Using an Algorithm Portfolio to Solve Sokoban

Abstract

The game of Sokoban is an interesting platform for algorithm research. It is hard for humans and computers alike. Even small levels can take a lot of computation for all known algorithms. In this paper we will 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, construct a number of different solvers and combine them into an algorithm portfolio. The solver we construct this way can outperform existing solvers when run in parallel, i.e. our solver with 16 processors outperforms the previous sequential solvers.

Introduction

The game of Sokoban is a complicated computational problem. It was first proven to be NP-hard (Dor and Zwick 1996) and then PSPACE-complete (Culberson 1997). 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.

Traditional parallelization approaches like the parallel exploration of a branch and bound tree can be difficult for Sokoban. We have to work with an uneven search space distribution, the branching factor is varying and can exceed 350 for a level bound by 20x20 walls and even the optimal length of a solution for a level can exceed 500 moves (Junghanns and Schaeffer 2001).

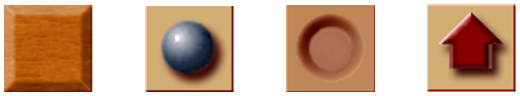
This work focuses on algorithm portfolios – a parallelization concept in which each processor solves the whole problem instance, using a different algorithm, random seed or other kind of diversification.

Preliminaries

Sokoban

Sokoban is a puzzle game that originated in Japan. 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 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 1: Entities of Sokoban

1. 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.
2. 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 $right$.

   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).
3. 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.
4. 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.

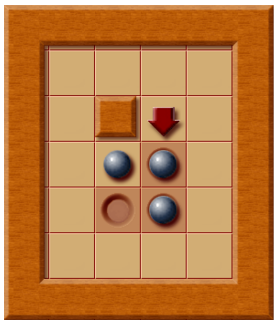


Figure 2: A possible solution string: $rddLruulDuullddR$

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.

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 (Gomes and Selman 2001).

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.

Algorithm Portfolios have been demonstrated to work well for problems like Planning or SAT (Balyo, Sanders, and Sinz 2015).

Related Work

Complexity. Towards the end of the nineties some theoretical results on Sokoban were achieved. 1996 Dor and Zwick 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 stated the question whether the original Sokoban is PSPACE-complete, which was answered by Culberson in 1997.

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

$L_{LBA} = \{(\langle M \rangle, w) \mid M$ is a 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.

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 (Gary and Johnson 1979) 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. The Rolling Stone solver they proposed is considered to be a milestone. 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, Müller, and Schaeffer 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,

[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

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, Van Lishout, and Gribomont 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.

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 (Demaret, Van Lishout, and Gribomont 2008):

- 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 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.

Our Solver: GroupEffort

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.

The source of our solver as well as the test set we used can be found at `http://baldur.iti.kit.edu/groupeffort/`.

Searching in State Space

We see solving Sokoban as a state space search problem. The state of a Sokoban level is defined by the position of each box and the player. 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.

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 edges are labeled with the push necessary to transition to the other state.

The graph is directed since a transition can be irreversible and different move sequences can lead to the same state therefore the graph may contain cycles.

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 actual maze in the current state from the last player position to the position that is required to execute the next push.

We use different graph search algorithms to find a path in the game space:

- uninformed
  - BFS
  - DFS
- informed
  - A*
  - CBFS

Complete Best First Search (CBFS) is a variation of A* (Hart, Nilsson, and Raphael 1968). The algorithm will always expand the vertex with the lowest estimated distance to a final one. Note that it is not an optimal search algorithm.

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 our definition of a state.

Our hash function is similar to the one described by Zobrist. 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.

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.

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 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.

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.

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 3 for an example.

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 3.

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:

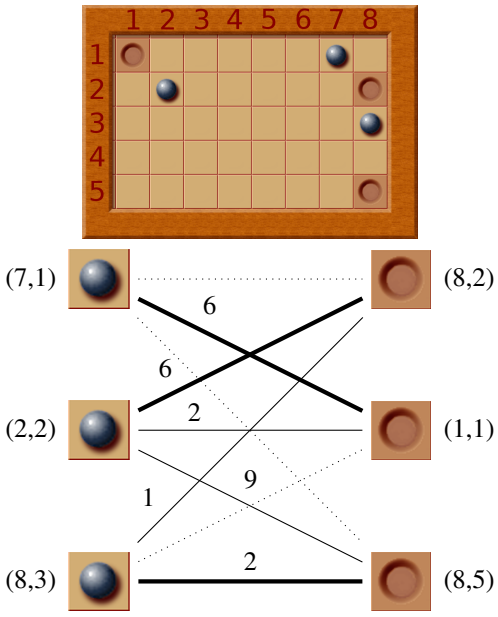


Figure 3: For calculating the distances we used the goal pull metric. The dotted lines have a weight of ∞. A possible assignment is marked. We determinded the lower bound $6 + 6 + 2 = 14$

- 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 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 (Kuhn 1955).

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.

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.

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 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 4 and a more subtle one in figure 5.

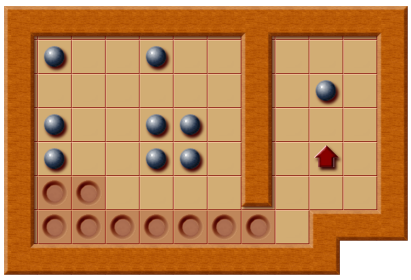


Figure 4: Examples of deadlocks. The box in the top left corner clearly cannot be pushed anywhere. The boxes in the four cluster in the middle prevent each other from being moved. The same goes for the two boxes along the left wall. The box at the upper wall can be moved but never reach a goal. The same applies to the box in the room on the right-hand side. (Virkkala 2011)

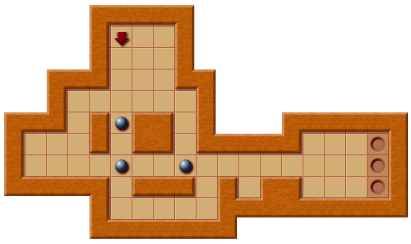


Figure 5: A more subtle deadlock. Every box and the player position is part of the deadlock.

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.

We use a pulling algorithm similar to the one used for the goal pull distance to compute 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.

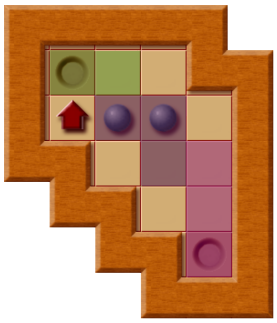


Figure 6: Two boxes are allowed in the middle area while only one is allowed in the corner areas.

Experimental Evaluation

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[4] in version 1.6.5.

Experimental Setup

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.

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

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

(Taylor and Parberry 2011). 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.

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 exception for the *Sven* collection. Since it is larger we select more levels from it.

Both test sets can be found at `http://baldur.iti.kit.edu/groupeffort/`.

Individual Solver

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.

To assess how useful a solver is for the portfolio we cannot simply compare how many levels they solved. If the best solver configuration achieves the best run time for each level, we gain nothing by using a portfolio.

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 7. 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 8. 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 9 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 10. Combining those solvers into a portfolio can be an advantage.

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

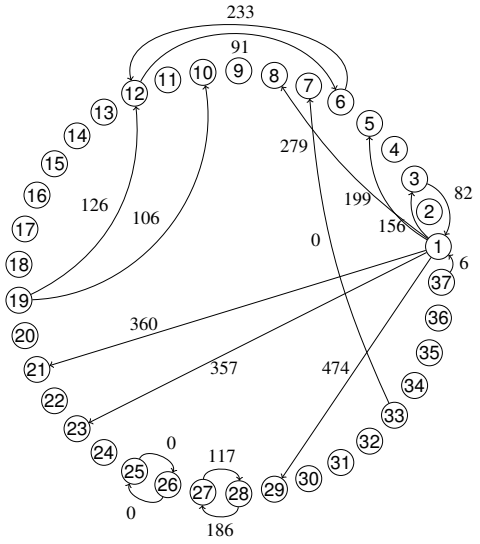


Figure 7: Number of levels that are solved significantly better, comparing two solvers. A node represents a solver and an edge from A to B is labeled with the number of levels that are solved significantly better by A compared to B.

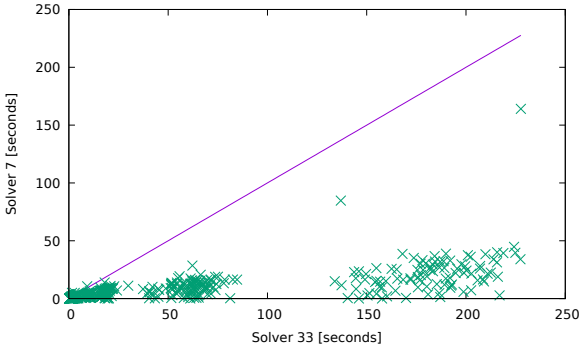


Figure 8: Each point represents one level. Its position along the x-axis is the time solver 33 needs to solve and the position along the y-axis is the time solver 7 needs. Solver 7 achieves a better run time than solver 33 on every level.

better with a feature reflects how important the particular feature is. This is presented in table 1.

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 all 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.

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

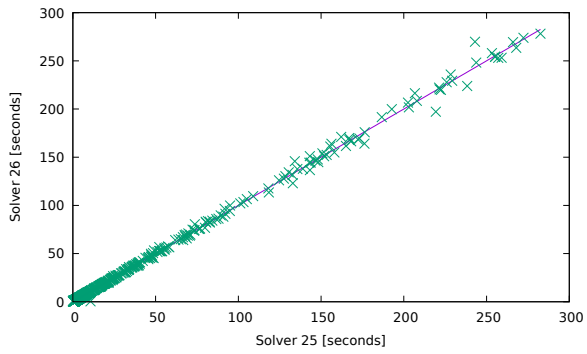


Figure 9: The two solvers have a very similar performance on all levels.

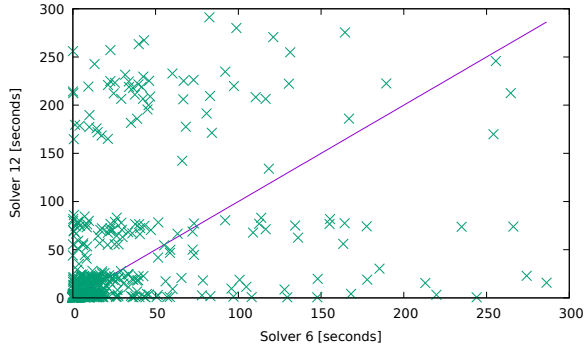


Figure 10: Two solvers that are different from each other. Not many points are close to the line which represents the area where both solvers are equally good. Especially interesting are the points that are close to the axes. They are solved very quickly by one solver but not by the other.

test set and the timeout is set to 300 seconds.

The results are given in figure 11. 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 2. 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.

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[5] from the developers sites. Some of the solvers are mentioned in the related work section. The results are presented in figure 12.

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

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

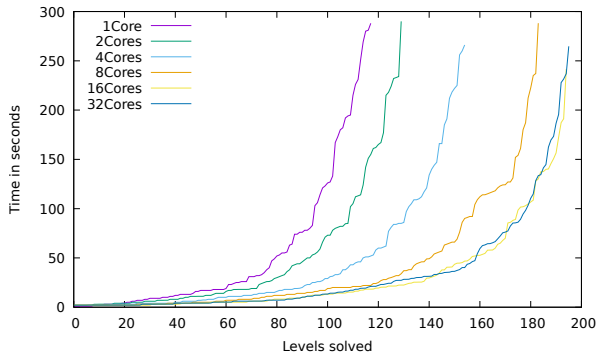


Figure 11: The performance of the portfolio on a varying number of cores. The levels are from the small test set.

Since GroupEffort is currently missing a number of important search enhancements (some are mentioned in the future work section) 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.

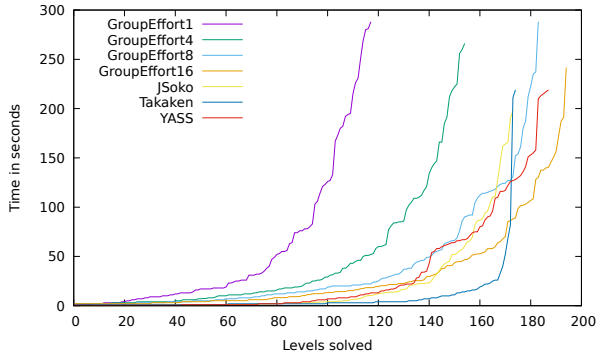


Figure 12: Comparison between GroupEffort and existing solvers. The number after GroupEffort denotes the number of cores available as well as the number of solvers in the portfolio.

		Speedup All			Speedup Big		
	solv	Avg	Tot	Med	Avg	Tot	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 2: 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).

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 (Junghanns and Schaeffer 2001) have only been considered by a few researchers. Expanding on these ideas can be a way to tackle the hardest Sokoban levels.

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 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 (Gomes and Selman 2001). 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 like level 29 of the XSokoban collection.

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. This class is defined in the related work section.

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* (Junghanns and Schaeffer 2001) 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 can be used. This database can be shared between the solvers of the portfolio. This is an alternative to maintaining a list of deadlocked states by their hash, similarly to the transposition table we presented. Compared to this list a forest of deadlocks 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 it. 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.

References

Balyo, T.; Sanders, P.; and Sinz, C. 2015. Hordesat: A massively parallel portfolio sat solver. In *International Conference on Theory and Applications of Satisfiability Testing*, 156–172. Springer.

Botea, A.; Müller, M.; and Schaeffer, J. 2002. Using abstraction for planning in sokoban. In *International Conference on Computers and Games*, 360–375. Springer.

Cazenave, T., and Jouandeau, N. 2010. Towards deadlock free sokoban. In *Proc. 13th Board Game Studies Colloquium*, 1–12.

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

Demaret, J.-N.; Van Lishout, F.; and Gribomont, P. 2008. Hierarchical planning and learning for automatic solving of sokoban problems. In *20th Belgium-Netherlands Conference on Artificial Intelligence*, 57–64.

Dor, D., and Zwick, U. 1996. Sokoban and other motion planning problems. *Computational Geometry* 13(4):215–228.

Gary, M. R., and Johnson, D. S. 1979. Computers and intractability: A guide to the theory of np-completeness.

Gomes, C. P., and Selman, B. 2001. Algorithm portfolios. *Artificial Intelligence* 126(1):43–62.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics* 4(2):100–107.

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

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

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

Taylor, J., and Parberry, I. 2011. Procedural generation of sokoban levels. In *Proceedings of the International North American Conference on Intelligent Games and Simulation*, 5–12.

Virkkala, T. 2011. Solving sokoban. Master's thesis, University Of Helsinki.

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