional room representing a warehouse and contains a player, walls, arbitrary placed boxes and goals for the boxes. The goal of the game is for the player to move all the boxes to their goals.

2.5.1 Rules of Sokoban

A level can be represented as a two dimensional $n \times m$ room with squares, where each square can be exactly one of the objects:

[4] `https://en.wikipedia.org/wiki/Sokoban` (21.05.2023)

Figure 2.5: Objects in a Sokoban level



source: the graphic was created by using JSoko `https://www.sokoban-online.de/`

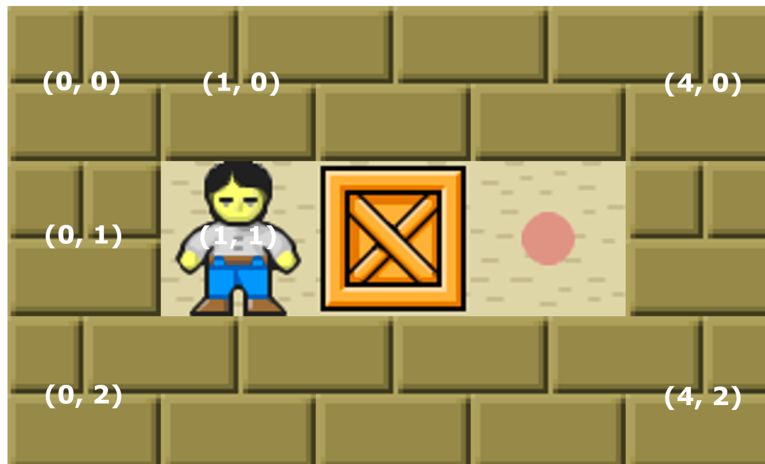
We define the following rules for Sokoban:

1. If a square in the level does not contain one of the listed objects from Figure 2.5, it will be called a floor square and is denoted as a *BLANK-SPACE* ASCII character
2. There has to be exactly one player on the board
3. The player can move freely on a floor and over goals in the directions *up, down, left, right*, we will call a move a push if the player moves a box with his move
4. The player can push one box at a time (but not pull)
5. The player can not move through walls or boxes
6. A legal level is surrounded by walls so the player and the boxes can not leave the level
7. There can only be one box on a specific goal at a time, a box can be pushed away from a goal as well
8. A legal level must contain at least as many goals as there are boxes
9. A level is completed or solved if there was found a solution
10. A solution in Sokoban is described as a sequence of moves and pushes by the player so that every box is on a specific goal.
11. For a solution the order of which box will stand on which goal does not matter

Each square has a (x, y) coordinate so that we can locate any square on the board. The coordinates are starting with $(0, 0)$ and goes up to (n, m) for each axis x and y (see Figure 2.6).

As already said we call a move a push if the player moves and pushes a box. A push contains a starting position (x, y) and a direction (l, r, u, d) where each letter stands for one of the four cardinal directions the player can move.

Figure 2.6: The Sokoban $n \times m$ level grid



source: the graphic was created by using JSoko `https://www.sokoban-online.de/`

Figure 2.7: Sokoban level in ASCII notation

```
#####
#@$.#
#####
```

The ASCII notation of the Sokoban level shown in Figure 2.6
source: own creation

Furthermore we want to define a legal push. A push is legal, when:

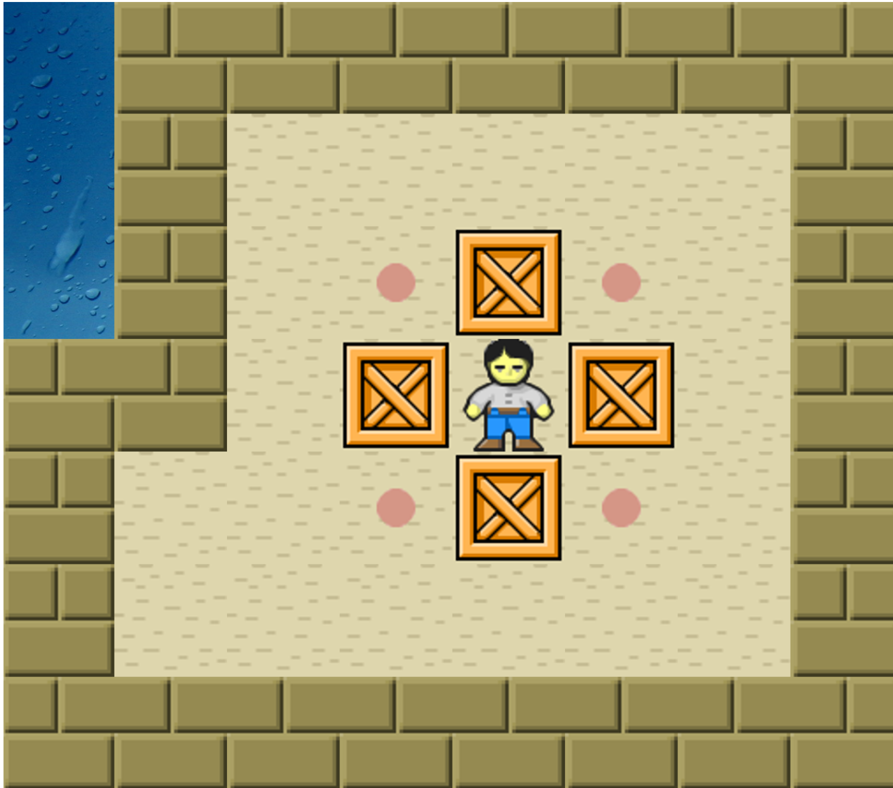
1. a player stands in front of a box (means that he can reach the square in front of the box)
2. if the player moves against the box, there must be a floor or a goal on the square behind the box relative to the direction the player moves the box

To denote a solution of a Sokoban level there are several options:

- storing the solution with the total number of push moves the player has to make, so that all boxes are pushed onto a goal
- storing the *LURD*-String, which is the ordered sequence of the direction of each push and move from the starting point of the player to the last push when the last box is pushed onto a goal

The *LURD*-String contains the starting letter of each direction of a move and the capitalized starting letter if the move was a push. Since we can easily calculate the number of pushes and moves from the *LURD*-String we will use this String notation to store the level solutions, see Figure 2.8 for a solution example.

Figure 2.8: Solution example of the 5th level from the *Microban I* level library



A possible solution for this level is: *LuRllDrdRdrruuLLdlUddlluR*
source: the graphic was created by using JSoko `https://www.sokoban-online.de/`

2.5.2 Why solving Sokoban levels is interesting

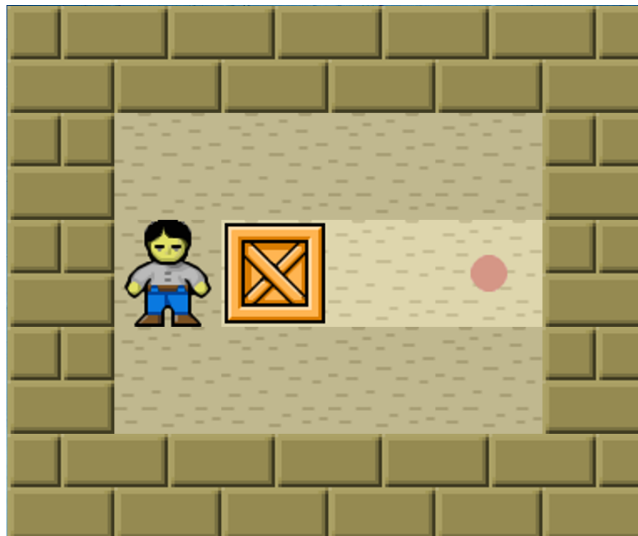
Solving small Sokoban levels seams easy for humans and computers but levels can become very large and with a high amount of boxes they will be hardly solvable by humans and even computer solvers need to do a high amount of computations to find a solution, this becomes even harder for humans and computer when the task is to find the optimal solution, meaning that a solutions *LURD*-String is the shortest solution string possible for a level to solve.

Sokoban was shown to be NP-Hard [1] and even PSPACE-complete [2], see also chapter 3. Therefore, Sokoban levels can have a high branching factor. If we assume that the branching factor in the relaxed move space is equal to the number of possible pushes of a current state, then the highest branching factor in the famous *XSokoban* level set is 136 [12]. For some levels even the optimal solution requires a long move / push sequence, meaning that Sokoban can have levels with a high solution depth. Some levels can have an optimal solution length that requires over 600 moves [12]. But not just the high branching factor and solution depth makes it difficult to find a solution. There is the existence of deadlocked states which makes the search even harder.

Deadlocked states are game states in the search space of a level where, once the state is deadlocked, the whole level becomes unsolvable. A deadlocked state has at least one deadlock. A deadlock is mostly a box that can not be pushed away anymore or boxes that will block each other. There are several types of deadlocks and we will list a few:

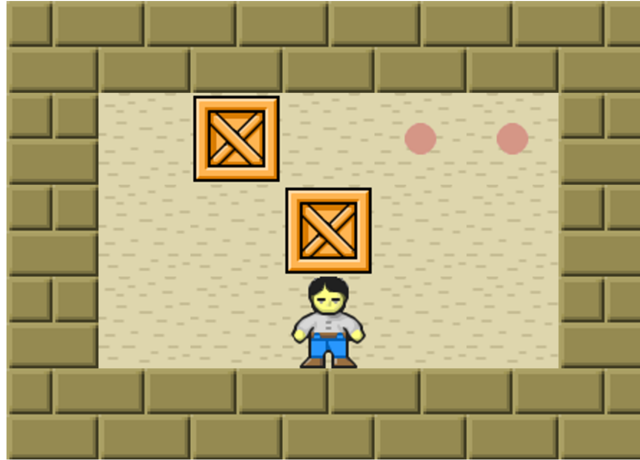
- **Simple deadlocks**
  Simple deadlocks or dead squares are squares in the level that, if a box is pushed onto them, the player can not push it away from that square or is not able to push this box to a goal anymore, hence pushing boxes to dead squares will lead to a deadlock, see Figure 2.9 for an example.
- **Freeze Deadlocks** Freeze Deadlocks are game states that contain at least one frozen box. A box is frozen if the box becomes immovable and is not located on a goal square, see Figure 2.10 for an example. Sometimes it is a series of boxes in a row that affect each other and will lead to a Freeze Deadlock, an example for this can bee seen in Figure 2.11.
- **Dead and Frozen Boxes Deadlocks** As we said a frozen box will not create a deadlock if the box is on a goal. But since the box is on a goal square and can not be moved away the box might block an area so that the level will become unsolvable, see Figure 2.12 for an example.

Figure 2.9: Simple Deadlocks in a Sokoban level



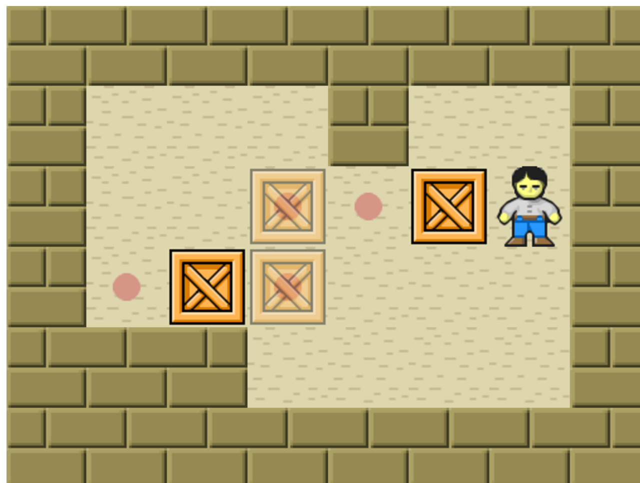
All simple deadlocks are marked with a darker color. In this example only the goal, the box and the floor between the both squares is not a simple deadlock.
source: the graphic was created by using JSoko `https://www.sokoban-online.de/`*, the level construction was found on:* `http://sokobano.de/wiki/index.php?title=Deadlocks`

Figure 2.10: Freeze Deadlock in a Sokoban level



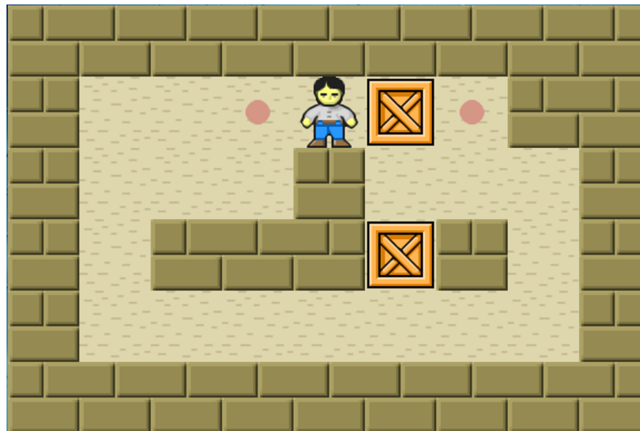
If the player pushes the box next to him up he will not be able to move both boxes again.
source: the graphic was created by using JSoko `https://www.sokoban-online.de/`*, the level construction was found on:* `http://sokobano.de/wiki/index.php?title=Deadlocks`

Figure 2.11: Recursive Freeze Deadlock in a Sokoban level



If the player pushes the Box next to him to the left the whole situation becomes a deadlocked state. That is because the pushed box will block all the other boxes on their goals but since there is one box not on a goal and unreachable after the push the state is frozen.
source: the graphic was created by using JSoko `https://www.sokoban-online.de/`*, the level construction was found on:* `http://sokobano.de/wiki/index.php?title=Deadlocks`

Figure 2.12: Deadlock due to a frozen box



If the player pushes the box next to him to the right, then he will not be able to move this box again. In this situation, the pushed box will block the space for the second box so the player can not move the box to the other goal.

source: the graphic was created by using JSoko `https://www.sokoban-online.de/`*, the level construction was found on:* `http://sokobano.de/wiki/index.php?title=Deadlocks`

3 Related Work

3.1 Complexity

Sokoban was shown to be NP-Hard by Dor and Zwick in 1996. They also showed that the problem is in PSPACE [1]. A year later Culberson showed that Sokoban is even PSPACE-complete [2].

3.2 Rolling Stone Solver

The first published Sokoban solver and attempt to solve Sokoban in an efficient way was done by Junghanns and Schaeffer in 1998. To realize their solver they developed a pattern search algorithm to detect deadlocks and prune them from the search space. The solver is based on IDA* and according to them with pattern search they reached a 15-fold reduction of the time IDA* needs to explore the nodes on a given depth for each IDA* iteration [13].

Junghanns and Schaeffer improved their solver over several years by also adding techniques that lead to semi optimal solutions (i.e. by overestimating through bounded relax methods, as described in chapter 2.3.6) as well as deadlock tables and macro moves. Lastly they reached a best performance of solving their own test set by adding techniques like relevance cuts and goal room macros, which enables the solver to push boxes in a more effective way to the goals, especially when it comes to a specific and more complicated order the boxes have to be placed onto the goals.

According to them their solver was able to solve 54 levels out of 90 from the used test set, also known as the *XSokoban* test set. [12].

Over years there were several attempts to develop new techniques for solvers to solve the complete *XSokoban* test set. *YASS* and *JSoko* are two more solvers that uses different and improved techniques like optimized heuristics and larger trained pattern databases to detect deadlocks and level structures like tunnels as well as improved search algorithms using more threads to compute on. With these improvements both solvers were able to solve more levels in the *XSokoban* test set than the Rolling Stone Solver[1].

[1] see `http://sokobano.de/wiki/index.php?title=Solver_Statistics` (visited: 23.05.2023)

3.3 Festival Solver

The Festival Solver is the first solver that was able to solve all *XSokoban* set levels and was written by Yaron Shoham and published in 2020[2]. The solver uses a new developed search algorithm called FESS, which stands for *Feature Space Search Algorithm*. The authors call heuristics features and the algorithm uses a set of domain-specific features, where a domain is the problem of solving a Sokoban level. The algorithm is proven to be complete but is focused on finding solutions in a domain space very fast instead of short solutions or the optimal solution [14]. See Figure 3.1 for the original pseudo FESS algorithm.

Shoham and Schaeffer introduced new heuristics (called features) for Sokoban for example the *Connectivity Feature*. When the level room is divided into spaces that are separated by the boxes and the player can not reach another space, then the heuristic value will increase [14], see Figure 3.2.

The FESS algorithm can solve all 90 levels in *XSokoban* in less than 4 minutes [14].

Figure 3.1: The FESS algorithm

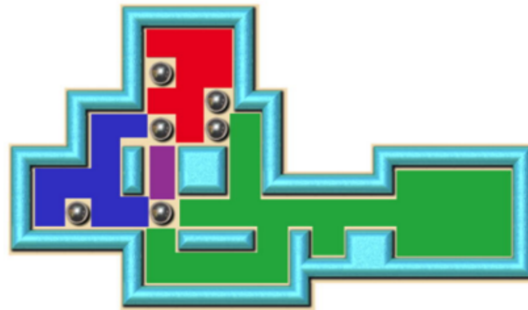
```
Initialize:
Set feature space to empty (FS)
Set the start state as the root of the search tree (DS)
Assign a weight of zero to the root state (DS)
Add feature values to the root state (DS)
Project root state onto a cell in feature space (FS)
Assign weights to all moves from the root state (DS+FS)

Search:
while no solution has been found
  Pick the next cell in feature space (FS)
  Find all search-tree states that project onto this cell (DS)
  Identify all un-expanded moves from these states (DS)
  Choose move with least accumulated weight (DS)
  Add the resulting state to the search tree (DS)
  Added state's weight = parent's weight + move weight (DS)
  Add feature values to the added state (DS)
  Project added state onto a cell in feature space (FS)
  Assign weights to all moves from the added state (DS+FS)
```

Here *FS* stands for feature space and *DS* stands for domain space, with transitions from state to state with moves by the player [14]

source: [14]

[2]see `https://festival-solver.site/` (visited: 23.05.2023)

Figure 3.2: Connectivity Feature example



Each colored area stands for a space divided by the boxes the player can not reach without pushing the corresponding boxes.
source: [14]

3.4 Pre-calculation approach

As already figured out the existence of deadlocked states makes searching for a solution hard. But what if the search algorithm knows a bunch of deadlocked states so the algorithm will have a better guidance to a solution inside the search space.

Cazenave and Jouandeau followed this approach by pre-calculating deadlocks and storing them in a deadlock table by doing a retro grade analysis for Sokoban levels. Through improved deadlock detection, they achieved that the classic IDA* algorithm had to search substantially fewer nodes than the variant in Rolling Stone. However, a lot of time had to be spent to fill the deadlock tables with information from the retrograde analysis [15].

4 Implementation - Conception and Realization

In this chapter we want to introduce our solver and the used techniques, as well as introducing the Graphical User Interface (GUI) to control the basic functionality of the solver in an easy way and the corresponding system architecture that comes with the solver and the GUI.

4.1 Python

For our solvers' implementation we make use of the high level programming language Python conceived in the late 1980s by Guido van Rossum. Our implementation is build with Python version 3.11.0, released on the 24. Oct. 2022.[1].

We use Python because the programming language is flexible and easy to understand. It is supported on all major operating systems. The programming language offers the developer a number of simplifications, such as a garbage collector and object-oriented programming. The focus of programming with Python is therefore on the implementation of the actual algorithm without having to worry about memory management and other issues. The downside is that the execution of individual instructions is relatively slow compared to more hardware-oriented programming languages such as C.

To extend the capabilities of Python and optimise the overall coding experience, we use common Python modules from third-party developers.

4.1.1 Used Python modules

- numpy (v. 1.25.0)
  numpy is a well known open source library for Python with a core written in C language[2]. Thus we use several C data types in our solver instead of Python's builtin data types to ensure a better performance in general.
- customtkinter (v. 5.2.0)
  customtkinter is an open source library for Python written by Tom Schimansky[3] and is used to extend Python's builtin tkinter module. The library allows us to create tkinter objects faster and with easier customization. As we want to provide a graphical user interface to control the solvers behaviour both modules are used to realize the graphical user interface.

[1] Python releases: `https://www.Python.org/downloads/source/`
[2] numpy: `https://numpy.org/`
[3] customtkinter: `https://github.com/TomSchimansky/CustomTkinter`

- pandas (v. 2.0.2)
  pandas is a Python library for fast and effective data analysis and manipulation.[4] We use this module to convert the solvers generated statistics into readable CSV files.
- dlib (v. 19.24.1)
  dlib is a library written in C++ containing machine learning algorithms[5] which we make use of as the used algorithms are implemented in a more effective way which will speed up the general solver performance as implementing the same algorithms in Python might be slower.
- pygame (v. 2.4.0)
  pygame is a popular library to build simple video games[6]. We use pygame for our game mode described in chapter 4.4.

Note: by installing these modules to a Python interpreter, some other helper modules may be installed, which we didn't list here as their only usage is to run or install the listed modules itself correctly.

4.2 Solver Development

Our solver consists of several feature components, which can be dynamically switched on and off via a solver configuration, as well as several search algorithms, one of which can be selected at a time. The solver will use different methods to explore the search space of Sokoban. Therefore we need to define what our search space for Sokoban looks like.

4.2.1 General program sequence

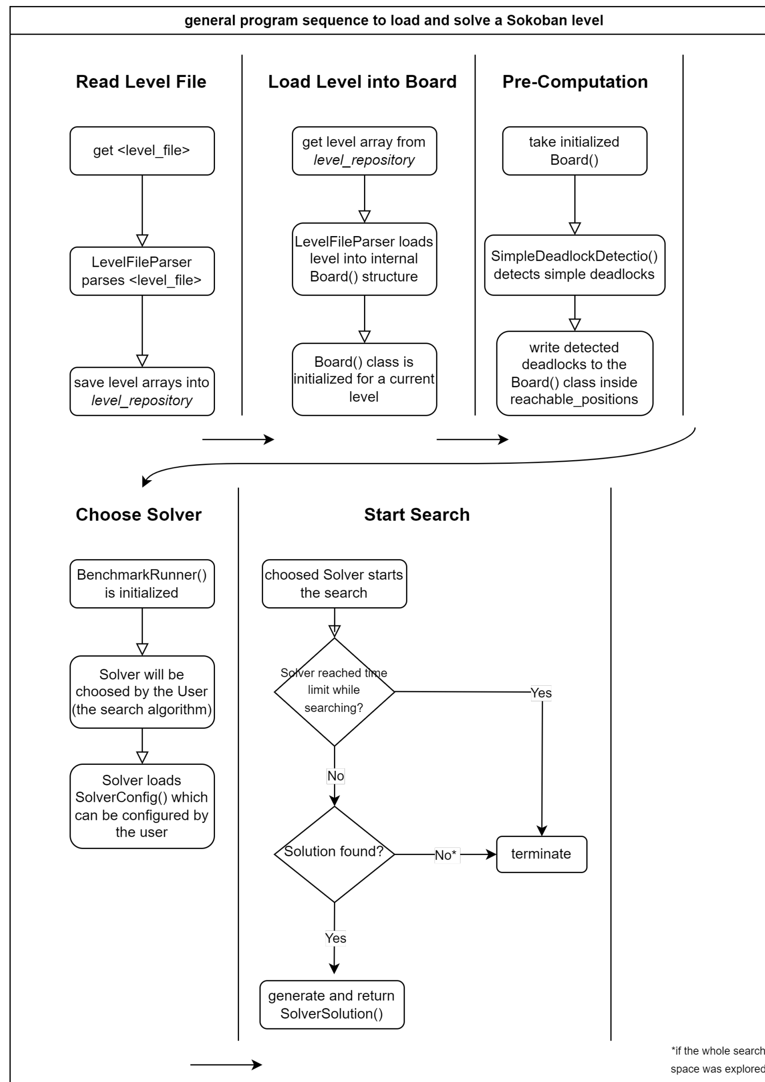
To solve a level, the solver first needs a level in the form of an ASCII array as shown in Figure 2.7. The solver can read in several levels at the same time in this format and then solve them one after another. How our solver reads in a level file, initialise a board class from the read level and then solve the level is described in Figure 4.1.

If a whole level set should be solved, the stages are started in a queue for each level. Then, the solver solutions will be appended for each level in a single CSV file.

The GUI provides the functionality to start a single run with all stages while giving the opportunity to change the solver configuration before starting the solver. More about the GUI in chapter 4.4.

[4]pandas: https://pandas.pydata.org/
[5]dlib: http://dlib.net/
[6]pygame: https://www.pygame.org/wiki/about

Figure 4.1: The general program sequence to solve a level



source: own creation

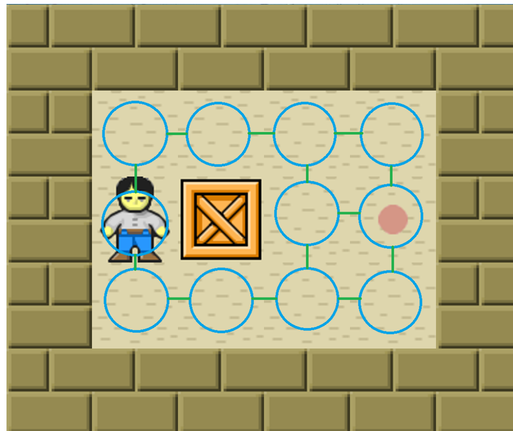
4.2.2 Search Spaces

We distinguish between two spaces, the *move space* and the *state space*. A similar concept of how to separate the search space for Sokoban was also introduced by [10]:

- **Move Space:**
  The nodes in the move space graph are all squares of a level state except squares with a box on them. The nodes are connected to an edge if the player can reach the respective square without moving a box (meaning that he can make a legal move to this square). See Figure 4.2 for an example. The move space graph will change every time a box is pushed, because by pushing a box there will be new squares reachable and unreachable for the player, so the edges between the nodes will change.

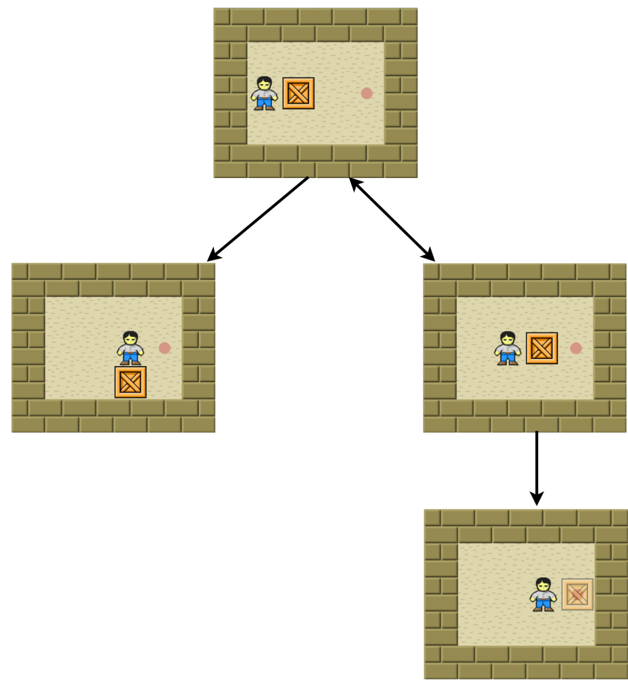
- **State Space:**
  The state space is a graph where the player can make moves, meaning that he is playing the actual game in that space, while each move will change either the player position or both the player position and a box position. Each change in at least one of these positions will lead to a new node inside the graph. For a better understanding of what the state space graph looks like, see Figure 4.3.

Figure 4.2: Move Space Graph Example



The blue circles are the nodes that the move space graph contains for this state. The green lines are edges symbolizing the connected nodes.
source: own creation, inspired by: [10]

Figure 4.3: State Space Graph Example



A new node in the state space graph is created when a box was moved. Since some positions can be reached again in the state space graph the graph can contain bidirectional edges. Note: This graphic shows the relaxed state space graph where a new node is created only when a box was moved. The non-relaxed state space will contain a new node if the player has moved too, see chapter 4.2.3.
source: own creation, inspired by: [10]

In chapter 4.2.3 we introduce two types of searching in the move space graph. One searching method will lead to the more relaxed state space graph described in Figure 4.3.

In the relaxed state space the graph will only change for each box that is movable from a current player position, meaning that changing the player position itself will no more lead to a new node if the player does not push any box.

Note that the graph in both versions the standard and the relaxed one can contain cycles, as different move sequences can lead to the same state where the player and the box positions are the same as in a neighboured node of the graph.

4.2.3 Move Generators

Our solver has two different move generators, one of which can be selected when the solver has to solve a level. We call the first move generator version the *Classic Move Generator* and the other one the *Extended Move Generator*.

- **Classic Move Generator**:
  The classic move generator searches in the move space. The algorithm checks the four cardinal directions *left, right, up, down* if a legal move (or push) can be made in that direction according to the defined rules in chapter 2.5.1. Each legal move or push will be added to an array and returned at the end, see Algorithm 5.

  Since the classic move generator only checks the four cardinal directions in the move space, the resulting state space can have a maximum degree of four.

Algorithm 5: Classic Move Generator

```
Input  : minifiedBoardState with (player, boxes),
         the current board instance
Output: possible moves
let moves ⟵ [];
let cardinalDirection ⟵ [(0, −1, l), (−1, 0, u), (0, 1, r), (1, 0, d)];
foreach cardinalDirection AS direction do
    let newPlayerPos ⟵ (player[0] + direction[0], player[1] + direction [1]);
    if newPlayerPos ∈ boxes then
        let newBoxPos ⟵ newPlayerPos[0] + direction[0], newPlayerPos[1] + direction[1];
        if newBoxPos ∉ board.walls and newBoxPos ∉ boxes then
            let move ⟵ createMove();
            moves.append(move);;
        end
    else
        if newPlayerPos ∉ board.walls then
            let move ⟵ createMove();
            moves.append(move);;
        end
end
return moves;
```

- **Extended Move Generator**:
  The extended move generator will also search the move space, but instead of just checking the four cardinal directions for possible legal moves or pushes, the generator will search for all legal pushes to all reachable boxes from the current player position. More specifically, the generator will return a move sequence of variable length with legal moves and a legal push (if there is at least one possible in the move space) at the end of the sequence.

We want the generator to generate a legal move sequence of minimal length to a legal push. To achieve this we use a simple breadth first search, which searches in the move space. The actual logic of whether a single move or push is legal and can be added to the move sequence can be used from the classic move generator.

With the extended move generator, the maximum degree of a node in the state space has the size of all current moveable boxes from all sides reachable by the player.

4.2.4 Board Representation

The board is our main object that will store all information about a Sokoban level. When loading a level from an ASCII representation as shown in Figure 2.7, a new board instance is created containing tuple arrays for each object in the loaded level.

Each tuple array contains the coordinates of where the square objects are placed on the level at the time the level is loaded. While there are non movable objects like walls and goal squares, there are also changeable objects like the player and the boxes. For that we can define the corresponding tuple arrays as read only or not whether, the objects are changeable.

Both move generator variants are placed in the board class as an implemented function. That is because both functions need the level arrays from the board class itself.

Furthermore the board class is able to check if a level has been solved or not by simply checking if all boxes are pushed onto a goal. For that we can simply check if the sorted tuple arrays for the boxes and the goals are identical.

Since the player position and the box positions are the only objects that are changeable of the board class, we will define a *minifiedBoardState* that only contains the player position as an (int, int) tuple and the box positions as a ((int, int), (int, int), ..., (int, int)) tuple.

4.2.5 Transposition Table

Our implementation of the Transposition Table uses the *minifiedBoardState* as the key and will store the, by a search algorithm computed, heuristic as the value. Since we use Python we can simply use the already defined data type *dict* (dictionary) for our Transposition Table as the dictionary data type uses by default key / value pairs and hashes the key. Hence our hash function fits the definition by Zobrist, defined in 2.4

Our Transposition Table fits to the definition by Zobrist [11], because each different *minifiedBoardState* also represents a different position of the level and can thus be uniquely assigned.

Our Transposition Table also supports the possibility of dynamically updating stored hash values if A* or IDA* find a better move sequence to a *minifiedBoardState* during their search.

4.2.6 Search Algorithms

Our solver mainly uses the breadth first search to search the move space. However breadth first search can be used to search the state space as well. To search the state space we also provide three other algorithms: Dijkstra's search algorithm, A* and IDA*.

All four algorithms are implemented as described in chapter 2.3. In some cases, slight adaptations, specific for Sokoban were made, but these do not change the basic functioning as described.

4.2.7 Heuristics

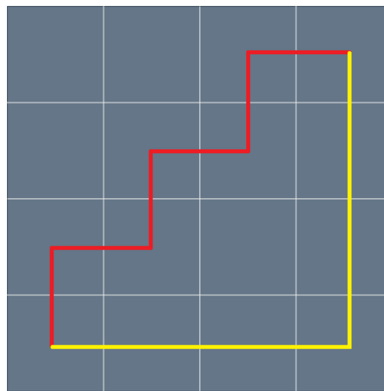
We use the Manhattan distance metric in A* and IDA* to estimate the number of pushes needed to solve the Sokoban level from a given position. We do this by calculating the Manhattan distance to the closest target for each box and then summing the individual values.
The Manhattan distance h_m between a box square $S_b = (B_x, B_y)$ and a goal square $S_g = (G_x, G_y)$, is defined as:

$$h_m((S_b, S_g)) = |B_x - G_x| + |B_y - G_y|$$

The summed metric then is the sum of all h_m values for all boxes in the Sokoban level to their nearest targets. Each box is assigned to exactly one target.

The Manhattan Distance is admissible because the player can only move a box along the four cardinal directions and not diagonally as seen in Figure 4.4.

Figure 4.4: The Manhattan Distance



The red line corresponds to the way the player pushes the box, while it has the same length as the calculated yellow line by the Manhattan metric.
source: own creation

4.2.8 Static and Dynamic Weighted A*

Our solver uses the bounded relaxation for A* in the static and dynamic variant as described in 2.3.6. The ε parameter can be freely chosen and set in the GUI presented in chapter 4.4. Both bounded relaxation variants can be switched by the start of the search or after a certain time. The threshold for the time can also be configured in the GUI.

In the dynamic variant, $d(n)$ corresponds to the depth of the search of a node n. The search depth in the search space of a Sokoban level corresponds to the number of pushes to a node n for the relaxed move space.

N corresponds in the dynamic variant to the anticipated length of the complete solution path and corresponds in the Sokoban level to the estimated number of moves needed until all boxes have been moved to a target, hence exactly the summed Manhattan distance as described in 4.2.7.

4.2.9 Deadlock Detection

Our solver implements two different deadlock detection methods[7]. Both methods take place in two different stages while solving. The solver uses two deadlock tables, each for every detection method. The deadlock table acts like a Transposition Table but instead of storing the heuristic value mapped by the minifiedBoardState, the information, if a specific position is a deadlock, is stored. Hence the solver does not have to calculate a once detected deadlock twice, it can simply look into the deadlock tables.

4.2.9.1 Simple Deadlocks

We can detect simple deadlocks by checking if a box can be pushed to a specific square, meaning that the square the box should be pushed onto must not be a deadlock square. To detect if a square is a dead square we can simply check if a box can be *PULLED* from a goal position to the square we want to check. If the box cannot be pulled from a goal to that square, we cannot push it from a starting point to that square either.

Since we do not want to calculate every time a box should be pushed if the goal square is a dead square, we can pre-calculate each dead square once and store it in the deadlock table, specifically for simple deadlocks. Then all we have to do is check if the square is inside the deadlock table, and if so, we can prune the move.

In our internal implementation we simplified this pruning by redefining what a legal push is by adding the following rule:

If the goal square of a push (the square the box should be pushed to) is a dead square, the move becomes illegal.

[7] algorithms for both deadlock detections taken from: `http://sokobano.de/wiki/index.php?title=How_to_detect_deadlocks` (visited: 18.04.23)

With the added rule the move generator would not return moves that are simple deadlocks, hence they are pruned from the search.

4.2.9.2 Freeze Deadlocks

Unlike simple deadlocks, freeze deadlocks cannot be detected before the search is started, but must be detected during the search. To detect freeze deadlocks in the best possible way, we use the following procedure:

When a box is pushed, the solver must check whether the pushed box is frozen. To do this, we need to check the horizontal and vertical axes to see if one of the following conditions is met:

1. Is there a wall on the left *or* on the right side?
2. Is there a simple deadlock square on the right *and* the left side?
3. Is there a box on the left *or* on the right side, which is already blocked?

We check these questions for each axis in turn. As soon as the first question from the above list can be answered with yes, the box is blocked along the axis just checked. If a box is blocked in both axes, the situation will be a freeze deadlock.

The 3rd check from the list must be implemented recursively. Boxes adjacent to the box just moved are then checked for a freeze deadlock according to the same principle as the box just moved. In order to avoid iterative endless loops, as they can occur for example in Figure 2.11, each box that has already been checked is marked as visited for a detection process. The iterative check then only examines adjacent boxes that have not yet been visited.

If a move and the resulting board position is detected as a freeze deadlock, the search is pruned at that point. A recognised position is stored in the deadlock table specific to freeze deadlocks. When the freeze deadlock detection is started, a check is made to see whether this position has occurred before and was classified as a deadlock. If so, the search is pruned directly without having to start the entire detection.

4.3 Solver Optimization Level Configurations

Our solver allows to be started with different configurations. Thus, it is possible to choose between the four search algorithms. In addition, the following enhancements can be selected as described in Table 4.1.
Each solver optimisation level described in Table 4.1 represents an improved variant of the solver. The solver variants build on each other, which means that a higher solver optimisation level already contains the improvements of the previous levels. In chapter 5 we compare the different optimisation levels with each other.

The different optimisation levels can be freely configured in the GUI.

Table 4.1: Possible solver configurations

	Algorithm used by solver			
Solver optimization level	**BFS**	**DIJKSTRA**	**A***	**IDA***
Level 0 (L0)	No Optimization			
Level 1 (L1)	Extended Move Generator			
Level 2 (L2)	/	/	L1 + Transposition Table	
Level 3 (L3)	L2 + Simple Deadlock Detection			
Level 4 (L4)	L3 + Freeze Deadlock Detection			
Level 5 (L5)	/	/	L4 + Bounded Relaxation	/

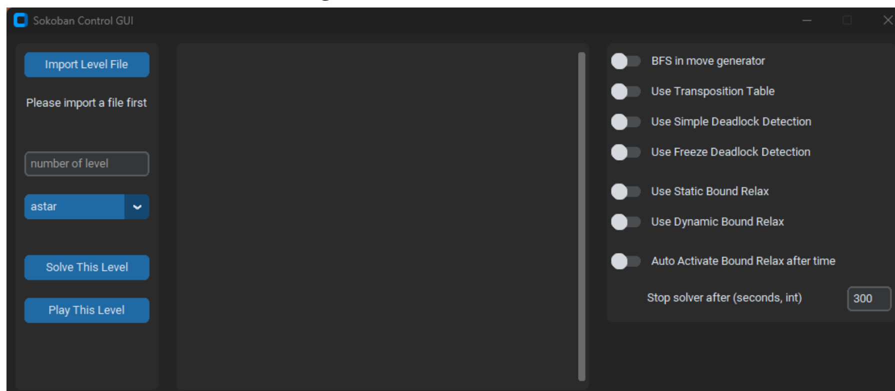
source: own creation

4.4 Graphical User Interface and Game Mode

To make it more simple for users to run the solver and freely try out the different solver configurations, we provide a GUI. The GUI helps to load a level, set a configuration and start the solver. It also offers to play the loaded level in a simple game environment.

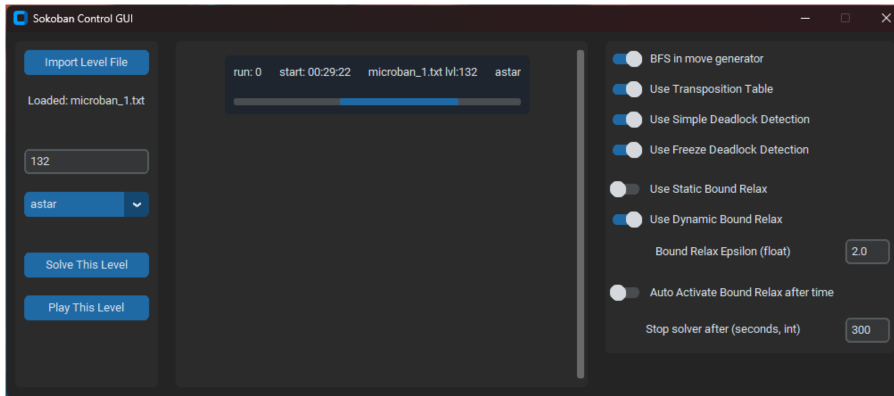
The GUI supports starting multiple solvers simultaneously with different configurations. For each started solver a new thread is created using the multi-threading library of Python. When a solver is finished with the search, the thread is terminated again and the results of the solver are sent to the GUI.
For systems without a graphic unit we offer a CLI variant of the GUI with which one can start and evaluate several solvers.

Figure 4.5: GUI Overview



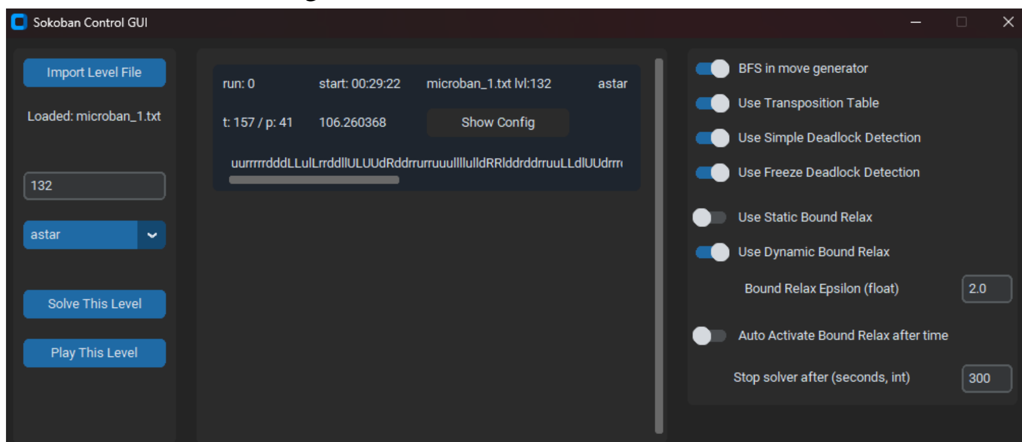
The GUI after it has been started
source: own creation

Figure 4.6: The GUI solves a level



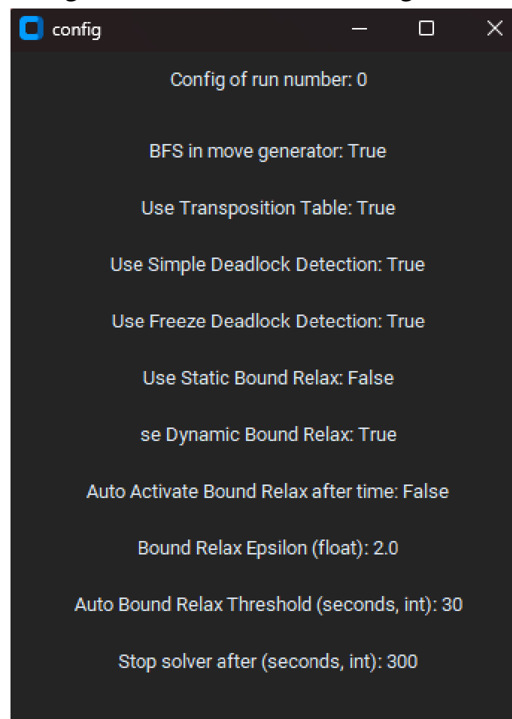
If a level file has been loaded and the search is started, a new entry appears in the results window in the middle. While the thread is running, a loading bar is showed.
source: own creation

Figure 4.7: GUI Level has been solved



After a level has been solved and the thread has been terminated, the GUI will show the results. Here t stand for the total steps and p for the pushes in the given *LURD*-String. Next to the push counter is the amount of time the solver needed to solve the level.
source: own creation

Figure 4.8: GUI solver configuration



When running multiple Solvers in the GUI it can be hard to remember which configuration was used for which thread. To overcome this problem a button *Show Config* is provided to display the configuration for that specific run.
source: own creation

Figure 4.9: Game Mode

Every imported level can also be played by moving the player with the arrow keys on the keyboard. If a game has been solved the player will be notified. The game mode uses the graphic sprites provided by JSoko.

source: own creation

5 Experimental results

5.1 Test Environment

Test Cluster:
The test cluster were the solver was tested on has 80 Intel(R) Xeon(R) Silver 4316 CPU @ 2.30GHz CPUs available where each CPU has 20 CPU cores and due to hyper threading 40 threads available. The cluster has a total amount of 528GB RAM available and is running Ubuntu 22.04.1 LTS x86-64 as operating system.
Each test run used one Intel(R) Xeon(R) Silver 4316 CPU instance and 64GB RAM.

Our local test systems, used for simple testing and development, have the following specs:

- **1. System:**
  CPU: Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz
  RAM: 32,0 GB @ 2667 MHz
  OS: Windows 11 Pro x86-64 (22621.1778)
  WSL 2.0: Ubuntu 22.04.2 LTS
- **2. System:**
  CPU: Intel(R) Core(TM) i7-8569U CPU @ 2.80GHz
  RAM: 16,0 GB @ 2133 MHz
  OS: MacOS Ventura 13.3.1 (a)

5.2 Test Set

In order to test our solver, we need several Sokoban levels with different levels of difficulty.
We provide two test sets containing 195 levels each. While the first test set includes easier level with less boxes, the other one includes larger dimensioned levels and levels with more boxes and more complex level structures.

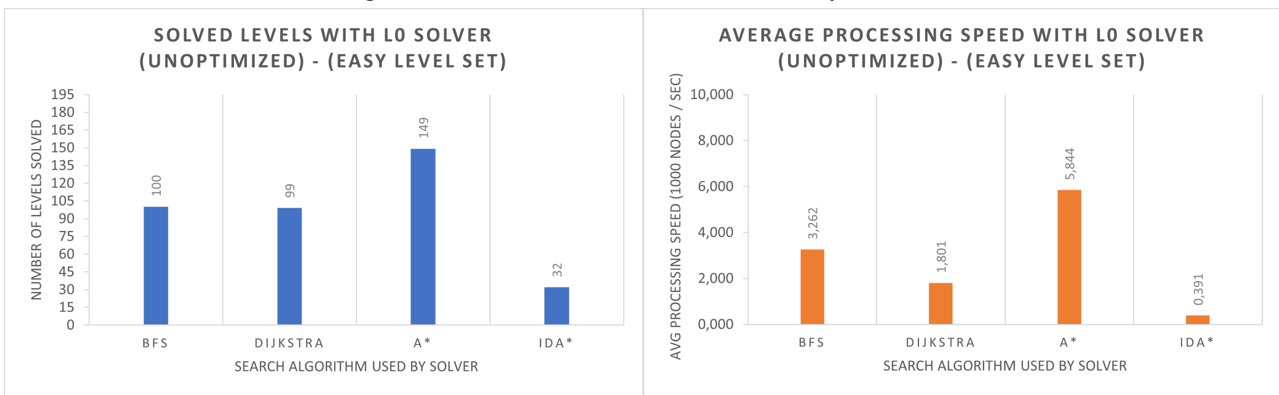
All the levels we use can be found at: `https://www.sourcecode.se/sokoban/levels.php` (visited: 13.06.23).
Our two level sets are composed from the Sokoban level libraries: *Microban I, Microban II, Microban III, Microban IV*. Both level sets were created by analysing all levels from the four libraries on our local test systems and then separating them into hard and easy levels. Easy and hard levels were distinguished by the time it took the non-optimised solver to solve the level.

Furthermore, we use Froleyks' level set (GroupEffort Level Set) [10] later on.

5.3 Comparing search algorithms

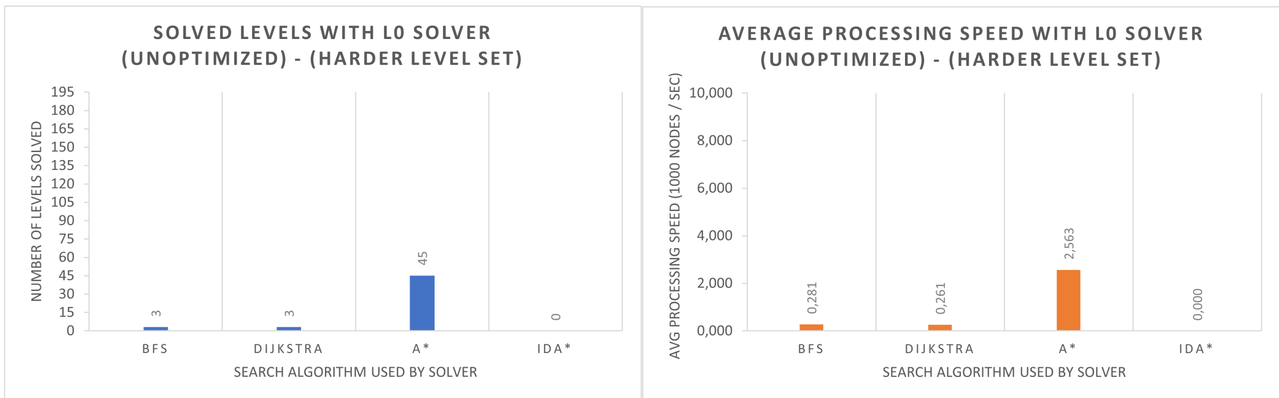
We first compare the individual four solvers in their basic form, meaning no optimisations have been made to the search algorithms and they were implemented as described in chapter 2.3. We will do this comparison for the easy level set and the harder level set. Unless otherwise stated, the solvers for each level are given 180 seconds to solve each level. If the solver needs more than 180 seconds to solve the level, it is considered unsolved. Unless explicitly stated, A* uses the Manhattan distance and each result was calculated on the test cluster.

Figure 5.1: L0 Solver tested on the easy level set



source: data from own implementation

Figure 5.2: L0 Solver tested on the hard level set



source: data from own implementation

Both Figure 5.1 and Figure 5.2 show the number of solved levels (blue) and the average processing speed (orange) for each algorithm of the respective level set in relation to the corresponding solver. The search algorithm used by the solver is indicated on the x-axis.

We calculate the processing speed of each search algorithm by measuring the number of nodes visited per time. We express the measurement in 1000 visited nodes per second.

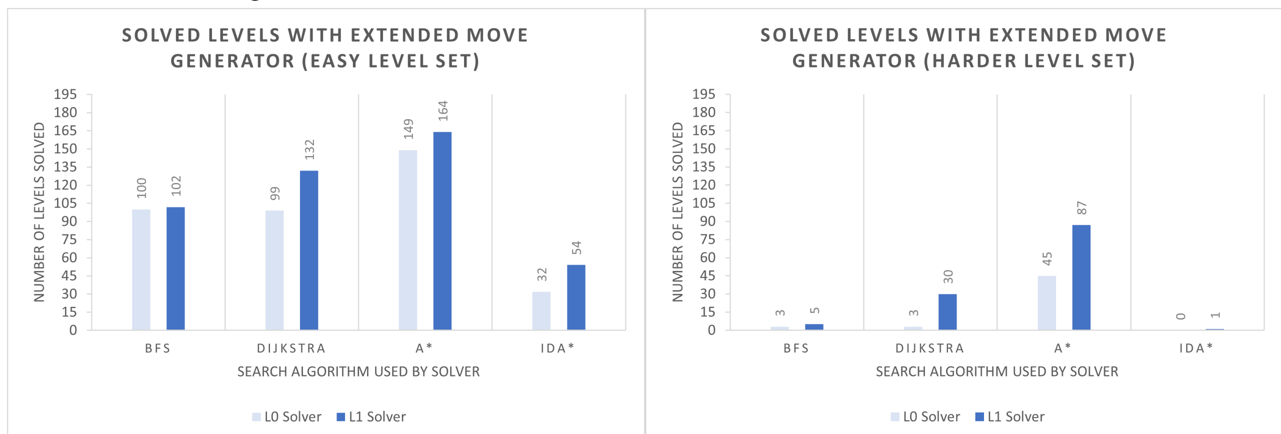
First, it can be seen that the four search algorithms can solve substantially more levels in the easier level set (Figure 5.1) than in the harder level set (Figure 5.2).
Looking at the processing speed, we see that A* has the highest processing speed, while BFS and Dijkstra visited about the same number of nodes per second in the hard level set. In the easier level set, Dijkstra visited slightly fewer nodes per second than BFS.

BFS and Dijkstra solved a similar number of levels for both sets of levels. A* solved by far the most levels for both level sets. IDA* has the lowest number of levels solved, and in the more difficult level set IDA* was not able to solve any level, so the processing speed was not calculated here.

5.4 Relaxing the move space

The Figure 5.3 shows the solver with optimisation level *L1* (coloured dark blue). The solver with optimisation level *L1* now uses the extended move generator instead of the classical move generator. Again, we have shown the results for both level sets. The results can be compared with the original solver *L0*, whose results are shown in light blue.

Figure 5.3: Results from L1 Solver with Extended Move Generator



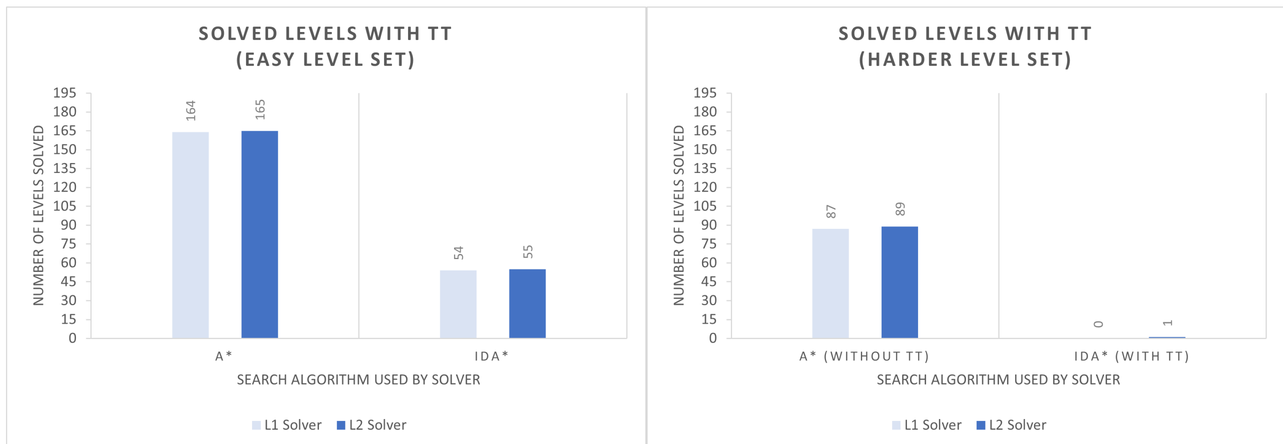
source: data from own implementation

Compared to the non-optimized solver *L0*, all graph search algorithms of the respective solver can now solve more levels. Dijkstra now solves more levels than the solver with BFS. Once again, A* comes out best in both level sets in terms of levels solved. IDA* has now managed to solve a level in the harder level set.

5.5 Speedup with a Transposition Table

This sections shows the results of the solver *L2*. Solver *L2* uses a Transposition Table as described in section 4.2.5. As the Transposition Table is only available for A* and IDA* based solver configurations the following graphics only compare these two search algorithms with the solver *L1* and the respective search algorithms.

Figure 5.4: Solved Levels of L2 Solver



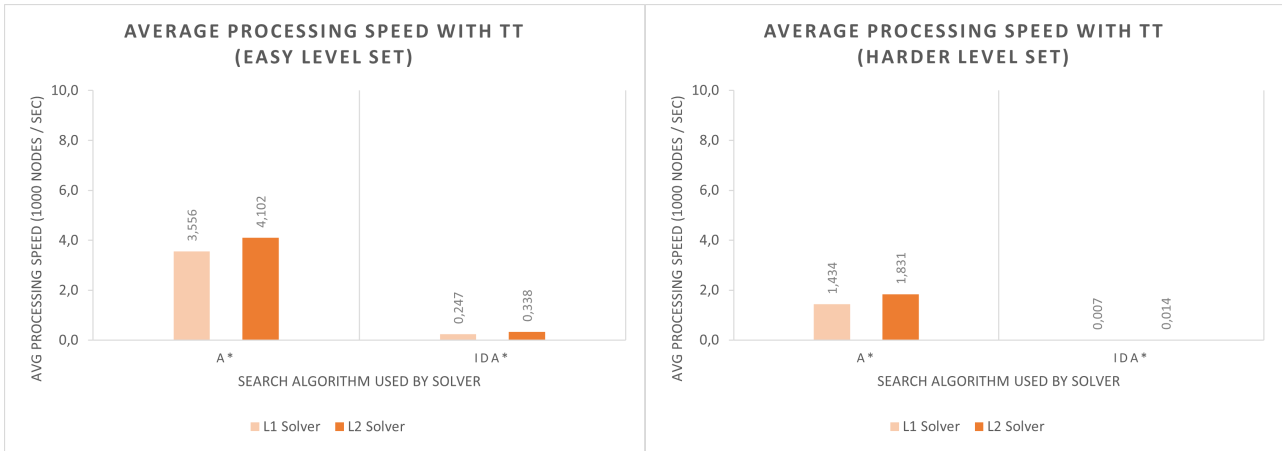
here TT stands for Transposition Table
source: data from own implementation

The graphic in Figure 5.4 shows the number of solved levels for both level sets. The results of solver *L2* are shown in dark blue, those of solver *L1* in light blue. The *L2* solver now solves one more level for both search algorithms, in the easy level set, than solver *L1*.
In the more difficult level set, A* solved two more levels with solver *L2* compared to solver *L1*, while IDA* solved one more level. Overall, therefore, the number of levels solved by each search algorithm is slightly increased.

Staying with the solver *L2*, we now compare the average processing speed with the corresponding results from solver *L1*.

The graph in Figure 5.5 shows the average processing speed of the two solvers *L1* and *L2* for the search algorithms A* and IDA*. The average processing speed is given as 1000 explored nodes per second. The value is slightly higher in the easy level set than in the harder level set for both algorithms. Compared to solver *L1*, the average processing speed for both algorithms has increased in solver *L2*.

Figure 5.5: Average processing speed of L2 Solver



here TT stands for Transposition Table
source: data from own implementation

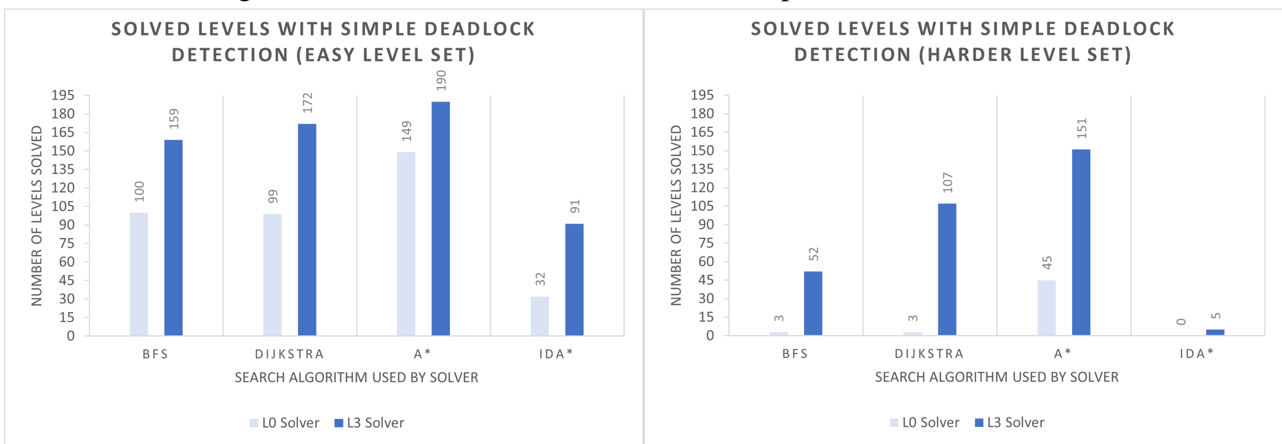
5.6 Efficient solving with domain knowledge enhancements

Since we can find deadlocks for Sokoban, as described in the chapters 2.9 and 2.11, and thus have the possibility to search the search space more efficiently, deadlock detection is one of the Sokoban-specific enhancements and thus belongs to the domain knowledge enhancements.

In the following we will look at the results of the solvers *L3* and *L4*.

5.6.1 Simple Deadlocks

Figure 5.6: Results from L3 Solver with Simple Deadlock Detection



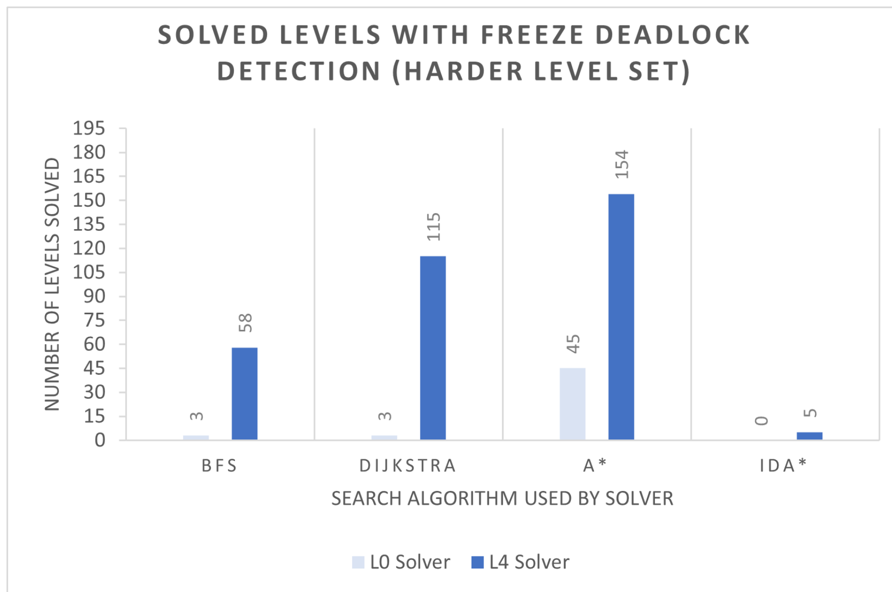
source: data from own implementation

The graphic in Figure 5.6 shows the solved levels of the solver *L3*, which uses the simple deadlock detection for all four search algorithms. The results of the current solver *L3* are shown in dark blue and the solver *L0* in light blue.
In comparison, more levels are solved with simple deadlock detection enabled for all search algorithms than without in solver *L0*. Furthermore, the solver variant with A* is the variant that solves the most levels.

Since the A* variant was able to solve almost all levels in the easy level set, we will only deal with the hard level set from now on.

5.6.2 Freeze Deadlocks

Figure 5.7: Results from L4 Solver with Freeze Deadlock Detection



source: data from own implementation

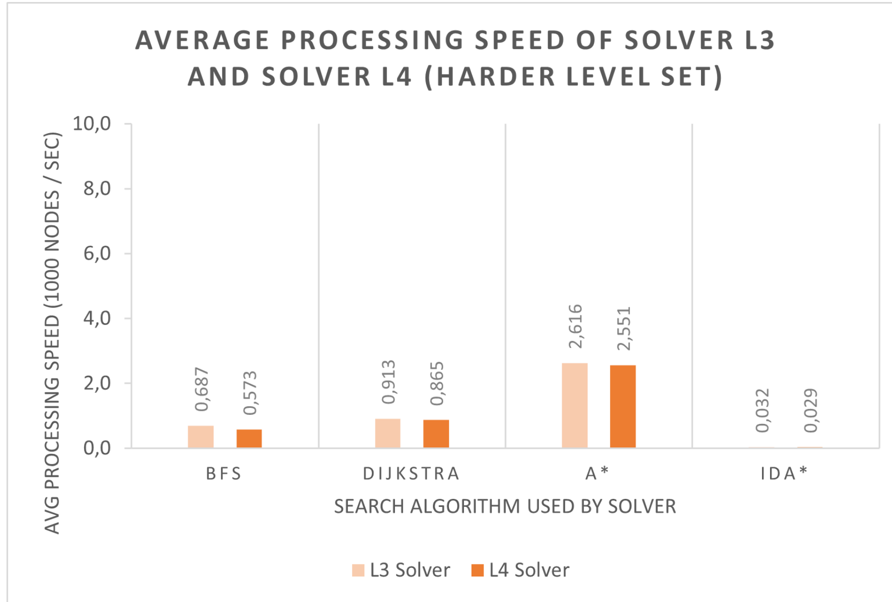
In the graphic in Figure 5.7 we see the number of solved levels of the *L4* solver (dark blue) with activated freeze deadlock detection for the harder level set. Compared to the corresponding values of the solver *L0* (light blue), solver *L4* again solves more levels.

A* still solves the most levels unchanged, followed by Dijkstra and finally BFS. Our IDA* implementation solves by far the fewest levels of all the results presented so far.

If we now compare the original solver variant *L0* without optimisations with *L4*, BFS, Dijkstra and A* solve substantially more levels. Dijkstra makes the greatest progress in *L4* with 112 more levels solved. A* solves 109 more levels in *L4* than in *L0*.

Compared to the results from Figure 5.6, the freeze deadlock detection shows an improvement in all search algorithms in terms of solved levels, except for IDA*, where the number of solved levels remains the same.

Figure 5.8: Processing speed comparison of Solver L3 and Solver L4



source: data from own implementation

In contrast to simple deadlock detection, freeze deadlock detection is executed during the search. It is therefore to be examined to what extent the detection has an influence on the processing speed.

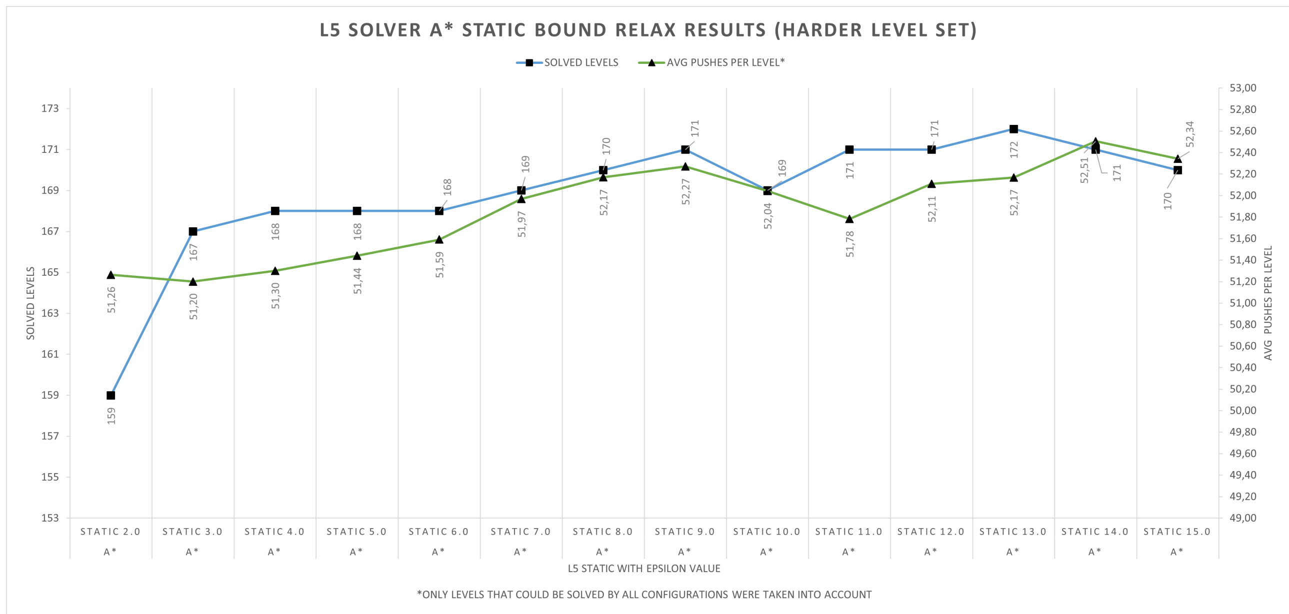
In the graphic Figure 5.8 we can see the average processing speed for solver *L3* and *L4*. It can be seen that all search algorithms have a lower average processing speed when using the freeze deadlocks than without.

5.7 Experiments with bound relax

We can use bound relaxation for A*, so we want to use solver *L5* to examine how bounded relaxation affects the number of solved levels. Since the heuristic used is no longer admissible due to the bounded relaxation, we also compare the solution length (the pushes needed per level). We can choose between weighted A* (static) or dynamic weighting A* and freely[1] select the parameter ε.
In the following, we measure the values, mentioned above, for static weighted A* and dynamic weighted A*. Thereby $\varepsilon \in \{2.0, 3.0, ..., 14.0, 15.0\}$, where each ε value represents a new measurement with the hard level set.

[1] values with $\varepsilon > 1$ are allowed per definition, see 2.3.6

Figure 5.9: Results from L5 Static Solver with different ε values

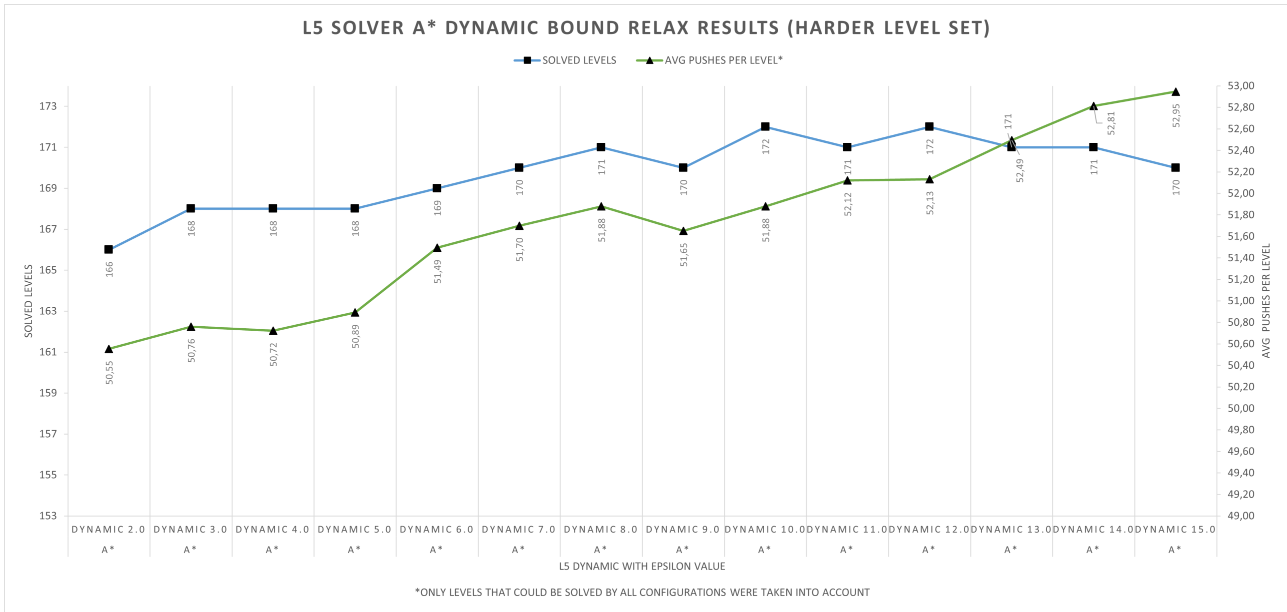


All *L5* static solver variants had a maximum time of 300 seconds per level. The number after the *Static* labels describe the used ε value.
source: data from own implementation

The graph in Figure 5.9 shows the results of the solver *L5* with static weighted A*. The number of solved levels is shown in blue and scaled with the left y-axis. The average number of pushes required per level is shown in green and scaled with the right y-axis. The x-axis shows the measurement per ε value. Both graphs interpolate between the measurement results. For the calculation of the average number of pushes required per level, only those levels were used that were solved by all solver configurations together. This leads to a more comparable mean value. The levels that are only solved by one configuration alone or not by all of them together were not taken into account for the calculation of the average.

In Figure 5.9 we first see that with a higher ε value the number of solved levels also increases. In some cases the number of solved levels decreases slightly. We get a peak with 172 solved levels at $\varepsilon = 13.0$. After the peak, the number of solved levels drops again as the ε value increases.
The average number of pushes per level also generally increases as the ε value increases. Even after the peak in solved levels at $\varepsilon = 13.0$, the average value increases.

Figure 5.10: Results from L5 Dynamic Solver with different ε values

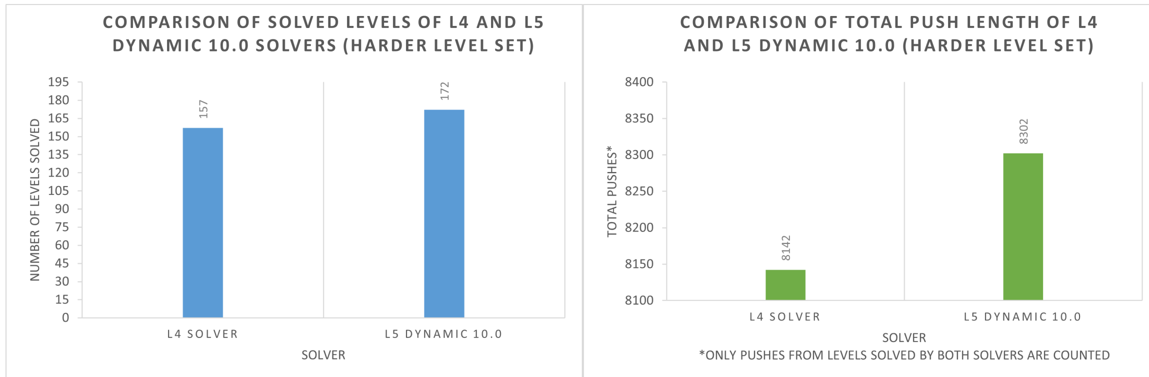


All *L5* dynamic solver variants had a maximum time of 300 seconds per level. The number after the *Dynamic* labels describe the used ε value.
source: data from own implementation

Next, in Figure 5.10 we look at the dynamic weighted A* variant with the different ε measurements. As in the static variant, we also see an increase in the solved levels with an increasing ε value in the dynamic variant. We see a first local high point at $\varepsilon = 10.0$. After that, the number of solved levels drops again and rises again to 172 at $\varepsilon = 12.0$ before dropping for all further measurements.
The average number of pushes required per level generally increases as the ε value increases. Even after the number of solved levels drops from $\varepsilon = 12.0$, the average value continues to increase.
Compared to the results from Figure 5.9, the dynamic variant also solves a maximum of 172 levels for the tested ε values. However, the dynamic variant already reaches this value with a lower ε value.

Since the solver *L4* still operates with an admissible heuristic, we now want to compare how the length of the *LURD* string, or more precisely the number of pushes, behaves when we use bounded relaxation in an *L5* solver variant.

Figure 5.11: Comparison between Solvers L4 and L5 Dynamic 10.0



Both solvers had a maximum time of 300 seconds per level. The search algorithm for both solvers is A*. The number after *Dynamic* describes the used ε value of *L5*.
source: data from own implementation

In Figure 5.11 we compare the number of solved levels and the accumulated number of pushes of the solved levels that the solver *L4* and the variant of the solver *L5* have both solved, i.e. the joint solved set of levels. With bounded relaxation enabled, substantially more levels can be solved when comparing the results with *L4*. The number of pushes for the jointly solved levels increases with higher bound relaxation value ε.

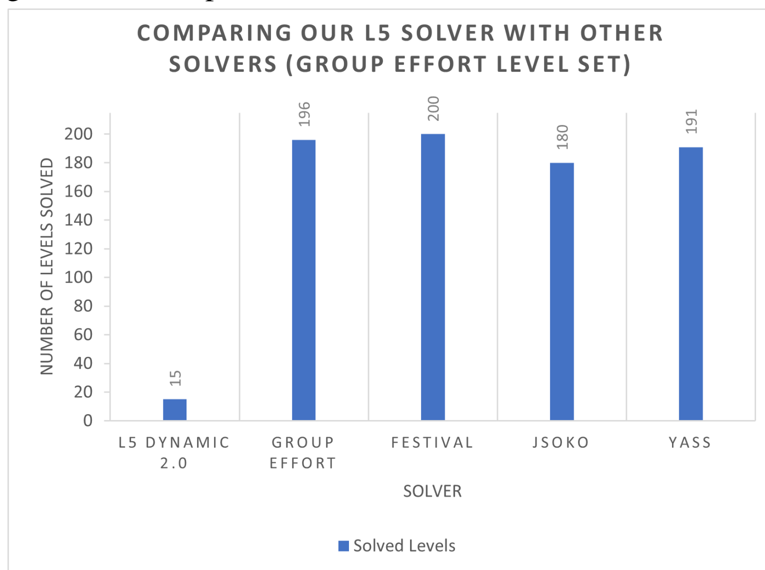
5.8 Comparison to existing solvers

To compare our solver with other existing solvers we use a level set first introduced by Froleyks, specifically we use the small level set called GroupEffort Test Suite (small test set) [10].

The small GroupEffort level set contains difficult levels that generally require a very good strategy to place boxes intelligently on an area of adjacent targets. The test set contains a total of 200 levels. [10].

In Figure 5.12 we can see at a first glance, that our *L5* solver variant solves substantially fewer levels with the new test set than all the other solvers. The Festival solver from [14] is the only solver that solves all 200 levels from the test set.

Figure 5.12: Comparison of a *L5* solver variant with other solvers



The *L5* solver variant had a maximum time of 300 seconds per level. The solver uses the dynamic bounded relaxation with $\varepsilon = 2.0$. The other solvers listed uses a computation time of 600 seconds per level, except Group Effort with 300 seconds per level. Since Group Effort uses an Algorithm Portfolio the solver uses the best configuration for each level. To determine the best configuration for each level, they ran all possible solver configurations in parallel and then selected the best result [10]. The other solvers have a higher calculation time per level. However, since all solvers solve the levels in less than 300 seconds, the higher time limit is irrelevant for the comparison.
source: data from own implementation and from `http://sokobano.de/wiki/index.php?title=Solver_Statistics` *(visited: 11.06.23)*

6 Discussion

In this chapter we will discuss and evaluate our results from chapter 5.

6.1 Search algorithm comparison

In Figure 5.1 and Figure 5.2 we first presented both level sets and compared them with the non-optimised solver *L0* for all four search algorithms.
The difference in difficulty between the easy and the harder level set becomes clear in the number of solved levels. We can therefore say that the harder level set is indeed more difficult for our solver to solve.
It is striking that Dijkstra can solve a similar number of levels as BFS in both level sets. We justify this with the fact that the move space is not yet relaxed and Dijsktra therefore has to search much deeper in the search tree, which makes the implementation only similarly fast as BFS.
A* can already solve substantially more levels than all other search algorithms from the beginning, which we justify with the informed search in connection with the Manhattan distance metric used by A*. Since a heuristic value is calculated for each node, A* automatically prefers moves that move the boxes closer to their targets due to the PriorityQueue, because these receive a better value according to the heuristic. In sum, the search space is searched more efficiently and in a more targeted manner, which leads to a faster result.
The results also show that our IDA* implementation presented in chapter 2.3.4 is not as good as it could be, because it can solve comparatively fewer levels than the other 3 search algorithms. Since we focused on A* in our work and only wanted to present IDA* as a memory-saving variant, the improvement of the IDA* implementation still offers room for future improvements. Furthermore, we did not experience any memory problems with our A* variant during the measurements. This means that the required memory space in the RAM never exceeded 2GB for the A* algorithm.

6.2 Move space relaxation

Through the relaxation of the state space with the extended move generator described in 4.2.3, we get a broader search tree and can thus go more quickly into the depths and search for solutions, because the entire move length to the boxes is now summarised in a move sequence.
This is also noticeable in Dijkstra, which can now achieve better results than BFS, in regard to Figure 5.3. Since BFS searches all nodes in the width of the search tree before it searches into a deeper level, BFS is only slightly faster here.
A* also benefits from the relaxed state space as it can avoid computing the shortest path of the player to a box and concentrate on moving the boxes to the targets. Since the Manhattan distance metric only changes by value when a box is moved, A* has no information from the heuristic whether the player's normal path to a box is the best one and thus spends time finding the shortest partial path.

The Extended Move Generator relieves A* of this task by directly providing A* with the shortest paths from the current player position to a movable box as move options.

6.3 Transposition Table speedup

Since we often get the same *minifiedBoardState* in the state space for different move sequences and have to calculate the heuristic value for it in A* and IDA*, the use of a Transposition Table seems to be useful here. With the Transposition Table we can save the recalculation of the heuristic value for a *minifiedBoardState* and take the stored value instead.
For this reason, we would expect savings in computation time because accessing the stored value in the Transposition Table should be faster than recalculating the heuristic value. As we save computing time, this is also visible in the average process speed for both search algorithms.
Given that this average actually decreases with the Transposition Table activated (see Figure 5.5) this confirms our assumption that accessing stored values in the Transposition Table is faster than recalculating the heuristic value. As the average processing speed is lower, the solver *L2* can also solve more levels, as can be seen in Figure 5.4.

6.4 Deadlock detection

With both variants of deadlock detection, we can detect a subset of the positions of a level that can no longer be solved and, if a deadlock is detected, prune the search in the search space at the current nodes. Since we prune the search tree, the search algorithms can reach nodes faster that actually still belong to solvable nodes. This makes the search more efficient.
Figure 5.6 and Figure 5.7 clearly show the effect of deadlock detection. Both solver variants *L3* and *L4* solve substantially more levels than without deadlock detection.
Since one of the major difficulties in Sokoban is the detection and avoidance of deadlocks, which was also recognised by Cazenave and Jouandea [15], precisely this detection is essential for fast and efficient Sokoban solvers. Thus, the larger the amount of deadlocks detected, the faster and more efficiently a solution can be found by the solver.

However, Figure 5.8 also shows that the average processing speed decreases with an active live deadlock detection. One should therefore carefully consider the computational effort required to detect deadlocks in relation to the actual benefit. Since our freeze deadlock detection resulted in more solved levels, the additional computing effort is compensated by the greater benefit.

6.5 Bound relax

From the results of chapter 5.7 we learned that with increasing ε both our bounded relaxation variants solve more levels (at least in most cases). As expected, however, the solution length also increases, i.e. the results found no longer have to be the shortest solutions, but can still be so although bounded relaxation is used.

We have only tested the values $\varepsilon \in \{2.0, 3.0, ..., 14.0, 15.0\}$ in our measurements. It therefore remains open whether the *L5* solver variants behave differently with other values. Furthermore, the results only apply specifically to the tested level set, since the respective *L5* solver can also behave differently for other levels.
A general statement that with higher ε more levels can be solved in less time cannot be made. Instead, the solver *L5* provides the possibility to find a good ε value to solve a maximum number of levels of a level set by trying different ε values for this set.

6.6 Results in comparison to existing work

As mentioned in chapter 5.8, the Group Effort Level Set contains the majority of levels whose goal squares are all adjacent. The solver must therefore arrange boxes intelligently in this goal area, while they are being transported to the goal, so that it can still transport later boxes to the goal area. Sometimes it is also necessary to move boxes temporarily away from the targets in order to be able to place the other boxes on the goal area and thus solve the level.
The other solvers use Goal Roam Macros introduced by Junghanns and Schaeffer [13] or other deadlock detection variants designed to place the boxes in the target area deadlock free as showed by Froleyks in [10].

Since our solver only recognises simple deadlocks and freeze deadlocks and the A* variant uses the Manhattan Distance as a metric, it is difficult for our solver to place the boxes ideally on a larger target area.
We analysed the unsolved levels in the Group Effort Level Set and saw that our solver moved all the boxes very close to their targets after a short time, but then spends a long time sorting and moving the boxes to the target area until the given time runs out.

To overcome this problem, a heuristic can be used that also considers the possibility of temporarily moving boxes, that have already been moved to a target, away from their targets like the Goal Roam Macros can do as described by Junghanns and Schaeffer [13].

7 Conclusion

We have shown in our work how to design an efficient Sokoban Solver. Furthermore, we have improved our Sokoban solver step by step and shown what effect each improvement has and explained how it was achieved. With our GUI we also provide the possibility to quickly test the solver with all optimisation levels for any Sokoban level. Since our solver was programmed in Python and Python is generally regarded as an easy-to-understand programming language, it is easy to extend the solver modularly with additional optimisation levels.

We have also shown which difficulties the solver still has despite optimisations and which possible solutions there are to alleviate these difficulties.

7.1 Constraints and limitations of our work

Our work addresses some Sokoban specific domain knowledge improvements, but mainly general applicable improvements. Our results have shown that our solver cannot solve certain, very hard, Sokoban levels quickly enough. Furthermore, we presented our results based on the test set we created, which means that the solver may show different behaviour for other test sets. This is especially relevant for the topic around the bounded relax variants of the *L5* solver.

Another limitation is the use of Python. Since Python is generally much slower compared to languages like C, we also get lower computational speeds for our solver.

7.2 Future improvements and application areas

The pruning techniques and improvements shown in our work can not only be used to solve Sokoban levels more efficiently, but can also be applied to general problems from the fields of Artificial Intelligence and game theory.

For future improvements, our IDA* implementation offers plenty of room for improvement, a first step would be, for example, to implement the algorithm non-iteratively. Furthermore, the deadlock detection can be improved further by reliably detecting other types of deadlocks. Another future improvement point would be the use of other heuristics that optimise the placing of boxes on their targets, which can be used in conjunction with the Goal Roam Macros and, according to the literature presented, would have a high effect in solving specific Sokoban levels substantially faster.

Furthermore, our programmed solver in Python can also be translated into C to obtain significantly higher computing speeds. Since Python is an easy-to-understand language, the translation to C should not be conceptually difficult.

Since A* was the search algorithm that worked best for us in all solvers, it is also worth improving it. This would be possible, for example, by implementing a backward search, in which A* searches for a path from the target node to the start node, which can then be combined with the forward search.

The efficient solving of Sokoban levels remains an exciting research topic. Whether it is research into improved detection methods for deadlocks or the development of new domain-specific features such as those used in the FESS algorithm as for example the connectivity feature.

Bibliography

[1] D. Dor and U. Zwick, "Sokoban and other motion planning problems," *Computational Geometry*, vol. 13, no. 4, pp. 215–228, 1999, ISSN: 0925-7721. DOI: https://doi.org/10.1016/S0925-7721(99)00017-6. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925772199000176.

[2] J. C. Culberson, "Sokoban is pspace-complete," 1997. DOI: 10.7939/R3JM23K33.

[3] *Solving sokoban - master thesis*, https://weetu.net/Timo-Virkkala-Solving-Sokoban-Masters-Thesis.pdf, Accessed: 05.05.23.

[4] A. Bundy and L. Wallen, "Breadth-first search," in *Catalogue of Artificial Intelligence Tools*, A. Bundy and L. Wallen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1984, pp. 13–13, ISBN: 978-3-642-96868-6. DOI: 10.1007/978-3-642-96868-6_25. [Online]. Available: https://doi.org/10.1007/978-3-642-96868-6_25.

[5] J. Erickson, *Algorithms. Independently published.*, 2019. [Online]. Available: https://jeffe.cs.illinois.edu/teaching/algorithms/book/Algorithms-JeffE.pdf.

[6] M. Barbehenn, "A note on the complexity of dijkstra's algorithm for graphs with weighted vertices," *IEEE Transactions on Computers*, vol. 47, no. 2, pp. 263–, 1998. DOI: 10.1109/12.663776.

[7] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, Dec. 1959, ISSN: 0945-3245. DOI: 10.1007/BF01386390. [Online]. Available: https://doi.org/10.1007/BF01386390.

[8] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968. DOI: 10.1109/TSSC.1968.300136.

[9] B. Blankertz and V. Röhr, "Algorithmen und datenstrukturen," 2023, TU Berlin, VL: Algorithmen und Datenstrukturen, Special Field Neurotechnology.

[10] N. Froleyks, *Using an algorithm portfolio to solve sokoban*, https://baldur.iti.kit.edu/theses/SokobanPortfolio.pdf, Accessed: 07.05.23, 2016.

[11] A. L. Zobrist, "A new hashing method with application for game playing," *ICGA Journal*, vol. 13, pp. 69–73, 1990.

[12] A. Junghanns and J. Schaeffer, "Sokoban: Enhancing general single-agent search methods using domain knowledge," *Artificial Intelligence*, vol. 129, no. 1, pp. 219–251, 2001, ISSN: 0004-3702. DOI: https://doi.org/10.1016/S0004-3702(01)00109-6. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0004370201001096.

[13] A. Junghanns and J. Schaeffer, "Sokoban: Evaluating standard single-agent search techniques in the presence of deadlock," R. E. Mercer and E. Neufeld, Eds., pp. 1–15, 1998. DOI: 10.1007/3-540-64575-6_36.

[14] Y. Shoham and J. Schaeffer, "The fess algorithm: A feature based approach to single-agent search," pp. 96–103, 2020. DOI: `10.1109/CoG47356.2020.9231929`.

[15] T. Cazenave and N. Jouandeau, "Towards deadlock free Sokoban," Apr. 2010. [Online]. Available: `https://hal.science/hal-02311590`.