Using an Algorithm Portfolio to Solve Sokoban

Nils Froleyks, Tomas Balyo
Karlsruhe Institute of Technology
76131 Karlsruhe, Germany
nfroleyks@gmail.com
biotomas@gmail.com

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 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.

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 is a puzzle game where each level consists of a two dimensional rectangular grid of squares that make up the ”warehouse”. If a square contains nothing it is called a floor. Otherwise it is occupied by walls (static obstacle), boxes (movable obstacle), goal locations or the player. The goal of the game is to push each box into a goal location by controlling the player which can move in four directions (up, down, left and right). The player is only allowed to push a single box at a time.

Our Solver: GroupEffort

Each critical part of our solver[1] is implemented using a multitude of different algorithms. When executed, GroupEffort will assemble a number of solvers using these parts. We use the following graph search algorithms to find a path in the state space of the game: BFS, DFS, A* and Complete Best First Search (CBFS), which 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.

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. We use multiple different metrics to estimate the distance a box has to be pushed in order to get from square A to square B: the Manhattan distance, the Pythagorean distance and the Goal Pull Distance, which is 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. In order to assign a goal to each box we first compute the distance of each box to each goal. Then we use one of the following assignment procedures: Closest Assignment – each box is assigned to the goal closest to it. The Hungarian Method (Kuhn 1955) – minimizes sum of box goal distances. Greedy Assignment – approximate a minimal perfect matching by choosing pairs in ascending order by their cost.

A game state is deadlocked if the level can no longer be solved. Our algorithm has two ways to recognize deadlocked states: 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.

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 solv-

[1]available at baldur.iti.kit.edu/groupeffort

ing 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).

Experimental Evaluation

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. 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 (solvable in under 3 seconds by all solvers) and then selecting a fixed amount randomly. Both test sets can be found at http://baldur.iti.kit.edu/groupeffort/.

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.

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.

Our experiments showed that 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.

We construct our parellel 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 (1,4,8 and 16) 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 summarized in figure 1.

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 the other solvers we use their latest version[2] from the developers sites. The results are presented in figure 1.

Since GroupEffort is currently missing a number of important search enhancements 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.

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

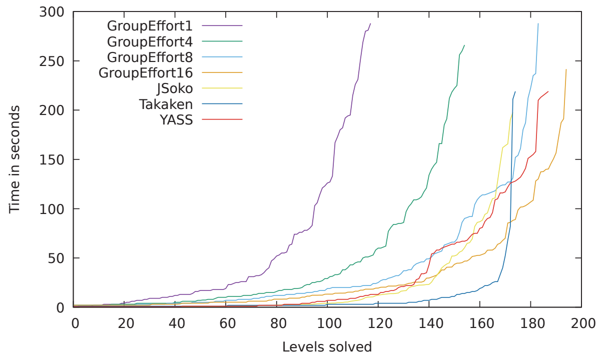

Figure 1: Comparison between GroupEffort and existing solvers. The number after GroupEffort denotes the number of cores available as well as the number of solvers in the portfolio.

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.

References

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

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

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.