Bachelor thesis

Using an Algorithm Portfolio to Solve Sokoban

Nils Froleyks

Date: December 22, 2025

Supervisors: Prof. Dr. Peter Sanders
Dr. Tomáš Balyo

Institute of Theoretical Informatics, Algorithmics
Department of Informatics
Karlsruhe Institute of Technology

Abstract

The game of Sokoban is an interesting platform for algorithm research. It is hard for humans and computers alike. Even with its simple rules and small average level sizes there are levels that take a lot of computation for all known algorithms.

In this thesis we will combine different Sokoban solvers with different domain specific enhancements into one portfolio. This portfolio can then be run in parallel on one problem until one solver finds a solution. Additionally the solvers in the portfolio can exchange data to speed up computation.

We will validate the approach of algorithm portfolios for designing a parallel Sokoban solver.

Zusammenfassung

Das Computerspiel Sokoban ist ein interessantes Testgebiet, um algorithmische Konzepte zu erforschen. Es ist sowohl für Menschen als auch für Computer ein schwieriges Problem. Trotz seiner simplen Regeln und der geringen durchschnittlichen Levelgröße, sind einige Level von keinem bekannten Algorithmus in annehmbarer Zeit lösbar.

In dieser Arbeit werden wir unterschiedliche Lösungsalgorithmen für Sokoban in einem Algorithmen Portfolio vereinen. Dieses Portfolio kann dann parallel auf einem Problem ausgeführt werden, bis einer der Algorithmen eine Lösung findet. Zusätzlich können die Algorithmen im Portfolio untereinander kommunizieren, um die Berechnungen zu beschleunigen.

Wir werden zeigen, dass Algorithmen Portfolios ein möglicher Ansatz zum parallelen Lösen von Sokoban sind.

Acknowledgments

First of all I want to thank my supervisors Prof. Peter Sanders and Dr. Tomáš Balyo. With the latter I had a lot of interesting conversations about Sokoban and strategies to solve it.

I also want to thank the people who worked with me in the institute during the time of my bachelor thesis. They always provided a friendly working environment and advice when needed.

Last I want to thank Matthias Meger, the creator of JSoko, for his implementation of the game. Having something nice to look at, helps when thinking about problems. I also used JSoko to create some of my graphics.

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, den 22. Dezember 2025

Nils Froleyks

Contents

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Contribution . . . 1
- 1.3 Structure of the Thesis . . . 1

2 Preliminary 3
- 2.1 Graphs . . . 3
  - 2.1.1 Bipartite Graphs . . . 3
  - 2.1.2 Matching . . . 3
- 2.2 Sokoban . . . 4
  - 2.2.1 Rules . . . 4
  - 2.2.2 Difficulties . . . 6
  - 2.2.3 Search Spaces . . . 8
- 2.3 Algorithm Portfolio . . . 10
- 2.4 Search Algorithms . . . 11
  - 2.4.1 Uninformed Search . . . 11
  - 2.4.2 Informed Search . . . 13

3 Related Work 19

4 Our Solver: GroupEffort 23
- 4.1 Searching in Game Space . . . 23
- 4.2 Searching in State Space . . . 23
  - 4.2.1 Depth First Search. . . . 24
  - 4.2.2 Depth First Search with Move Ordering . . . 24
  - 4.2.3 A* . . . 24
  - 4.2.4 Complete Best First Search . . . 25
- 4.3 Transposition Table . . . 25
- 4.4 Heuristic . . . 25
  - 4.4.1 Distance Metric . . . 25
  - 4.4.2 Assignment Algorithm . . . 26
- 4.5 Deadlock Detection . . . 29
- 4.6 Shared Information . . . 31
- 4.7 Restarts . . . 31

5 Experimental Evaluation **33**
5.1 Implementation 33
5.2 Experimental Setup 33
5.2.1 Environment 33
5.2.2 Test Levels 34
5.3 Individual Solver 34
5.4 Diversity 36
5.5 Portfolio 39
5.6 Communication 41
5.7 Comparison to existing solvers 41

6 Conclusion **43**
6.1 Future Work 43

A Test Sets **45**
A.1 Large Test Set 45
A.2 Small Test Set 45

Bibliography **47**

1 Introduction

1.1 Motivation

The game of Sokoban is a complicated computational problem. It was first proven to be NP-hard [6] and then PSPACE-complete [4]. While the rules are simple, even small levels can require a lot of computation to be solved. To reduce the computation time, parallelization seems necessary.

Due to the uneven search space distribution it is hard to parallelize the exploration of a branch and bound tree. As an alternative to traditional parallelization approaches this work focuses on algorithm portfolios – a parallelization concept in which each processor solves the whole problem instance, using a different algorithm, random seed or other kind of diversification.

1.2 Contribution

We describe how a search based Sokoban solver can be structured and which algorithms can be used to realize each critical part. We implement a variety of those and construct a number of different solvers. We test the solvers against each other and combine them into a portfolio to evaluate its performance.

1.3 Structure of the Thesis

In section 2 we will present the game of Sokoban and introduce the notation and necessary definitions this thesis will be using. Additionally we will introduce the concept of algorithm portfolios and present a number of basic algorithms. In section 3 we will review the previous work on Sokoban solvers and the game itself. After that we will present our solver in section 4. In section 5 we will evaluate the results of our experiments and validate the portfolio approach to solving Sokoban. Finally, we will conclude this work in section 6 and give some ideas for future work on the topic.

2 Preliminary

2.1 Graphs

A graph $G = (V, E)$ is a tuple of vertices V and edges $E \subseteq V \times V$. Two general types of graphs can be distinguished: *directed* and *undirected* graphs. A graph is undirected if

$$\forall x, y \in V : (x, y) \in E \Rightarrow (y, x) \in E$$

and directed otherwise.
A graph is *weighted* if a cost function $c : E \to \mathbb{R}$ is defined.
A vertex y is called a *successor* of x if $(x, y) \in E$. The number of successors a vertex has is called its *degree*:

$$d(x) = |\{(x, y) \in E\}|$$

A finite list of distinctive vertices p is a *path* of length n, if for every vertex in the list there is an edge connecting it to the one following it:

$$p = v_0, ..., v_n \text{ with } \forall i < n : (v_i, v_{i+1}) \in E$$

2.1.1 Bipartite Graphs

A graph is *bipartite* if the vertices of the graph can be divided into two disjunct sets so that no edges connect two vertices in the same set:

$$A, B \subseteq V, A \cap B = \emptyset, A \cup B = V, \forall (x, y) \in E : x \in A, y \in B \vee y \in A, x \in B$$

2.1.2 Matching

A *matching* is a subset of pairwise non-adjacent edges, i.e. no two edges in the matching share a vertex. A vertex is called *matched* if it is the endpoint of one edge in the matching.
A matching M of a graph $G = (V, E)$ is perfect if $|M| = \frac{|V|}{2}$
If G is weighted, a perfect matching M is *minimal* if $c(M) := \sum\limits_{e \in M} c(e)$ is minimal among all perfect matchings.

2.2 Sokoban

Sokoban is a puzzle game that originated in Japan. It was invented by Hiroyuki Imabayashi, and published in 1982 by Thinking Rabbit, as a PC game.[1] The word Sokoban is Japanese for warehouse keeper.

Each level represents a warehouse, where boxes are randomly placed. A warehouse keeper has to push the boxes around the warehouse so that all boxes are on goals at the end of the game.

2.2.1 Rules

Each level consists of a two dimensional rectangular grid of squares that make up the "warehouse". The squares are indexed starting from the top left with $(0, 0)$. If a square

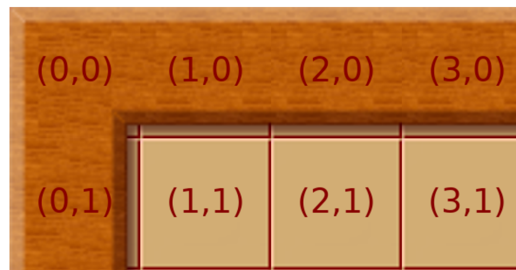
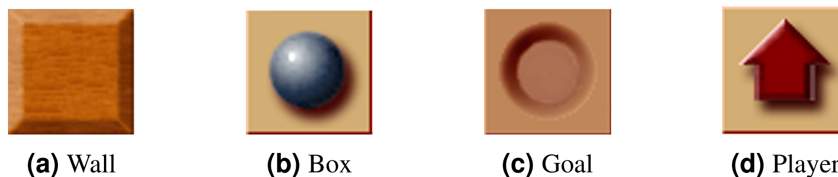


Figure 2.1: Level with the index of each square given. This graphic was created using JSoko `http://www.sokoban-online.de/jsoko.html`

.

contains nothing it is called a floor. Otherwise it is occupied by one of the following entities:



(a) Wall **(b)** Box **(c)** Goal **(d)** Player

Figure 2.2: Entities of Sokoban

(a) Walls make up the basic outline of each level. They cannot be moved and nothing else can be on a square occupied by a wall. A legal level is always surrounded by walls.

[1] `https://en.wikipedia.org/wiki/Sokoban` (10.11.2016)

(b) A box can either occupy a goal or an otherwise empty square. They can be moved in the four cardinal directions by *pushing*.

A *push* is defined by a *start position* and a *direction*. The start position is given by a tuple of indices (x, y).

The direction can be either up, $down$, $left$ or $right$.

As figured below let us suppose the direction to be $right$.

For a push to be legal the following conditions must be met:

- A box is at position (x, y).
- The player can reach the position $(x - 1, y)$.
- Position $(x + 1, y)$ is either floor or a goal.

After executing the push the box is located at position $(x + 1, y)$ and the player at position (x, y).

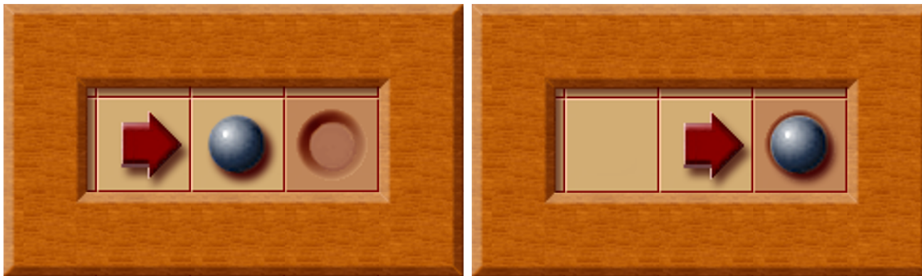


Figure 2.3: A push to the right

(c) Goals are treated like floors for the most part. Only when each goal is occupied by a box the game is completed. In a legal level the number of goals matches the number of boxes. For the sake of simplicity, we will call a square that is either a goal or a floor square *free* since the player and boxes can enter both.

(d) The player can execute *moves* to alter his position.

A *move* can be up, $down$, $left$ or $right$. A move is also a push if it alters the position of a box.

The player cannot move through walls or boxes. It can only move onto a square occupied by a box if it can execute a push to move it out of the way. Therefore the player cannot push more than one box at a time, nor can he pull them.

The goal of the game is to find a *solution*.

A *solution* is a sequence of moves. Executing a solution leads to every box being on a goal. It does not matter which box ends up on which goal.

A solution is commonly represented as a list of the letters: u, d, l, r. They are capitalized if the move was a push.

A level with solution is given in figure 2.4.

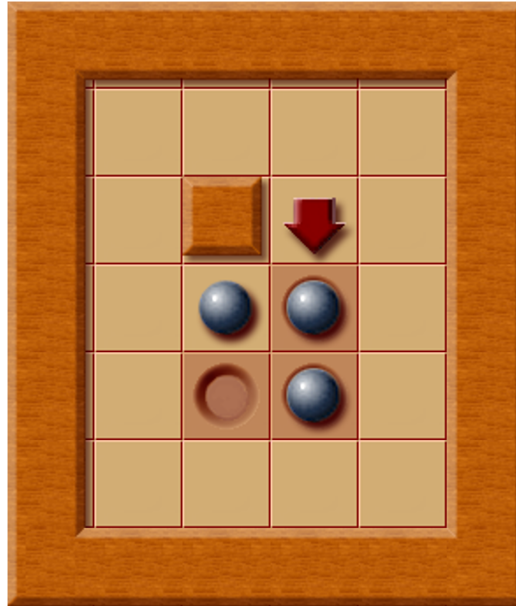


Figure 2.4: A possible solution string: $rddLruulDuullddR$

A solution can be scored either by the number of moves the player does or the number of pushes he executes to reach the goal state. In this work we will focus on finding any solution. See chapter 3 for more detail.

2.2.2 Difficulties

Sokoban is interesting for testing algorithmic concepts, since it is hard to solve. While the rules are simple there are levels that are small enough to be solved by humans[2] but all solvers that exist need a lot of computation. An example is given in figure 2.5.

The game has been proven to be PSPACE-complete (see section 3). Beside this there are a number of reasons why it is hard to solve a Sokoban level with conventional methods:

- The branching factor is large. It can exceed 350 for a level bound by 20x20 walls. See figure 2.6 for an example of a level with a high branching factor.
- The solution length may be very long. In some cases the optimal solution requires over 500 moves. [12]
- The existence of *deadlocks*.

Deadlocks

A game state is deadlocked if the level can no longer be solved. Every deadlocked game state contains a *deadlock*; a configuration of boxes and possibly the player that cannot be

[2] smaller than 20x20

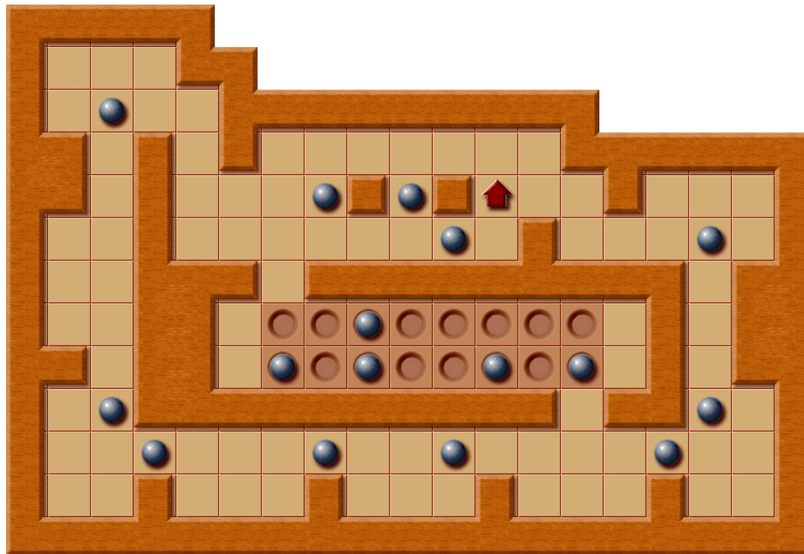


Figure 2.5: Level 29 of the XSokoban collection. For more information on Sokoban levels see section 5.2.2. All tested solvers (JSoko, YASS, Takaken more details in section 3) fail to solve it in under 10 minutes. See sokobano.de for more information.

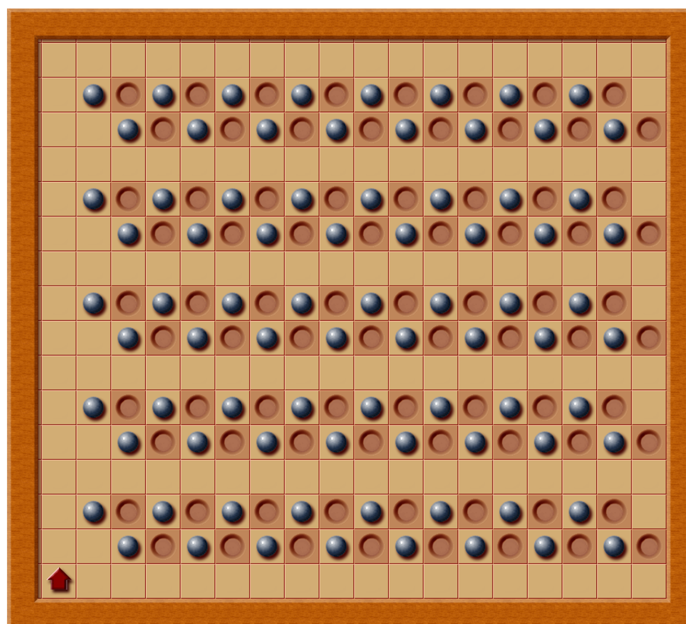


Figure 2.6: Level with a high branching factor

resolved. It is possible that every box is part of the deadlock, but most of the time only one or a few boxes are part of a deadlock. At least for the deadlocks that can be easily detected. Some simple deadlocks are presented in figure 2.7 and two more subtle ones in figure 2.8.

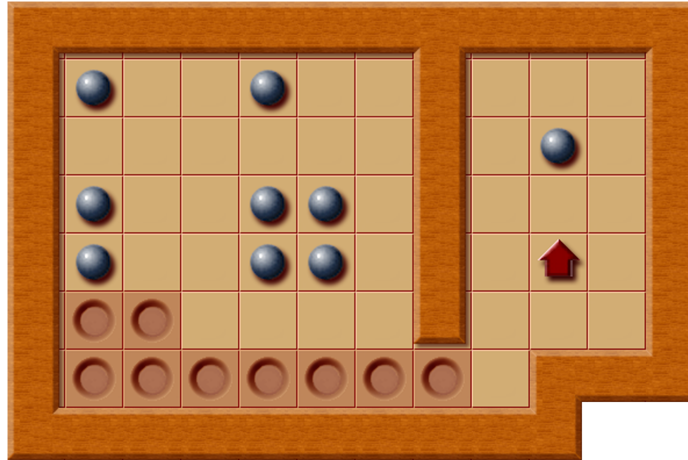


Figure 2.7: Examples of deadlocks. The box in the top left corner clearly cannot be pushed anywhere. The boxes in the four cluster in the middle prevent each other from being moved. The same goes for the two boxes along the left wall. The box at the upper wall can be moved but never reach a goal. The same applies to the box in the room on the right-hand side. [19]

2.2.3 Search Spaces

For Sokoban we have to consider two different search spaces. The *game space* and the *state space*. In the following both will be conceptualized as a graph.

Game Space. The game space is the actual maze. Vertices are made up of free squares and two vertices are connected by an edge iff the player can switch positions from one to the other in one move. Since the player can always move back the graph is undirected. Searching it is useful to find a path for the player or a single box. Note that the game space graph changes every time a box is moved. The game space graph for a specific level is given in figure 2.9.

State Space. The state of a Sokoban level is defined by the position of each box and the player. In section 4.2 we will define a relaxation of this definition.

Playing the game can be seen as transitioning from one state to another. Interpreted this way each level induces its own state graph, of which the vertices are the reachable states and an edge represents a legal transition from one state to another. The graph is directed since a transition can be irreversible. In figure 2.10 an example of a state graph is given.

Different move sequences can lead to the same state, therefore the graph may contain cycles.

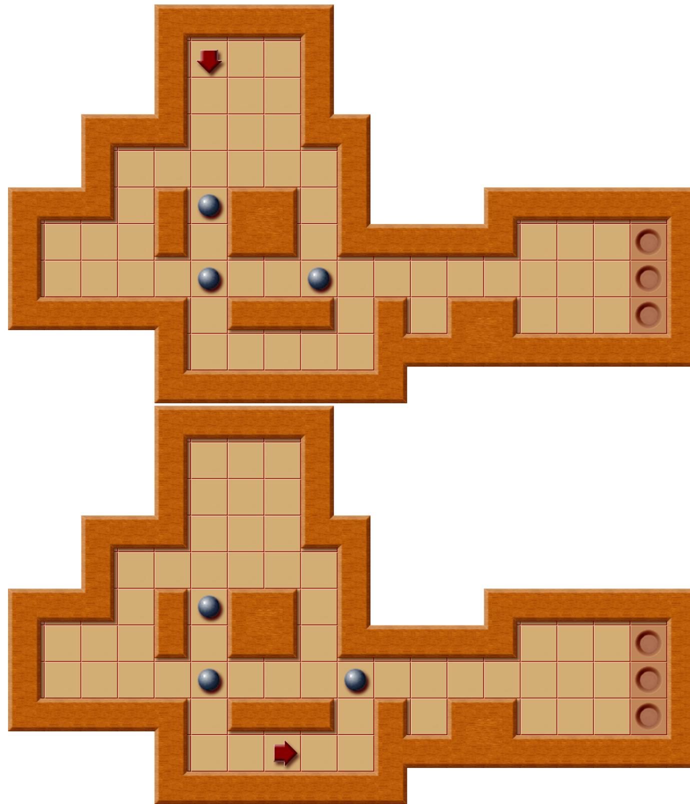


Figure 2.8: Two more subtle deadlocks. Every box and the player position are involved in both of them.

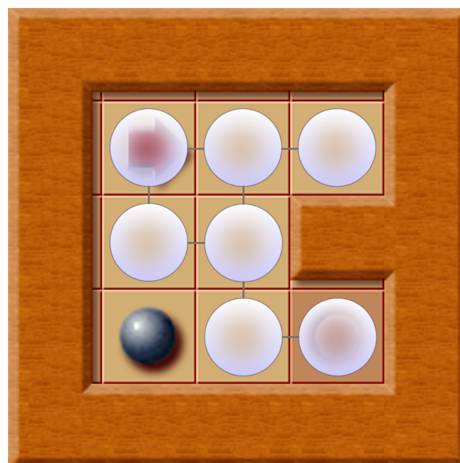


Figure 2.9: The game space graph for one state of a level.

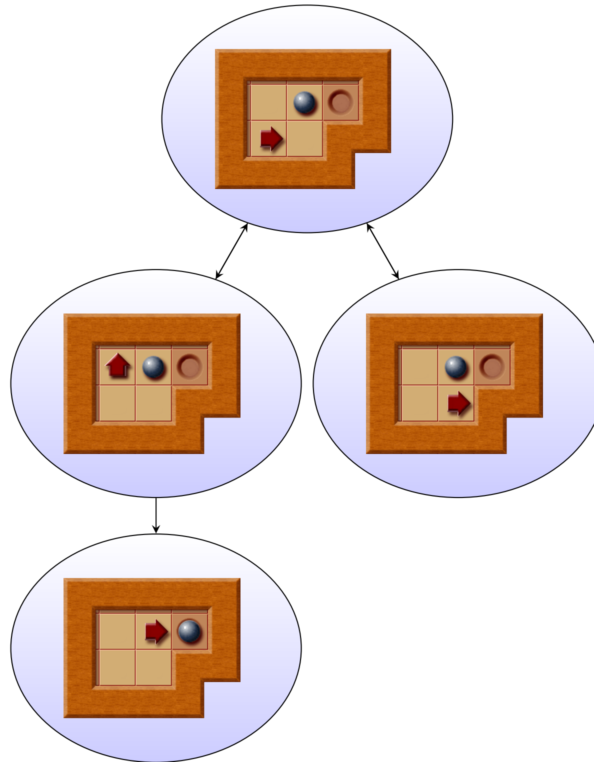


Figure 2.10: A partial state space graph. One edge is not bidirectional since the move cannot be reversed.

2.3 Algorithm Portfolio

For NP-hard search problems the run time of different algorithms can vary greatly from instance to instance. This property is also observed with randomized algorithms using different random seeds.

If the best algorithm configuration for each instance was known, only this one would be run. We call this configuration the *virtual best solver*.

While we cannot know this configuration beforehand, we can emulate the virtual best solver by combining our different algorithms into a *portfolio*.

This portfolio can then be run in parallel on multiple processors or interleaved on a single one. When one of the algorithms solves the problem all computation can be stopped.

To gain an advantage from running multiple algorithms we need to ensure that they are *diverse*. A portfolio is diverse if the algorithms will make different decisions, for search algorithms that leads them to explore different parts of the search space.

Since each algorithm in the portfolio solves the whole problem instance, communication between the processes is not necessary, except for sharing the termination condition.

Sharing information can however improve the run time. One possible example are different

conflict driven SAT solvers sharing clauses they learned. A clause is learned if it was not a part of the initial problem instance but was proven to be implied by the formula. [1]

There are advantages of this approach over exploring a branch and bound tree in parallel or other divide and conquer approaches. For example no time needs to be spend on load balancing and the communication does not halt the processes since the correctness of the results is not compromised by not receiving a message.

2.4 Search Algorithms

In general there are two different types of search algorithms. Tree search and general graph search algorithms. Since there are multiple paths to reach one vertex in the game space as well as in the state space[3] we will focus on graph search algorithms. For our purposes a graph search algorithm starts with a root vertex and will return a path to a solution vertex. For these there are two important attributes:

Completeness. An algorithm is complete if it is guaranteed to exhaust the search space and find a solution if there is one. Given that it does not run out of time or memory and the solution length as well as the branching factor (the number of successors) of each vertex is finite. If a part of the search space is *pruned* incorrectly by other means[4] the completeness of a search algorithm can be compromised.

If it is ensured that a part of the search space will not contain a solution vertex, it can be ignored. This is called *pruning*.

Optimality. An algorithm is considered optimal if it will always return the shortest path to a found solution.

2.4.1 Uninformed Search

Breadth First Search explores a graph, starting with the root vertex, by inserting all of its successors at the end of a queue and then continuing with the first one in the queue. Figure 2.11 gives an example of how BFS progresses through a graph.

This entails that all generated vertices that will be explored later need to be kept in memory. This set of vertices is also known as the *search frontier*. Assuming a constant branching factor of b and the current solution depth to be d, the size of the search frontier is in $O(b^d)$ [14]. For big search spaces where most of the vertices have a lot more than one successors this will lead to a memory problem. An implementation of BFS is given in algorithm 1.

[3] Even when using transposition tables as described in section 4.3.
[4] Deadlock detection as described in section 4.5 is one example of pruning.

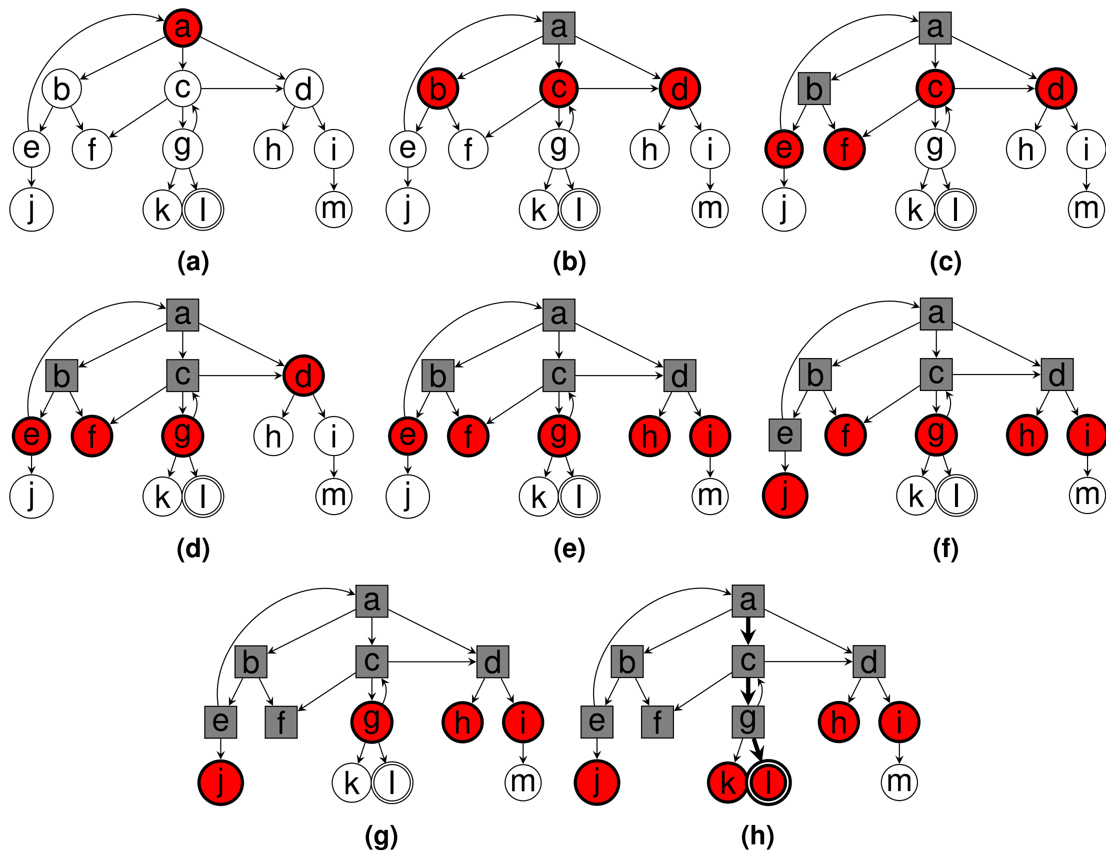


Figure 2.11: Breadth First Search. a is the root vertex and l the only solution vertex. White vertices are unexplored and grey vertices already have been explored. The red vertices are in the search frontier. In figure (h) the found path is marked.[19]

Depth First Search. To avoid the memory bottleneck depth first search (DFS) can be used. Instead of exploring all the vertices of a particular depth before moving on to a vertex with a higher depth, depth first search will always follow the successors of a vertex to their maximum depth, i.e. to a vertex that has no successor. When the search cannot progress further down it has to backtrack to the deepest vertex that still has unexplored successors. Figure 2.12 gives an example of how DFS progresses through a graph. An implementation of DFS is given in algorithm 1.

At any time depth first search only has to keep the vertices it explored along the current path in memory. This gives depth first search a memory consumption in $O(bd)$ with b being the branching factor and d the solution length again [14]. One major drawback of the depth first search is that it is not bound to find a solution in an infinite search spaces even if the maximum solution length is rather short. This also poses a problem if the search space is many orders of magnitudes larger than the solution length.

```
Input: rootVertex
Result: path to a solution vertex
begin
    visited ← ∅ // Set of visited vertices
    visited.add(rootVertex)
    vertices.push(rootVertex) // queue or stack with rootVertex as the only element
    while vertices ≠ ∅ do
        vertex = vertices.pop()
        foreach successor of vertex do
            if isSolution(successor) then
                return pathTo(successor)
            if not visited.contains(successor) then
                vertices.push(successor)
                visited.add(successor)
    return no solution found
```

Algorithm 1: BFS / DFS. The used data structure for *vertices* determines the implemented algorithm. If a queue (FIFO) is used this in an implementation of *BFS*. If a stack (LIFO) is used, *DFS* is implemented.

2.4.2 Informed Search

If a depth first search would always explore the right successor towards the solution vertex first, it would always find the optimal solution without backtracking. While this is not possible we can try to make a guess which of the successors is most likely to be part of the optimal path. For this a heuristic can be used.

A heuristic estimates the cost of the optimal path from any vertex to a solution vertex. When searching a route around obstacles in an euclidean space the Pythagorean theorem provides a commonly used heuristic. An important attribute of this heuristic is *admissibility*.

A heuristic is *admissible* if it never overestimates the distance to the closest solution vertex, i.e. the lowest possible cost of reaching a solution is never lower than the value of the heuristic.

Another import attribute of a heuristic is *monotonicity*. A heuristic h is *monotone* if

$$\forall(x, y) \in E : h(x) \leq d(x, y) + h(y).$$

A* is one example of informed search algorithms. It constructs a tree of partial paths starting from the root vertex until one of the paths ends in a solution vertex. The last vertex of each partial path is inserted into a priority queue ordered by the cost of the path to reach that vertex plus the remaining distance to the solution as estimated by the heuristic:

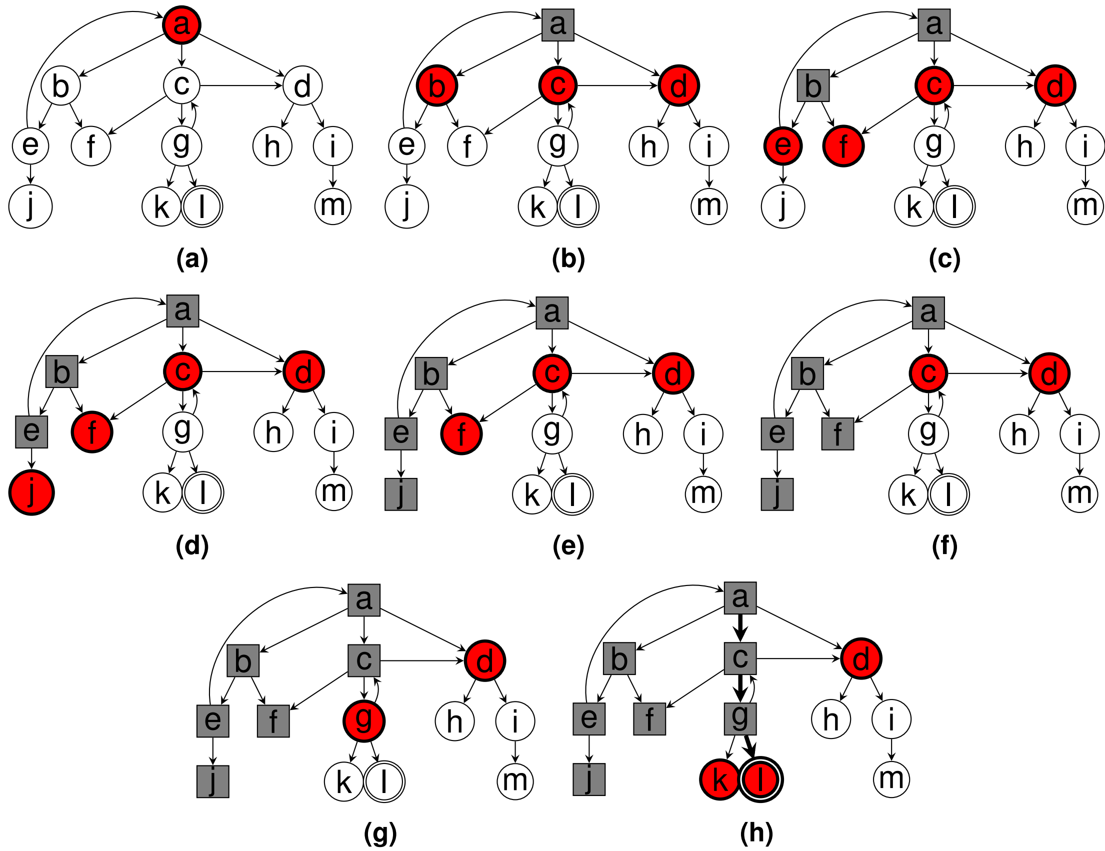


Figure 2.12: Depth First Search.

$$f(n) = g(n) + h(n)$$

Figure 2.13 gives an example of how A* progresses through a graph. In algorithm 2 an implementation is given.

If the heuristic used is monotone, each vertex has to be expanded at most once [10]. After evaluating all successors of a vertex and inserting them into the priority queue, the vertex can be marked as closed and has never to be reviewed again.

If the heuristic used is admissible A* is optimal [10].

```
Input: rootVertex
Result: path to a solution vertex
begin
    visited ← ∅ // Set of visited vertices
    visited.add(rootVertex)
    rootVertex.distance = 0
    priorityQueue.push(rootVertex, 0) // The vertices in the priority queue are
      ordered by the estimated total distance
    while priorityQueue ≠ ∅ do
        vertex = priorityQueue.pop() // returns the lowest cost node
        visited.add(vertex)
        foreach successor of vertex do
            if visited.contains(successor) then
                continue
            if isSolution (successor) then
                return pathTo (successor)
            totalDistance = vertex.distance + c(vertex, successor) + Heuristic
              (successor) // Heuristic estimates the distance to the closest
              solution distance
            if priorityQueue.contains(successor) then
                if successor.distance < vertex.distance +c(vertex, successor) then
                    successor.distance = vertex.distance + c(vertex, successor)
                    successor.predecessor = vertex
                    priorityQueue.decreaseKey(successor, totalDistance)
            else
                successor.distance = vertex.distance + c(vertex, successor)
                successor.predecessor = vertex
                priorityQueue.push(successor, totalDistance)
```

Algorithm 2: A* [10]

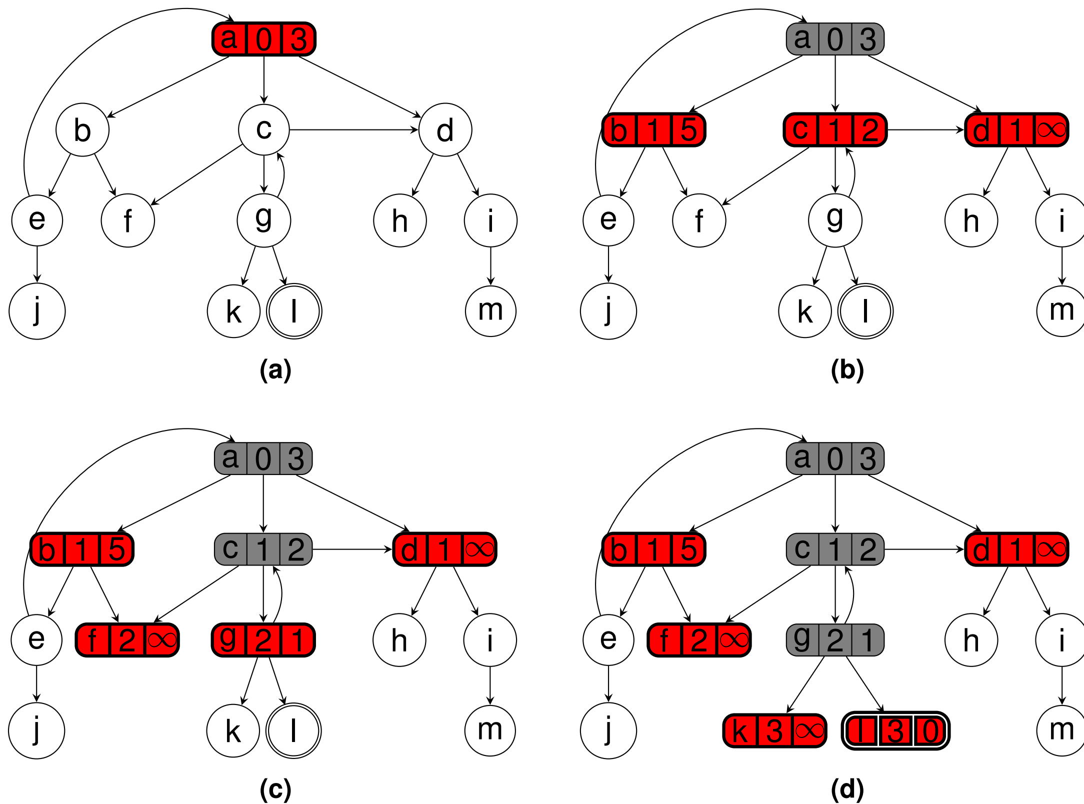


Figure 2.13: A*. The used heuristic is *perfect*. That means it will always return the exact length of a minimal path to the final node l. The second value of the explored vertices is the distance from the root vertex and the third value the remaining distance estimated by the heuristic.

Complete Best First Search is similar to A*. The difference is that the vertices in the priority queue are ordered only by the estimated remaining distance to the solution. An implementation is given in 3. It has to be noted that complete best first search is not optimal [10]. Figure 2.14 gives an example of how a complete best first search progresses through a graph.

```
Input: rootVertex
Result: path to a solution vertex
begin
    visited ← ∅ // Set of visited vertices
    visited.add(rootVertex)
    priorityQueue.push(rootVertex, 0) // The vertices in the priority queue are
      ordered by the estimated total distance
    while priorityQueue ≠ ∅ do
        vertex = priorityQueue.pop() // returns the lowest cost node
        visited.add(vertex)
        foreach successor of vertex do
            if visited.contains(successor) then
                continue
            if isSolution (successor) then
                return pathTo (successor)
            if not priorityQueue.contains(successor) then
                successor.predecessor = vertex
                priorityQueue.push(successor, totalDistance)
```

Algorithm 3: Complete Best First Search

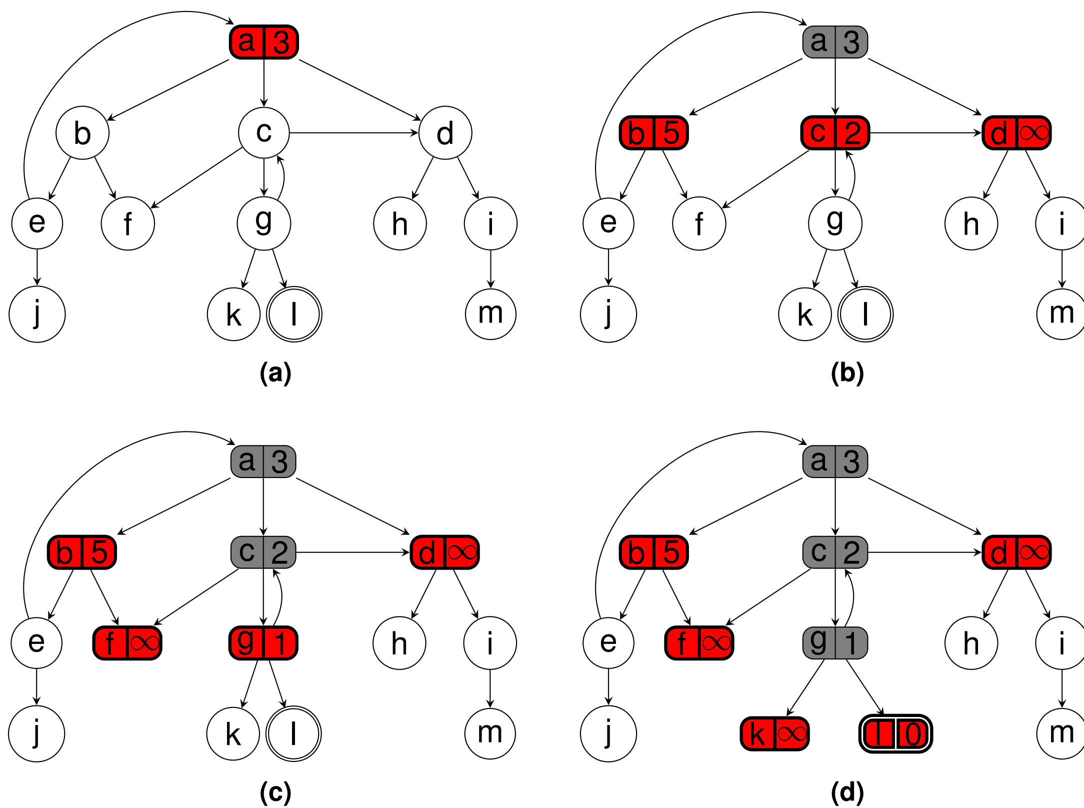


Figure 2.14: Complete Best First Search. Only the value of the heuristic is shown. Since we use the same optimal heuristic as in figure 2.13, the order in which the vertices are explored is the same. If the used heuristic is not perfect this is generally not the case.

3 Related Work

Complexity. Towards the end of the nineties some theoretical results on Sokoban were achieved. 1996 Dor and Zwick [6] presented $SOKOBAN^+$, a family of motion planning problems which are similar to Sokoban. They differ in the number of boxes the player can push, the shape of the boxes as well as the ability of the player to pull a number of boxes. They showed some members to be PSPACE-complete and for the original Sokoban they proved the NP-hardness and showed that the problem is in PSPACE. Dor and Zwick [6] stated the question whether the original Sokoban is PSPACE-complete, which was answered by Culberson [4] in 1997.

They showed how to reduce the word problem for the language L_{LBA} to Sokoban.

$$L_{LBA} = \{(\langle M \rangle, w) \mid M \text{ is an linear bound automaton that accepts } w\}$$

For a given instance I of the word problem, they presented a polynomial construction for a Sokoban level that has a solution if and only if $I \in L_{LBA}$.

For this construction they used a number of gadgets that relied on principles that are also known to the solving community, but since some of their gadgets alone exceed a size of 40×90, not much of the content of their work is applicable to solving Sokoban. An example of a simple gadget is given in figure 3.1.

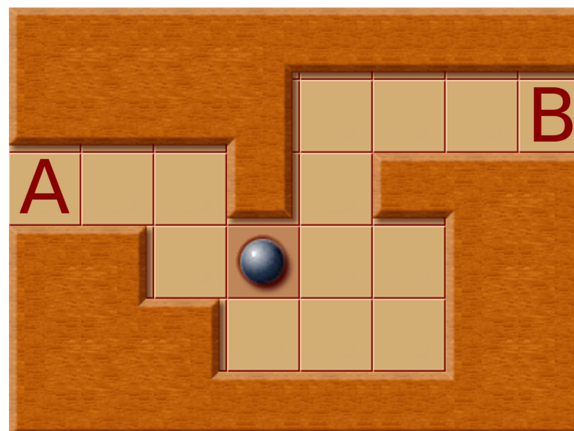


Figure 3.1: A simple one way gadget. The player can only pass it from A to B without leaving the box in an unrecoverable state.

If the constructed level has a solution, its length will be in $\Theta(|w| + t(|w|))$.

Where $t(|w|)$ is the number of transitions the linear bound automaton made on the input w. Since the word problem for L_{LBA} is PSPACE-hard [8] they have proven Sokoban to be PSPACE-complete.

Rolling Stone. The first published efforts to build a Sokoban specific solver were done by Junghanns and Schaeffer [11]. The Rolling Stone solver they proposed is considered to be a milestone. They kept developing it over a period of three years. The solver is based on iterative deepening A* and uses multiple domain independent search enhancements, such as transposition tables and move ordering.

To further reduce the size of the search space, they added domain dependent enhancements that preserved the optimality of the solution like deadlock tables, macro moves, the inertia heuristic and a lower bound estimation specific to Sokoban.

At this point, the solver was still only able to solve a fraction of the test set of Sokoban levels the developers chose. With the next features they added, the solution was no longer guaranteed to be optimal. These features included goal room macros, relevance cuts and a lower bound function that provided a better lower bound but was allowed to overestimate sometimes. With these features Rolling Stone was able to solve a significantly higher number of levels, but still not their whole test set.

Most solvers that were released since then implemented at least some of these enhancements and therefore did not guarantee an optimal solution as well. Most notable of these are *JSoko*[1], *YASS*[2] and *Takaken*[3].

Level deconstruction. Botea et al. [2] deemed heuristic searches to be of limited value in Sokoban and proposed a planning based solver. To make the planning approach viable they introduced *abstract Sokoban* as the planning formulation of the domain. In abstract Sokoban, each level consists of a small graph of rooms and tunnels connecting them. Solving a Sokoban level consists of two parts. The high level planning in abstract Sokoban and after that the translation of the abstract moves to actual box pushes and player movements.

Lishout subclass. Demaret et al. [5] introduced a solver that also used hierarchical planning. It decomposed a Sokoban problem not by dividing the level into rooms and tunnels, but by distinguishing the steps necessary to get a single box to its goal position.

First the solver figures out an order in which the goals should be filled and then an *extraction* for a single box is computed in which a number of pushes on different boxes may be executed. After the extraction follows the *storage* phase. During this phase only one box may be pushed. At the end of the sequence of moves the box will end up on its designated goal. Those two steps will be repeated for every box.

[1] `http://www.sokoban-online.de/jsoko.html`
[2] `https://sourceforge.net/projects/sokobanyasc/`
[3] `http://www.ic-net.or.jp/home/takaken/e/soko/index.html`

They described an interesting subclass of Sokoban problems. Every problem in this class can be solved instantaneously by their solver – essentially because the extraction phase is not necessary. A Sokoban level is in the subclass iff it satisfies the following three conditions [5]:

- It must be possible to determine in advance the order in which the goals will be filled.
- It must be possible to move a box to the first goal to be filled without having to modify the position of any other box.
- For each box, satisfying the previous condition, the problem obtained by removing that box and replacing the first goal by a wall must also belong to the subclass.

Deadlock free Sokoban. Cazenave and Jouandeau [3] followed another interesting approach. They generated a solution by searching the state space as well, but the main part of the computation was not spend there but rather in the preprocessing of the search space. Their goal was to never search in a branch that was already deadlocked, which most solvers will only recognize very deep in the search tree.

To achieve this, they built level specific deadlock tables before starting the search, using retrograde analysis. During the search they consulted the deadlock table for every possible move, reducing the size of the search space significantly. It has to be noted that they achieved this at the cost of spending a lot of time building the deadlock tables.

4 Our Solver: GroupEffort

The design of our solver *GroupEffort* follows the search approach used by Rolling Stone and later JSoko and others. Since we use an algorithm portfolio each critical part of the solver is implemented using a multitude of different algorithms. When executed, GroupEffort will assemble a number of solvers using these parts. This is done in order to diversify the portfolio of solvers.

4.1 Searching in Game Space

To find paths in the game space we use A* with a simple Manhattan distance heuristic.

Since the game space changes every time a push is executed, using Floyd-Warshall [7] or similar precomputations is not efficient.

4.2 Searching in State Space

To reduce the size of the search space we will first relax our definition of a state in Sokoban.

Two states are the same, if the box positions are identical and the player can move from the position it occupied in the one state to the position it occupies in the other state without moving a box. Figure 4.1 gives some examples of states with this definition.

An edge connecting a vertex to its successors is labeled with the push necessary to transition to the successor.

With this definition the maximum degree of a vertex is no longer four, i.e. the four directions the player could move, but instead the degree of a vertex is equivalent to the number of possible moves in the corresponding state.

Hence solving a Sokoban level essentially boils down to finding a path from the beginning state to a final state. A solution can be generated by collecting the pushes executed along this path and inserting the other necessary moves in between. The latter is done by searching a path in the game space from the last player position to the position that is required to execute the next push.

For finding a path in the state space we have implemented four algorithms with different characteristics.

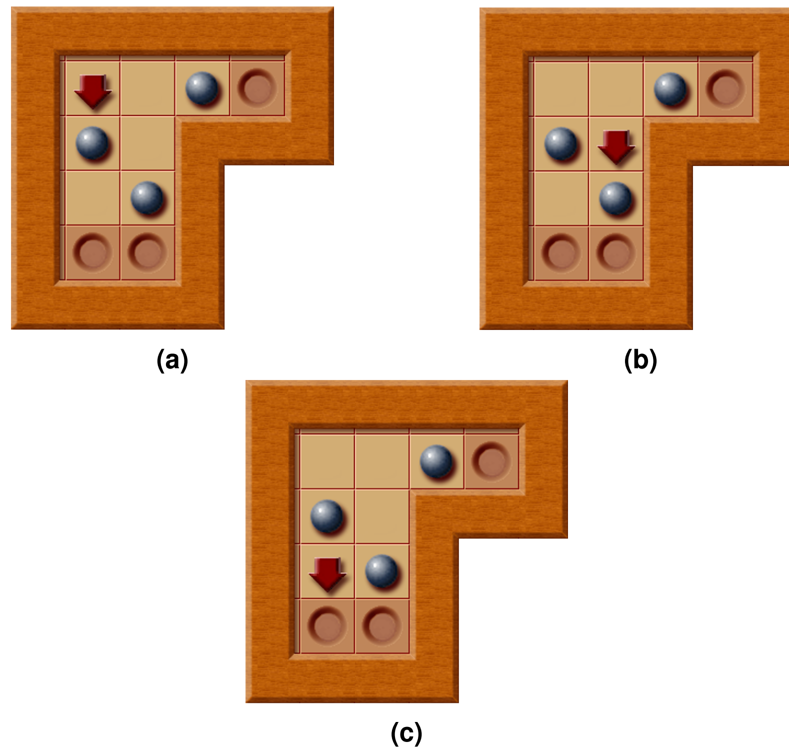


Figure 4.1: Figure (a) and (b) show the same state. The player can move from its position in (a) to its position in (b) without moving a box. Figure (c) depicts a different state.

4.2.1 Depth First Search.

While for most levels a simple depth first search is not viable, if the search space is restricted enough by other means[1] and the goals are concentrated in one area, a "blind" search algorithm can be a good option. One example of a level this search algorithm performs particularly well on is given in figure 4.4.

4.2.2 Depth First Search with Move Ordering

In contrast to the previous algorithm this one does not pick a random move but uses a *heuristic* to estimate the gain of each possible move. It then executes the move with the highest estimated gain and continues with the depth first search.

4.2.3 A*

If used with the right heuristic function configuration A* is optimal. A* is also proven to be *optimaly efficient* [17]. That means that among all algorithms that start from the same

[1] One way to restrict the search space is deadlock detection, presented in Section 4.5.

root vertex and use the same heuristic, A* expands the minimal number of paths. This, however, is only true for optimal algorithms.

4.2.4 Complete Best First Search

While A* has to expand all vertices of a particular depth in order to prove that there is no path of this length, Complete Best First Search (CBFS) does not have this limitation. It will always expand the vertex with the lowest estimated distance to a final one.

4.3 Transposition Table

In the state space graph, multiple paths can lead to the same vertex. Therefore we need a way to recognize already explored states to prevent unnecessary computation. This is done by the use of a transposition table. A common way to implement it is to use a hash table of states. For this we need a hash function that fits the definition of a state given at the beginning of section 4.2.

Our hash function is similar to the one described by Zobrist [20]. It takes the position of every box and the *normalized player position* into account.

The normalized player position is the topmost and then leftmost position the player can reach. Using this position the hash function will hash the same box positions and different player positions to the same hash value if the player positions are connected by a legal player path. Thus the hash function satisfies the relaxed definition of a state.

4.4 Heuristic

We use a heuristic to estimate the minimum number of pushes necessary to solve a Sokoban level from a given state. This is done by assigning a goal to each box and then summing up the distance of each box to its goal.

4.4.1 Distance Metric

There are multiple different metrics to estimate the distance a box has to be pushed in order to get from square A to square B. In our case we are only ever interested in the distance from a square to a goal.

Manhattan. The Manhattan distance d between two squares (A_x, A_y) and (B_x, B_y) is defined as:

$$d_1((A_x, A_y), (B_x, B_y)) = |A_x - B_x| + |A_y - B_y|$$

Since a box can only be moved along the four cardinal directions and not diagonally, the Manhattan distance is admissible.

Figure 4.2: The Manhattan distance heuristic. The length along the blue path is the same as the length along the red path.

Pythagorean. The Pythagorean distance is defined similarly:

$$d_2((A_x, A_y), (B_x, B_y)) = \sqrt{(A_x - B_x)^2 + (A_y - B_y)^2}$$

Compared to the Manhattan distance two squares are closer together if they are diagonal of each other.

Goal Pull Distance. This metric requires some precomputation. We compute the distance a box has to be pushed from each square to a goal if no other boxes were present and the player could reach every part of the level. We do this by using a breadth first algorithm to "pull" a box from a goal, i.e. checking for each of the four cardinal directions whether a box placed on the square in that direction could be pushed onto the goal. These squares are then marked with their distance to the current goal and we continue by checking their adjacent squares. This is repeated for every goal. The pseudocode is given in algorithm 4.

It has to be noted, that the distance from some squares to a goal will be infinite since not every goal can be reached from every square. See figure 4.3 for an example.

4.4.2 Assignment Algorithm

In order to assign a goal to each box we first compute the distance of each box to each goal. We represent these distances as a weighted undirected bipartite graph $D = (V_D, E_D)$. The vertices consist of the boxes and the goals. The edges are weighted with the distance between them. See figure 4.3.

```
Result: distance from all positions to all goals
begin
    distanceToGoal [Goals][Positions] ← ∞
    foreach goal do
        distanceToGoal [goal ][goal.position] = 0
        queue.push(goal.position)// FIFO queue with the position of the goal as the
          only element
        while queue ≠ ∅ do
            position = queue.pop()
            foreach direction do
                boxPosition = position + direction
                playerPosition = position + 2 · direction
                if distanceToGoal [goal ][boxPosition ] = ∞ then
                    if not wallAtPosition (boxPosition) and
                    not wallAtPosition (playerPosition) then
                        distanceToGoal [goal ][boxPosition ] =
                        distanceToGoal [goal ][position ]+1
                        queue.push(boxPosition)
```

Algorithm 4: Goal Pull Distance

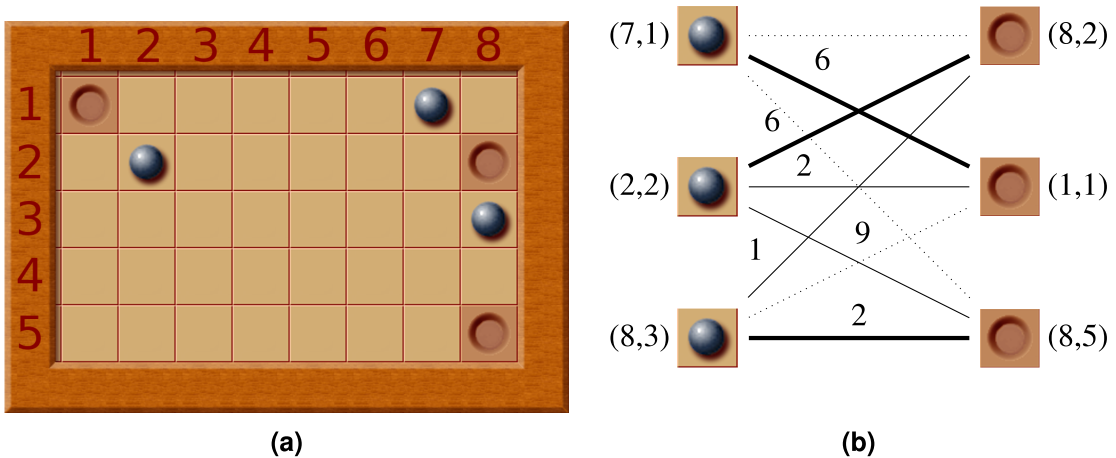


Figure 4.3: For calculating the distances we used the goal pull metric. The dotted lines have a weight of ∞. A possible assignment is marked. We determinded the lower bound $6 + 6 + 2 = 14$

Assigning goals to boxes can be seen as choosing a subset A of edges from E_D. Since each box has to end up on a goal every box should have an edge incident to it in A. We look for the minimal assignment to get a lower bound for the number of moves that are necessary to reach a final state. An assignment A is minimal if

$$c(A) = \sum_{e \in A} c(e)$$

is minimal among all assignments.

Closest Assignment. We can minimize $c(A)$ under these constrains by simply iterating over each box and assigning it to the goal closest to it. This is fast and simple but provides only an inaccurate lower bound. This is due to the fact that in the final solution only one box can occupy a goal. Taking this into consideration we have the following problem: Find a subset A of edges from E_D so that:

- Each box is assigned at most once.
- Each goal is assigned at most once.

This is equivalent to finding a matching in D. Since each box has to be assigned and the number of boxes equals the number of goals we have to find a perfect matching. Among all perfect matchings we are looking for the minimal one. Finding the minimal perfect matching in a given bipartite graph is called the *assignment problem*. In figure 4.3 a minimal perfect matching is given.

Hungarian Method. The assignment problem can be solved using the *Hungarian method*, which is in $\mathcal{O}(N^3)$, where N is the number of boxes to be assigned [15].

Greedy. Since this computation is expensive we can use a greedy heuristic to approximate a minimal perfect matching. We iterate over the list of all edges in ascending order by their cost. We maintain a list of all matched boxes and goals. If both endpoints of the current edge are not matched we add it to the matching. It is possible that not every box is assigned a goal; even if there is a valid matching. In that case we will assign a box to its closest goal. A simple possible implementation is given in algorithm 5.

```
Input: Edges
Result: Matching
begin
    priorityQueue ← Edges // The edges in the priority queue are ordered
        ascendingly by their weight
    matchedBoxes ← ∅ // Set of already matched boxes
    matchedGoals ← ∅
    while priorityQueue ≠ ∅ do
        (u,v) = priorityQueue.pop() // returns the edge with the lowest weight
        if not matchedBoxes.contains (u) and not matchedGoals.contains
          (v) then
            Matching.add ((u,v))
            matchedBoxes.add (u)
            matchedGoals.add (v)
    foreach u in Boxes do
        if not matchedBoxes.contains (u) then
            v = closestGoal (u)
            Matching.add (u,v)
    return Matching
```

Algorithm 5: greedy matching

4.5 Deadlock Detection

The existence of deadlocks is an essential part of Sokoban. Most states the player encounters while solving a level are only one or two moves away from becoming deadlocked. If a state is deadlocked the level can no longer be solved. However this does not mean that there are no possible moves left. The search through the state space would therefore continue and waste computation. For that reason it is important to detect as many deadlocks as possible and pruning the search as early as possible. Our algorithm has two ways to recognize deadlocked states. Either directly by the use of a *deadlock detector* or through the *recursive property*: If all states that can be reached from a state S are deadlocked, S is

deadlocked as well. Our deadlock detectors expand on the idea of *dead squares* presented by Junghanns and Schaeffer [11].

Simple Deadlock. A square in a level is called *dead* if a box placed on it can never be pushed to a goal. Placing a box on a dead square will result in a deadlock. Simple examples are corners that are no goals themselves. But there are also more subtle examples. In figure 4.4 all dead square of the level are marked. We only use static information, like the position of the walls and goals to compute these squares. We do this by "pulling" boxes from the goals, i.e. checking for each of the four cardinal directions whether a box placed on the square in that direction could be pushed onto the goal. Doing this for each goal, we get an area of the level from which a box can be pushed onto the current goal. If a square cannot be reached by that method it is a dead square.

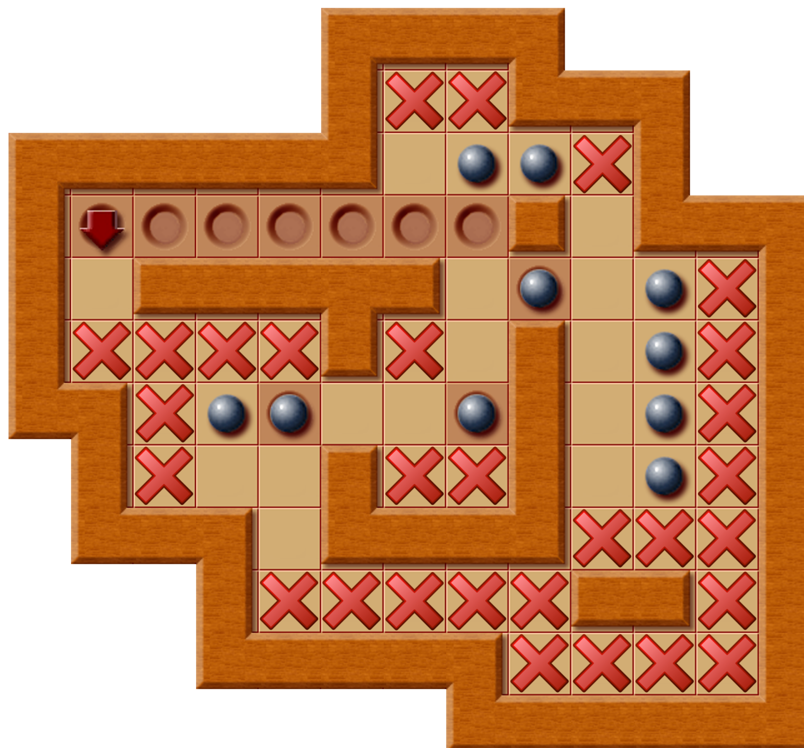


Figure 4.4: Level 1694 of the collection Sven. All dead squares are marked with a red X.

Count Area Deadlock. The first method does not take into account that only one box can occupy a goal at a time. If two boxes are in an area from which only one goal can be reached the game is also deadlocked. Even if no box is placed on a dead square. To detect this kind of deadlock we begin with the same kind of pulling algorithm as before and store which goals can be reached from each square. With this we then compute connected areas

of the level from which the same goals can be reached. We store the number of reachable goals as the maximum number of boxes allowed in each area.

During the search we need to check for every move if a box left an area. If this is the case we decrease the number of boxes in the old area and increase it in the new one. If the number of boxes in the new area surpasses the maximum the search can be pruned. This can be done in $\mathcal{O}(1)$ and we only need to keep the area number for each position and the number of boxes in each area in memory.

Since this deadlock detector outperforms the simple deadlock detection and the overhead is minimal, we always use *count area deadlock detection* in our experiments.

4.6 Shared Information

During the search a list of deadlocked states is maintained. When visiting a new state this list is checked. We do this similarly to the transition table described in section 4.3. If a match is found we can prune the search. This list can be shared periodically between the solvers of the portfolio. This sharing is optional since every solution that will be found with sharing, will also be found without it and vice versa. Enabling sharing just prevents unnecessary computation.

4.7 Restarts

Restarting the search means clearing the transposition table and starting the search again from the root vertex. However the list of deadlocked states and other static information is not cleared. Restarting the search can prevent it from getting stuck in one part of the search space after a bad branching decision. It is also useful after receiving new deadlocked states. Otherwise it is possible to explore the successor of a state that was already closed by another solver. Luby et al. [16] presented a universal restart strategy S^{univ} independent of the algorithm. A restart strategy is an infinite sequence of run times, an algorithm will be run for before restarting.

$$S^{univ} = (1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, 1, ...) = (t_1, t_2, ...)$$

$$t_i = \begin{cases} 2^{k-1} & \text{if } i = 2^k - 1 \\ t_{i-2^{k-1}+1} & \text{if } 2^{k-1} \leq i < 2^k - 1 \end{cases}$$

The sequence is plotted in figure 4.5. They have proven that the run time until a solution is found using S^{univ} is in $\mathcal{O}(T_o log(T_o))$, where T_o is the run time achieved with the optimal restart strategy. They have also shown that without further knowledge about the algorithm, their strategy is optimal up to a constant factor. We interpret the elements of S^{univ} as the number of states to explore before restarting. Since we do not want to actually restart after

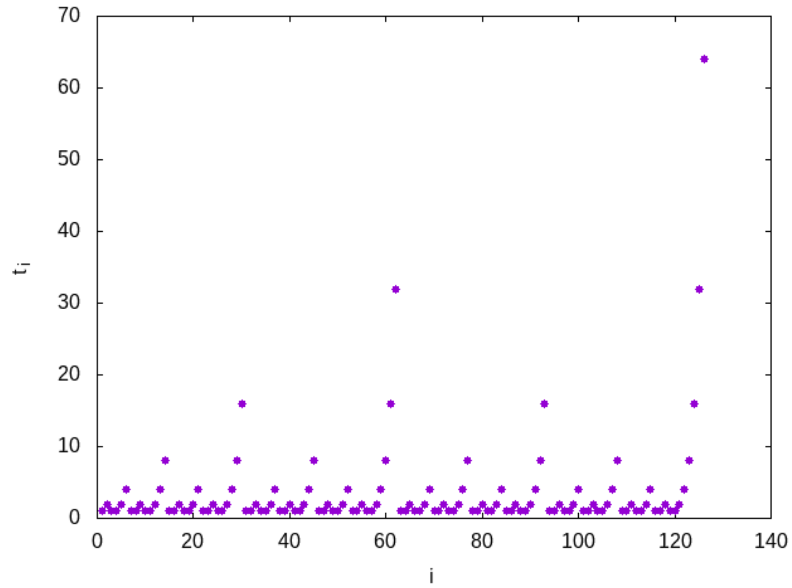


Figure 4.5: The luby-sequence.

exploring just one state, S^{univ} is scaled with a constant. Instead of computing the t_i with the definition above we use reluctant doubling as presented by Knuth [13]:

$$(u_1, v_1) = (1, 1)$$

$$(u_{n+1}, v_{n+1}) = u_n \& -u_n = v_n ? (u_n + 1, 1) : (u_n, 2v_n)$$

The syntax used is close to C. Therefore "$-$" is the unary minus and "&" is the bitwise AND.

The v_i computed this way are equivalent to the t_i defined by Luby et al. [16].

5 Experimental Evaluation

5.1 Implementation

Our algorithm is implemented using C++ and compiled using g++ v. 4.9.4 with full optimization flags turned on (-03 flag).

GroupEffort has two levels of parallelization: First we have two threads per core. One is running the solving algorithm and the other one manages communication with other solvers in the portfolio. The two threads use the shared memory model to communicate with each other. While the solving thread runs uninterruptedly until a solution is found, the communication thread will only run periodically and sleep in between for a fixed amount of time (usually around one second).

The second level of parallelization is running multiple instances of the program on different CPUs. They communicate using the Message Passing Interface (MPI). The used implementation is Open MPI[1] in version 1.6.5.

5.2 Experimental Setup

5.2.1 Environment

Each run of our solver is made on a varying number of cores of two Intel Xeon E5-2650 v2 processors. Each has 8 real cores with a maximum clock rate of 2.6 GHz. 128 GiB of DDR3 RAM are accessible. The PC is running Ubuntu 14.04 64-bit Edition.

The other solvers we test for comparison have to be run on a windows machine. It has two Intel Xeon X5355 2.66 GHz with only 4 cores each. Since we only use a single core at a time this will not pose a problem. No solver ever exhausted the available 24 GiB of DDR3 RAM. The PC is running Windows 2008 Server.

The difference in performance between the two machines can be neglected for what we want to show.

[1]`https://www.open-mpi.org/`

5.2.2 Test Levels

A lot of Sokoban levels have been published. A collection of close to 40.000 levels can be found at `www.sourcecode.se/sokoban/levels`. Most of the levels are made by hand but some are procedurally generated [18]. From those we select a number of level collections from different authors. After removing a few duplicates this test set, called *large test set* in the following, has a size of 2851 levels. See A.1 for a list of the used collections.

For other tests we use a *small test set* with a size of 200 levels. It is created from the large test set by first removing all levels that are easy. We deem a level to be easy if it is solved in under 3 seconds by all solvers. This is most likely to happen if the state space of the level is too small. After that all levels that are not solved by any solver are removed. From the remaining levels of each collection we select a fixed amount randomly. We make an except for the *Sven* collection. Since it is larger we select more levels from it. In section A.2 a list of the selected levels is given.

5.3 Individual Solver

GroupEffort can assemble different solvers by varying the following parameters:

- the search algorithm operating on the state space
- the distance metric
- the assignment algorithm
- how often the assignment of boxes to goals is recalculated

To test the solver configurations we run a subset of all possible configurations independently on the large test set. This subset contains all combinations of search algorithm, distance metric and assignment algorithm. The timeout for each level is 300 seconds. The results of this test are given in table 5.1.

Plot 5.1 gives a rough overview of the performance of each solver over time but it is not too useful to assess the gain of combining the solvers into a portfolio, since it does not show which levels are solved. If the best solver configuration achieves the best run time for each level, we gain nothing by using a portfolio.

solver number	search algorithm	distance metric	assignment algorithm	assignment recalculation	levels solved
1	CBFS	Pull	Hungarian	Once	1566
2	CBFS	Manhattan	Hungarian	Once	1561
3	CBFS	Manhattan	Greedy	High	1533
4	CBFS	Pythagorean	Hungarian	Once	1516
5	A*	Pythagorean	Greedy	High	1499
6	A*	Pythagorean	Hungarian	Once	1489
7	A*	Pull	Hungarian	Once	1488
8	A*	Manhattan	Greedy	High	1469
9	CBFS	Pythagorean	Greedy	High	1457
10	CBFS	Manhattan	Closest	High	1445
11	A*	Manhattan	Hungarian	Once	1429
12	CBFS	Pull	Closest	High	1390
13	CBFS	Pythagorean	Closest	High	1384
14	CBFS	Pull	Closest	Once	1358
15	CBFS	Manhattan	Closest	Once	1334
16	Directed DFS	Pull	Hungarian	Once	1308
17	Directed DFS	Manhattan	Hungarian	Once	1297
18	Directed DFS	Pythagorean	Hungarian	Once	1294
19	DFS	–	–	–	1290
20	Directed DFS	Pull	Closest	Once	1274
21	Directed DFS	Pull	Closest	High	1273
22	Directed DFS	Manhattan	Closest	High	1271
23	Directed DFS	Manhattan	Closest	Once	1271
24	CBFS	Pythagorean	Closest	Once	1263
25	Directed DFS	Pythagorean	Closest	High	1261
26	Directed DFS	Pythagorean	Closest	Once	1259
27	A*	Pythagorean	Closest	High	1257
28	Directed DFS	Pull	Greedy	High	1256
29	A*	Pull	Closest	High	1222
30	A*	Manhattan	Closest	High	1214
31	A*	Pythagorean	Closest	Once	1203
32	A*	Pull	Closest	Once	1193
33	A*	Manhattan	Closest	Once	1182
34	Directed DFS	Pythagorean	Greedy	High	1106
35	Directed DFS	Manhattan	Greedy	High	1100
36	CBFS	Pull	Greedy	High	992
37	A*	Pull	Greedy	High	966

Table 5.1: The table shows the total number of levels solved in under 300 seconds by each configuration. The solver numbers will be used in the following to identify different configurations. The configurations are sorted by the total number of levels solved.

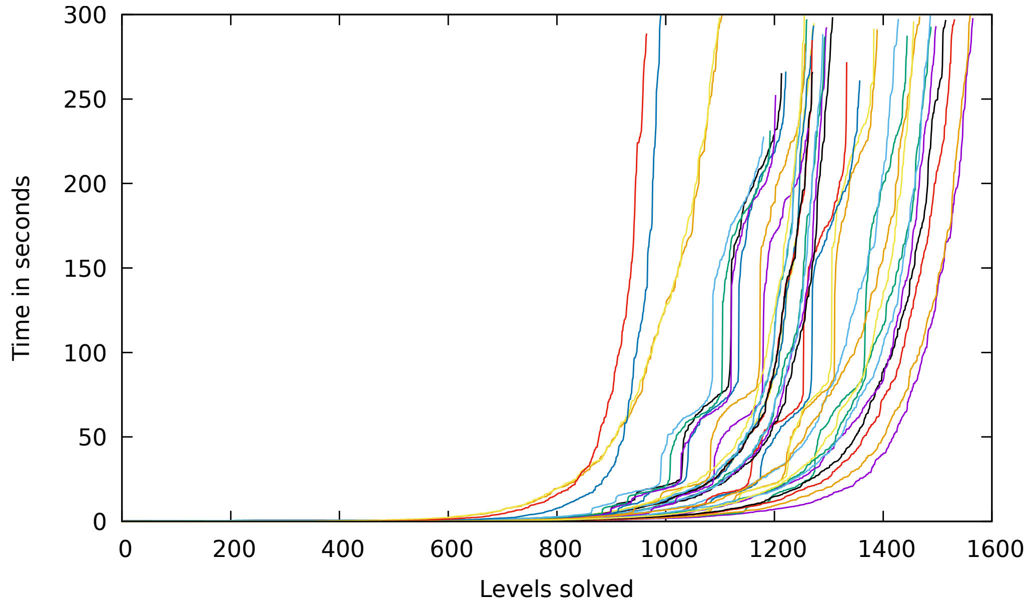


Figure 5.1: The plot shows how many levels each solver configuration can solve over time. All solvers show the same behavior: A high number of levels is solved in a very short time. After that, with an increase in computation time, not that many more levels are solved.

5.4 Diversity

We want our solver configurations to be different from each other, i.e. they should make different decisions while searching for a solution and therefore solve different levels. To analyze this we calculate how many levels each solver solves *significantly better* than another one.

A level is solved significantly better by solver A than solver B if A solves it at least 30 seconds (10% of the maximum run time) faster or B does not succeed at all.

Some of these relations are depicted in figure 5.2. While the general trend is that the overall better solvers solve a higher number of levels significantly better compared to the worse solvers, some interesting observations can be made. Even the worst overall solver (37) solves some levels significantly better than the best solver (1), while others solve no level better than any other solver.

One example for this are solver 7 and solver 33. Their relation is depicted in more detail in figure 5.3. It is clear that if solver 7 is part of the portfolio solver 33 should not be.

An other example are solver 25 and solver 26. See figure 5.4 for more detail. They have a near equal performance on all levels. Therefore it does not matter which is part of the portfolio.

Most of the solvers compare similar to solver 6 and solver 12. The details are given in

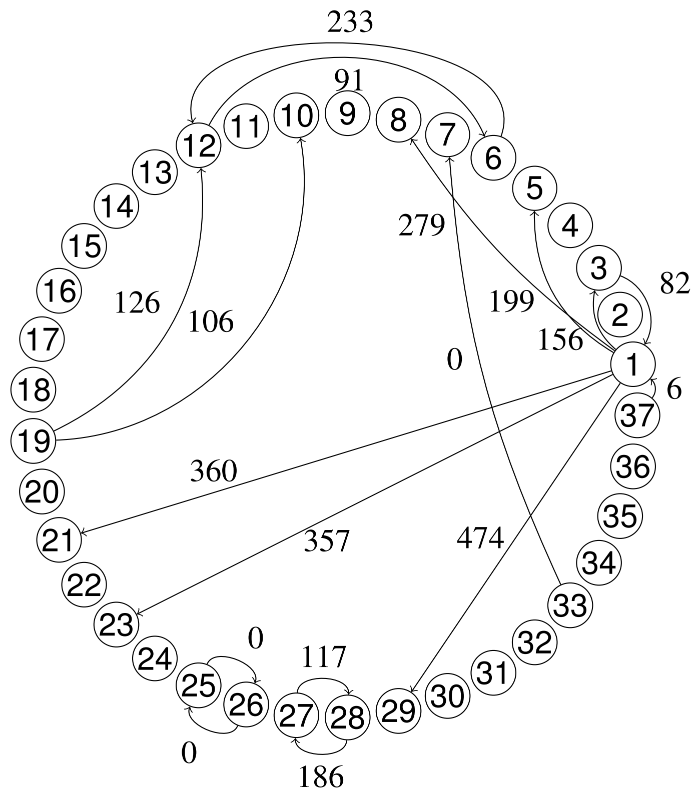


Figure 5.2: Number of levels that are solved significantly better, comparing two solvers. A node represents a solver and an edge from A to B is labeled with the number of levels that are solved significantly better by A compared to B.

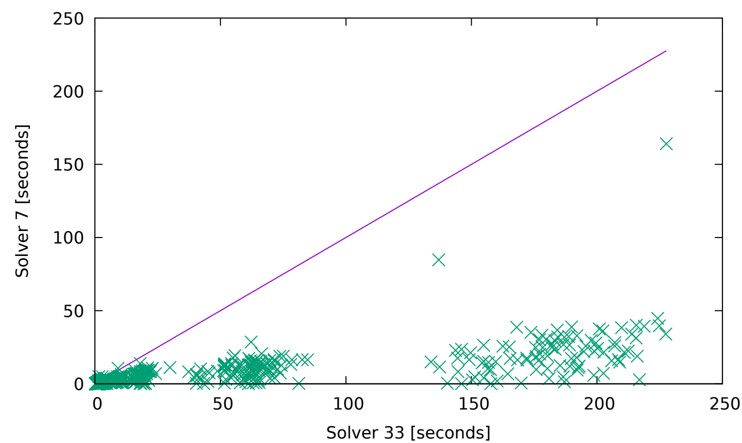


Figure 5.3: Each point represents one level. Its position along the x-axis is the time solver 33 needs to solve and the position along the y-axis is the time solver 7 needs. Solver 7 achieves a better run time than solver 33 on every level.

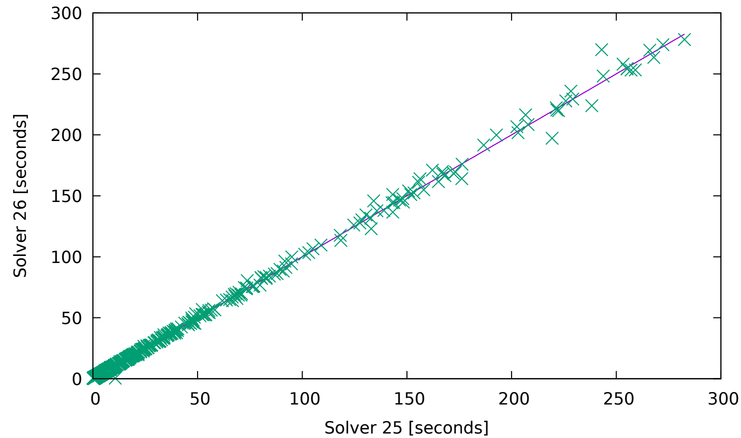


Figure 5.4: The two solvers have a very similar performance on all levels.

figure 5.5. Combining those solvers into a portfolio can be an advantage.

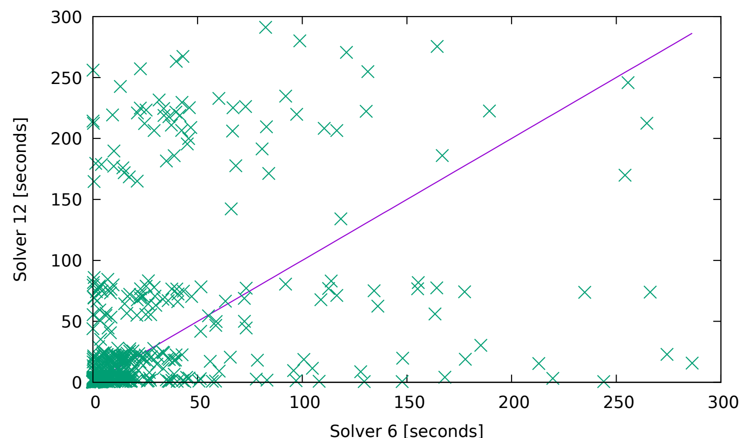


Figure 5.5: Two solvers that are different from each other. Not many points are close to the line which represents the area where both solvers are equally good. Especially interesting are the points that are close to the axes. They are solved very quickly by one solver but not by the other.

It is still possible that a number of solvers combined outperform a third on every level. To test for that we can look at the best solving time for each level and compare that to the best solving time without a specific feature. This feature can either be a search algorithm, distance metric or matching algorithm. The number of levels that are solved significantly better with a feature reflects how important the particular feature is. This is presented in table 5.2.

The choice of the search algorithm has the biggest impact on the performance of the solver. The depth first approaches do not perform exceptionally well on a lot of levels. Even if both

Feature	Significantly better solved levels
CBFS	248
A*	72
DFS	1
Directed DFS	1
Greedy	91
Hungarian	75
Closest	36
Manhattan	68
Pull	60
Pythagorean	44

Table 5.2: The table shows the number of levels that are solved significantly better if a particular feature is present.

the directed an undirected depth first searches are taken out only 5 levels are solved worse. This can be explained by the big state space that has to be searched and the comparatively small solution length that are usual for a Sokoban level.

5.5 Portfolio

We construct our portfolio by combining the solvers with the best single core performance on the large test set. We run the portfolio on a varying number of cores with an equal number of solvers in the portfolio. We use the levels from the small test set and the timeout is set to 300 seconds.

The results are given in figure 5.6. Since the machine we are using for our experiments only has 16 real cores it is expected that the difference between using 16 and 32 cores is smaller.

The speedup measured in this experiment is presented in table 5.3. For instances that were not solved within the time limit by the sequential solver we generously assume that it would be solved shortly after and use the timeout for our calculations. Since spending a lot of resources on solving easy levels is unusual we have the column *Speedup Big* where only the harder instances are considered.

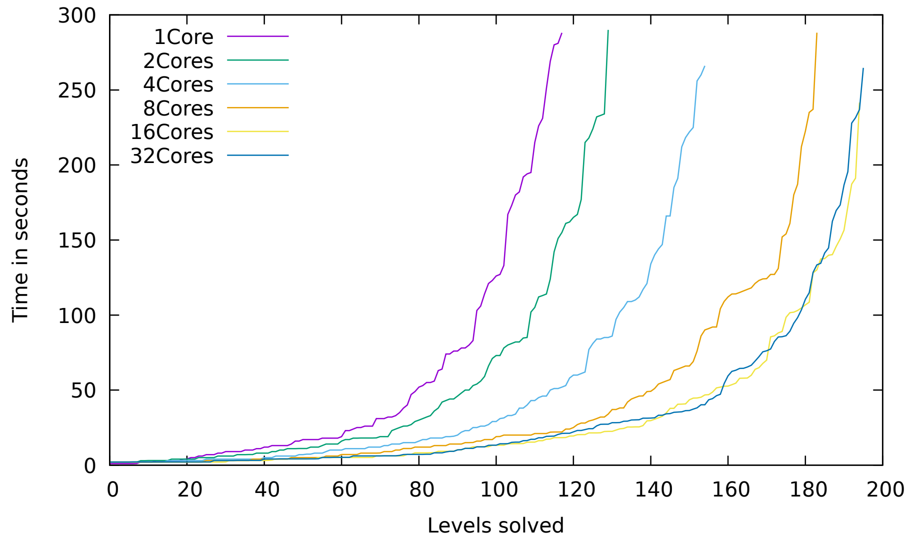


Figure 5.6: The performance of the portfolio on a varying number of cores. The levels are from the small test set.

Cores	Levels Solved	Speedup All Avg.	Speedup All Tot.	Speedup All Med.	Speedup Big Avg.	Speedup Big Tot.	Speedup Big Med.
1	118						
2	130	3.21	1.15	1.00	3.48	1.12	1.00
4	155	7.05	1.55	1.00	8.07	1.55	1.06
8	184	14.75	2.63	1.97	19.07	2.70	2.88
16	195	19.71	4.19	2.94	29.87	4.59	8.64
32	196	19.66	4.05	2.88	29.89	4.49	8.89

Table 5.3: For each number of cores tested the number of solved levels and the average, total and median speedup is given. For all instances or only the big instances (solved after at least $2 \cdot Cores$ seconds by the sequential solver).

5.6 Communication

We run the test from the section above with and without communication enabled. The results of these experiments are given in figure 5.7.

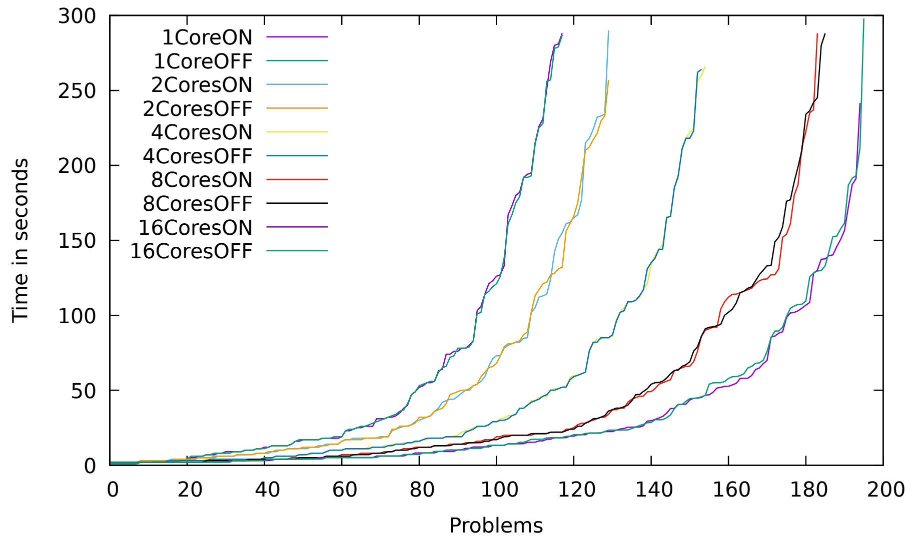


Figure 5.7: Effect of communication. ON means that communication is enabled and OFF means that it is disabled.

The way communication is implemented at the moment, it does not seem to have a noticeable effect. At least not a consistently positive one. Two solvers need to visit exactly the same state for the communication to have a positive effect. In our experiments this does not happen often since the search spaces are vast and the timeout is low, so each solver will only explore a small fraction of a search space. As shown in section 5.4 this fraction is different for each individual solver.

If the communication is implemented in the way we present in section 6.1 the benefit of communication might increase greatly.

5.7 Comparison to existing solvers

No existing Sokoban solver uses any kind of parallelization. Therefore we can only compare the single core performance of the existing solvers with *GroupEffort*. For GroupEffort we use the same data as above. For the other solvers we use their latest version[2] from the developers sites. See section 3 for more information. The results are presented in figure 5.8.

[2] JSoko:1.74, Takaken:7.2.2, YASS:2.136

Since GroupEffort is currently missing a number of important search enhancements (see section 6 for more information) it lacks behind the other solvers in single core performance. However, due to the algorithm portfolio it can, with the use of more resources, catch up to the other and eventually surpass them.

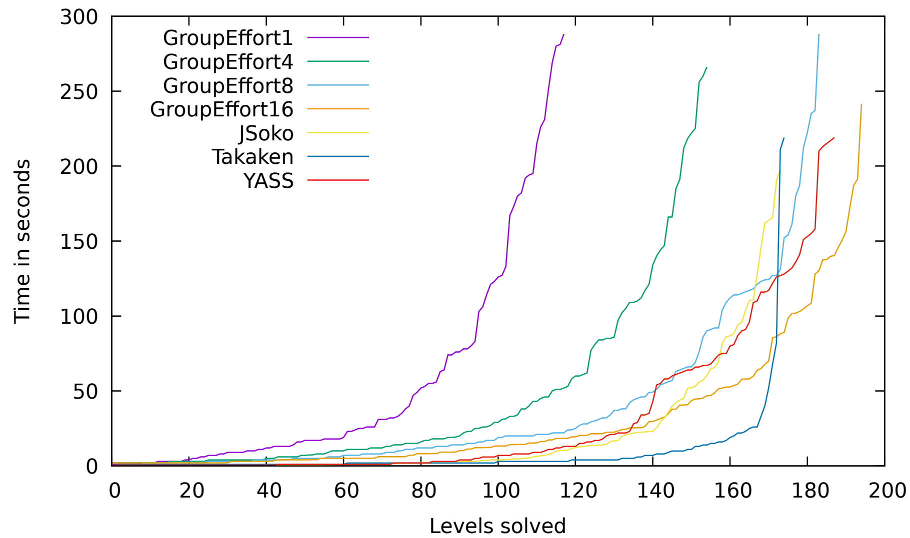


Figure 5.8: Comparison between GroupEffort and existing solvers. The number after GroupEffort denotes the number of cores available as well as the number of solvers in the portfolio.

6 Conclusion

We have shown that the group of solvers we presented is diverse and therefore the approach of algorithm portfolios can be of value for solving Sokoban. The solver we designed outperforms existing solvers but only with the use of more resources.

In order to take full advantage of algorithm portfolios, we need search enhancements that speed up the search to a higher degree, even if they might not succeed every time. In other words; in the context of an algorithm portfolio we can tolerate an incomplete search.

Most research on Sokoban solvers until now has focused on more conservative solving methods. Especially since a lot of algorithms focus on finding the best solution, i.e. least amount of box pushes. More aggressive search enhancements like relevance cuts [12] have only been considered by a few researchers. Expanding on these ideas can be a way to tackle the hardest Sokoban levels.

6.1 Future Work

For Sokoban a lot of domain specific search enhancements have been presented in literature. Some of those that might work good in an algorithm portfolio are presented in this section.

Goal Room Macros. Most notable of these are *goal room macros*. According to Junghanns and Schaeffer [12] turning them off in their solver *Rolling Stone* reduces the numbers of levels it can solve by 60%. On the other hand they do not always work. If a box has to be stored in the goal area temporarily in order to solve the level, *Rolling Stone* will not find a solution. This risk seeking behavior can be beneficial in an algorithm portfolio [9]. The idea of goal room macros can also be expanded to allow more than one entrance to a goal room. Doing this will increase the risk of missing a solution but it might be a way to tackle currently hard problems such as the one presented in figure 2.5.

Lishout-solver. Especially in the context of an algorithm portfolio a *Lishout-solver* is another interesting technique. A Lishout-solver assumes the level to be in the Lishout subclass (see paragraph 3 for a definition).

First it decides an order in which the boxes will reach their goals. It will then start with the first box an try to move it to its designated goal. Since no moves on other boxes will

be considered, it is sufficient to search a path in the game space. It then proceeds with the next box.

If the Lishout-solver succeeds a solution is found. If not the search continues from the state before the Lishout-solver started. Since checking whether a state is in the Lishout subclass is equally hard as solving it like this, we can simply start it periodically. A lot of levels are not in the Lishout subclass initially but after a few moves in the beginning to unravel a difficult box configuration they satisfy the definition of the subclass. Therefore, the Lishout-solver might have a good success rate and the payoff is a potentially huge reduction in computation.

Deadlock Detection. A lot of different ways to detect deadlocks have been used by other solvers. Besides those we implemented *local deadlock matching* is a possible way to detect deadlocks. For local deadlocks only a limited number of positions around the player is taken into consideration. *Rolling Stone* [12] used a 5 by 4 grid around the player to check if the last push created a deadlock. To do that the local grid can be match either against a hard coded database of deadlocks or a dynamic one that is generated for each level.

For this database or a database of found deadlocks in general a *forest of deadlocks* as described by Cazenave and Jouandeau [3] can be used. This is an alternative to the list of deadlocked states described in section 4.6. Compared to a list of deadlocked states it adds a lot of complexity. Whenever a deadlocked state is found, the deadlock has to be identified and inserted into the forest of deadlocks. Also checking whether a state contains a deadlock that is saved in the forest of deadlock is more complex.

The added effort is offset by the possibly smaller size of the forest compared to a list of deadlocked states and the added efficiency of sharing. If a solver receives a deadlocked state it will not visit this state. It will however explore the successors of this state if there is an other path to them in the state space graph. Even when the successors contain the same deadlock. When using a forest of deadlocks the same deadlock will be found in every state that contains it.

A Test Sets

A.1 Large Test Set

The following collections have been used in the large test set:

- grigr2001
- grigr2002
- Grigr Special
- Handmade
- massasquatch
- Microban 01 Arranged
- Microban 02 Arranged
- Microban III
- Microban IV
- Mulholland D
- sasquatchiii
- sasquatchiv
- sasquatchv
- sasquatchvi
- sasquatchviii
- Sasquatchx
- Sven
- XSokoban

A.2 Small Test Set

levels	out of	collection
5	100	grigr2001
5	39	Grigr Special
5	54	Handmade
5	46	sasquatchviii
5	50	Sasquatchx
5	100	Microban IV
5	50	sasquatchiv
5	40	grigr2002
145	1911	Sven
5	50	sasquatchv
5	57	Mulholland D
5	49	sasquatchvi

The table shows the number of levels from each collection that make up the small test set.

Bibliography

[1] Tomáš Balyo, Peter Sanders, and Carsten Sinz. Hordesat: A massively parallel portfolio sat solver. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 156–172. Springer, 2015.

[2] Adi Botea, Martin Müller, and Jonathan Schaeffer. Using abstraction for planning in sokoban. In *International Conference on Computers and Games*, pages 360–375. Springer, 2002.

[3] Tristan Cazenave and Nicolas Jouandeau. Towards deadlock free sokoban. In *Proc. 13th Board Game Studies Colloquium*, pages 1–12, 2010.

[4] Joseph Culberson. Sokoban is pspace-complete. In *Proceedings in Informatics*, volume 4, pages 65–76. Citeseer, 1997.

[5] Jean-Noël Demaret, François Van Lishout, and Pascal Gribomont. Hierarchical planning and learning for automatic solving of sokoban problems. In *20th Belgium-Netherlands Conference on Artificial Intelligence*, pages 57–64, 2008.

[6] Dorit Dor and Uri Zwick. Sokoban and other motion planning problems. *Computational Geometry*, 13(4):215–228, 1996.

[7] Robert W Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6): 345, 1962.

[8] Michael R Gary and David S Johnson. Computers and intractability: A guide to the theory of np-completeness, 1979.

[9] Carla P Gomes and Bart Selman. Algorithm portfolios. *Artificial Intelligence*, 126 (1):43–62, 2001.

[10] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[11] Andreas Junghanns and Jonathan Schaeffer. Sokoban: Evaluating standard single-agent search techniques in the presence of deadlock. In *Conference of the Canadian Society for Computational Studies of Intelligence*, pages 1–15. Springer, 1998.

[12] Andreas Junghanns and Jonathan Schaeffer. Sokoban: Enhancing general single-agent search methods using domain knowledge. *Artificial Intelligence*, 129(1):219–251, 2001.

[13] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*. Addison-Wesley Professional, 1st edition, 2015.

[14] Donald Ervin Knuth. *The art of computer programming: sorting and searching*, volume 3. Pearson Education, 1998.

[15] Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[16] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of las vegas algorithms. In *Theory and Computing Systems, 1993., Proceedings of the 2nd Israel Symposium on the*, pages 128–133. IEEE, 1993.

[17] Stuart Jonathan Russell, Peter Norvig, John F Canny, Jitendra M Malik, and Douglas D Edwards. *Artificial intelligence: a modern approach*, volume 2. Prentice hall Upper Saddle River, 2003.

[18] Joshua Taylor and Ian Parberry. Procedural generation of sokoban levels. In *Proceedings of the International North American Conference on Intelligent Games and Simulation*, pages 5–12, 2011.

[19] Timo Virkkala. Solving sokoban. Master's thesis, University Of Helsinki, 2011.

[20] Albert L Zobrist. A new hashing method with application for game playing. *ICCA journal*, 13(2):69–73, 1970.