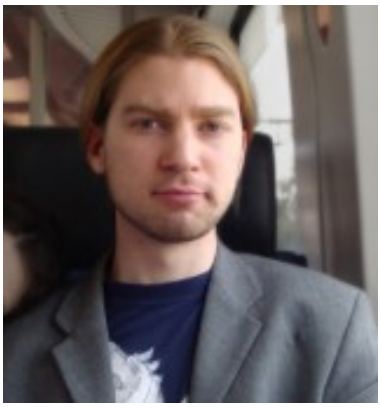
Solving the Sokoban Problem

(Artificial Intelligence 11)

M. Dorst, M. Gerontini, A. Marzinotto, R. Pana

October 14, 2011



Matthijs Dorst
1985-02-05
dorst@kth.se

Maria Gerontini
1988-01-26
mger@kth.se

Radu-Mihai Pana-Talpeanu
1988-08-06
rmpt@kth.se

Alejandro Marzinotto
1990-01-02
almc@kth.se

Abstract

Sokoban is a game based test, used to quantify the capabilities of an intelligent agent. The simple rules, and high complexity, make it a perfect fit for usage in artificial intelligence applications. This paper describes the implementation of a Sokoban puzzle solver using various single-agent search algorithms, augmented with a set of heuristics to reduce the size of the search tree.

The discussed techniques are: depth-first search, breadth-first search, iterative-deepening search, best-first search, a Genetic Algorithm and a push based solver. The heuristics involved depended mostly on the Manhattan distance between player and boxes, and boxes and goals. The methods were tested on a large set of maps, showing that in general the BestFS algorithm outperforms other search methods.

1 Introduction

Sokoban is a popular single player game, which originated in Japan. Its name translates to " Warehouse Keeper" and it describes the role assumed by the player. He must take control of a man who pushes boxes onto specific target locations, in a certain environment.

The rules of the game are simple, contributing to its appeal. The playing area, depicted in figure 1, consists of squares, which can contain 5 types of objects: empty spaces (white), boxes (red circles), goals (empty circles), walls (grey blocks) and the player (blue rectangle). The number of boxes can vary in each puzzle, but there is always only one player. The number of boxes equals the number of goals. The player can move left, right, up, or down, if there is not a wall blocking his path.

The purpose of the game, is to have the player push the boxes onto the goal squares. Boxes can only be pushed, not pulled. They can only occupy an empty square or a goal. Getting the boxes to the goals is a difficult task in itself, while doing it with the minimum number of moves is much harder. Humans rely heavily on heuristics and planning, in order to solve Sokoban puzzles, while easily being able to avoid repeated states that give no benefit. An automatic solver does not have the benefits of the human mind, and even trivial man-made decisions become difficult to implement in an agent. [1]

From the computer science point of view, the game can be seen as a tree structure, the map configurations being the nodes and the original map acting as the root node. Thus, a solution state is the map where every goal square is covered by a box. The game's complexity derives from the high branching factor (the number of possible moves the player can make) and the enormous depth (some maps need more than 200 moves in order to be solved). Assuming an average branching factor of 2.5 (we rarely want to make a step back and a direction may be blocked), and a requirement of 100 moves to find the solution, we reach a complexity of 2.5^{100}, well beyond today's computational capabilities.

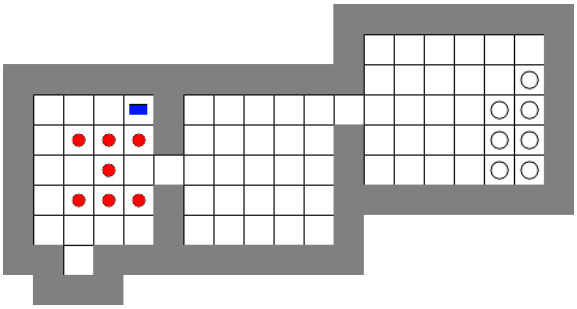


Figure 1: A Sokoban map

Our project focused mainly on implementing the informed best-first search algorithm and optimizing it as much as possible. Also, we have implemented a number of interesting methods like the push solver, or genetic algorithms.

2 Related Research

Popular research on the actual Sokoban problem itself is limited to roughly the past two decades. However, Sokoban is a P-space complete problem [2], similar on an abstract level to for example the game of Go, which has been researched for at least three decades [3]. Most of the context and research in this field has thus been well defined, and little more can be added to it without extensive knowledge of the theoretical foundations underlaying problems of this complexity.

Our research has therefore focused on compar-

ing the efficiency and effectiveness of previous research. We consider solutions based on enhancing general single-agent search [4], optimized Depth First Search [5], Best First Search [6] and Genetic Algorithms [7]. In addition, shared components for these solutions are based on work in intelligent path-finding [8] to generate the essential heuristics.

In setting out to compare these solutions it is important to note that they are highly dependent on implementation. As such, the proposed results may also reflect the ability for external researchers to duplicate the algorithms in question, rather than their objective performance. We invite the original authors to consider our research and update our results for the given maps if these prove to be unrepresentative.

3 Methods and implementation

Due to the complexity of the Sokoban problem, blindly expanding nodes in hopes of finding the solution will most likely fail on the majority of maps. This rules out the usage of un-informed search methods, such as depth-first search - DFS (expand every first child node) and breadth-first search - BFS (expand every node at a certain depth) as feasible methods of solving Sokoban puzzles. Uninformed algorithms are characterized by the lack of "judgement" in making node expansion decisions. They simply follow a fixed pattern, as mentioned above, that has no correlation to the values of the nodes.

The concept of informed-search emerges from the need of a better method and the general form of informed-search algorithms is the best-first search (Best-FS). As its name states, best-first search is a method that expands the most promising node first, according to a predefined criteria, a score. This score is given to every map and is based on heuristics, which are strategies using readily accessible information to control problem-solving processes in man and machine [9]. Heuristic methods are used to speed up the process of finding a satisfactory solution, where an exhaustive search is impractical. In our case, some good heuristics, which contribute to the score of a map, are: the number of boxes on goal positions, the distance from the boxes to the goals, the distance from the player to the closest box.

Another important aspect of Sokoban maps is the existence of deadlocks. These are states that cannot lead to a solution. Deadlocks that are inherent to a map, such as boxes being stuck in a corner, are easily avoidable. Other types of deadlocks are more difficult to manage (for example having 2 boxes at the ends of a tunnel). A good solver must be aware of as many deadlocks as possible, in order to properly prune the search tree and not consider branches that are not able to produce a solution. Some trivial deadlock positions are marked in figure 2, using a green X symbol.

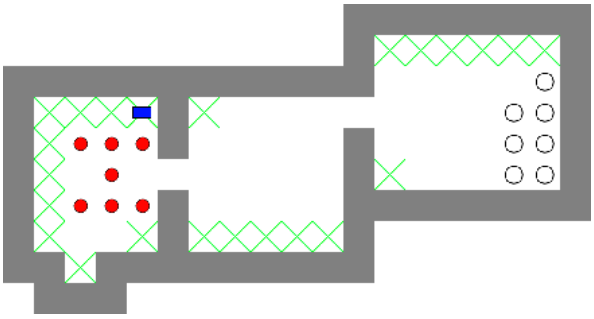


Figure 2: Sokoban map with marked deadlocks

3.1 Architecture

Several key components can be identified in the architecture, as visualized in figure 3. While interaction is subtly more complex in places than the diagram implies, in general they can be attributed to the following components:

1. **Client** handles communication with the Sokoban server. This loads the textual representation of the map, and sends the proposed textual representation of the solution back to the server.

2. **Core** initializes the client, then creates a Map and Solver object. The core acts as a process controller, ensuring that each component receives the information it requires.

3. **Map** object representation of the loaded map. Map objects can be manipulated, in that the location of the player and boxes can change, and keep track of those changes to provide a

move history. They also simplify other tasks through deadlock detection and state checking.

4. **Heuristics** based on a certain map, the Heuristics calculate a score to assign to that map which corresponds to the state of that map, solver-wise. The closer a map gets to being solved, the higher its heuristics score will be.

5. **Solver** any solver extends a base Solver class, which loads information from the core and heuristics and returns a solution when it is found.

6. **Solver Implementations** Each Solver implementation runs in a separate thread and performs the necessary operations to solve the loaded Map. Actual implementation of each solver is discussed in more detail in section 3.4.

3.2 The map

The map is received as a string of characters and is parsed into a matrix of bytes, where there is an assigned bit to each of the following: player, box, wall, goal, deadlock. Also, if 2 or more of the above mentioned entities are not mutually exclusive (the player and a goal for example), there is a value assigned to their combination. These values can then be manipulated every time a move is performed, in order to produce the resulting map.

The first action taken after the map is parsed, before attempting to solve it, is to mark the squares where if a box is pushed, the map would become deadlocked. These positions are the corners and the sides of the walls connecting the corners. No move that pushes a box onto these positions is allowed.

Later on in the process of moving the player, the map is checked for other types of deadlocks, such as 2 boxes being next to each other against a wall, a position from which they cannot be moved. If such situations occur, the solvers discard the current branch of the search tree, saving time.

3.3 Heuristics

Heuristics are used in informed-search algorithms to help with ordering the states (maps in our case), prior to expanding. We used heuristics by assigning an integer value to each map. The higher this value is, the better the map becomes and the likelihood of choosing it for expanding grows. The map's score is directly proportional to the number of boxes that are on goal positions and inversely proportional to the distances from the boxes to the goals and from the player to the nearest box. That translates to: it's good to have as many boxes on goals, it's good to have a short distance from the boxes to the goals, it's good to have a short distance from the player to the closest box.

3.4 Solvers

3.4.1 Breadth-First Search

Breadth-First Search (BFS) is a graph search algorithm that aims to examine all nodes through exhaustive exploration. The method is trivial and frequently used. BFS uses a FIFO(First In, First Out) queue which stores nodes that need to be expanded. Each visited node is placed in a separate list, called a closed set, and is not examined again. The algorithm ends successfully when it finds a solution node, without taking into account if it is the best one.

A basic implementation of this algorithm is used to solve a Sokoban puzzle. Each node represents a map in a particular state. When the algorithm runs, it expands the nodes based on the move performed by the player. BFS stops when a node in which all the boxes are placed on the goal positions is found.

3.4.2 Depth-First Search

The depth-first search algorithm is one of the most basic tree search algorithms available. Though many variants exist (see for example Nishihara and Minamide, 2004 [5]), most share common properties: a low cost in memory (since only the current branch needs to be tracked) and completeness, if the solution is within the maximum search depth and the branching factor is finite, conditions that are met in the Sokoban puzzle case.

3.4.3 Best-First Search

Best-FS is a heuristic based search algorithm. A node is expanded if it has the best score of all the nodes. The score is calculated by an evaluation

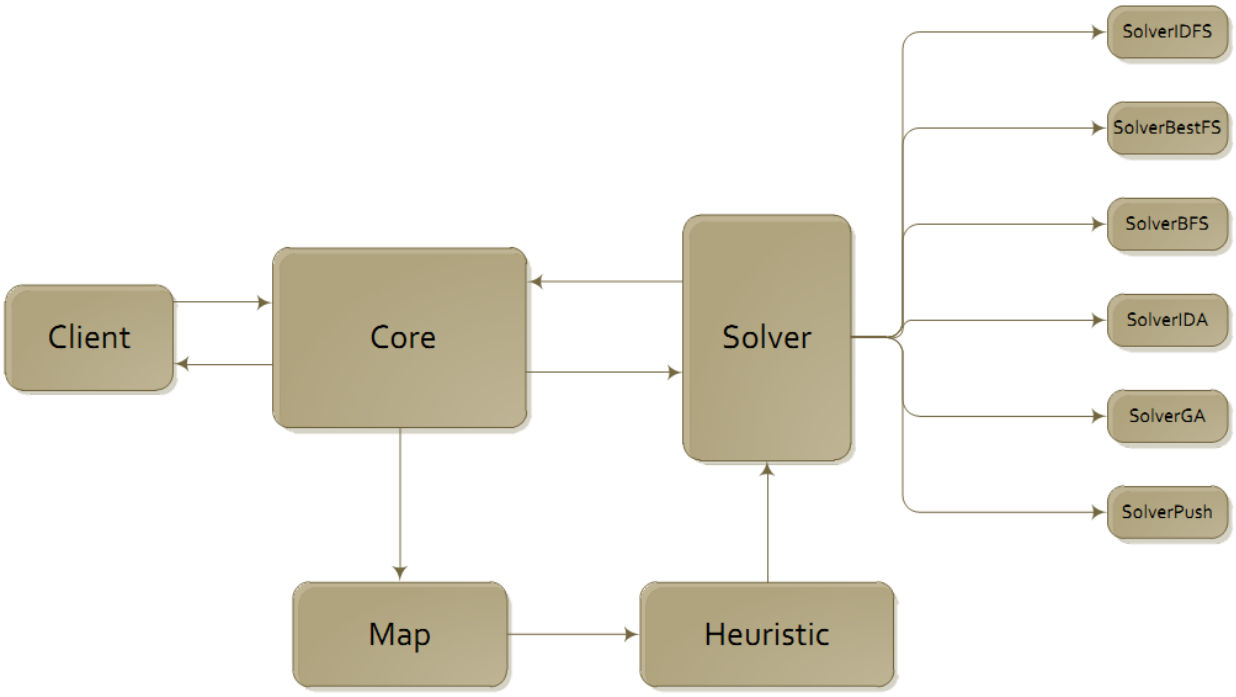


Figure 3: Architectural components and workflow

function f(n). Best-FS uses a priority queue in order to obtain the best node while the algorithm runs and a closed set in order to avoid repeated states and cycles. It is a greedy algorithm and does not always return the optimal solution. However, it gives fast results when it finds a very good solution.[6]

We used an implementation of Best-FS as the main solver of a Sokoban puzzle. Each state is represented by an object(NewNode) which contains a map and the score of the map. The nodes are expanded based on an evaluation function which is described above3.3. In each turn we reorder the expanded nodes in a priority queue and we choose the node with the highest value. The agent chooses moves which have smaller distances between the player position and the boxes or the boxes' positions and the goals. In this way, it finds an optimal way to reach the boxes quickly and afterwards push them onto the goal positions.

3.4.4 Iterative-Deepening A* Search

IDA* is an optimization of the A* algorithm which uses iterative deepening to keep the memory usage lower than in A*. Traditionally, A* uses best-first search with a heuristic function linked to the distance from the start position to the current position. It finds the least-cost path from an initial node to a goal node.[8]

In Sokoban puzzles we used IDA* in order to find the best path to a goal position. The heuristic that we used is based on Manhattan distances between the player and the boxes and between the boxes and the goal positions. The algorithm increases the depth of searching until it can find a solution state.

3.4.5 Genetic Algorithm

Genetic Algorithms belong to the class of Evolutionary Algorithms, which are based on evolutionary processes like mutation, cross-over and survival of the fittest. They have been used in search, optimization and machine learning problems for decades [7]. In fact, solving so called motion planning problems with Genetic Algorithms has been researched before [10][11].

A basic genetic algorithm is employed to solve the Sokoban puzzle. Each "individual" consists of

a series of moves, each move denoted by one of the four possible directions (Up, Down, Left, Right). Impossible moves, those who would result in either the player or a box moving through a wall or a box, are ignored. The fitness of each individual is determined through the heuristics score of the resulting map after all legal moves have been applied, see section 3.3.

Mutation causes random moves to switch to alternate directions. Since a single change can significantly decrease the fitness of an individual, this mechanism is not used for the top five individuals. These do however participate in cross-over and survival-of-the-fittest to create a new generation, thus propagating their excellent genes to the rest of the population. While this approach does not necessarily yield an optimal solution, nor any solution at all given a certain time limit, it does provide a unique ability to 'stumble upon' a good solution much faster than brute force would allow.

3.4.6 Push Solver

The main characteristic of this solver is that the search tree is build based on non-trivial moves (moves which involve a box push). The fact that this solver disregards trivial movements (moves which do not modify the box locations) makes it capable of solving the same map at a lower depth than other algorithms. The downside of this algorithm is that once the program finds the solution it has to reconstruct the path of trivial moves between box pushes.

Another problem is that the heuristics have to be adapted to prioritize not only moves but also boxes. This algorithm becomes very useful for Sokoban boards where there are plenty of possible player moves but only a few reachable boxes and few possible moves per box. The way the algorithm works makes it good for this task because it will only expand the possible moves of the reachable boxes keeping the search tree at the minimal depth.

A key aspect implementing this solver is being able to determine efficiently if the player can reach certain board positions. This is achieved with recursion using a function that spreads the influence of the player by calling itself on the surrounding squares until it can't spread any more.

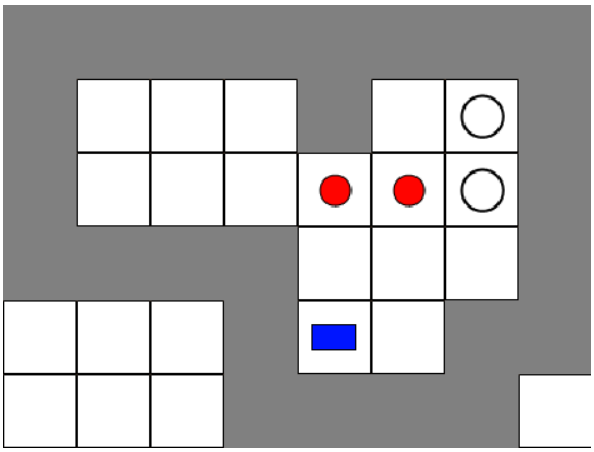


Figure 4: Map 1

Figure 5: Map 2

4 Results

To test the performance of the proposed algorithms, each was tested on several maps. Each solver was run 5 times in identical environments, keeping track of the time it took to solve the map, as well as the number of steps proposed in the solution. Since not all solvers were able to find a solution within a time limit of 60 seconds, only results for four solvers are given: BestFS, Genetic Algorithm, Iterative Deepening Search and Iterative Deepening A*. The maps are denoted Map 1, in figure 4, and Map 2, in figure 5.

The results for the given algorithms are listed in figure 6. Error bars indicate a 95% confidence interval after 5 repeated tests of each algorithm for each map.

4.1 Performance on Map 1

It should be clear from the performance on Map 1 in figure 6 that a great deviation exists between solvers, both in their proposed solutions as well as

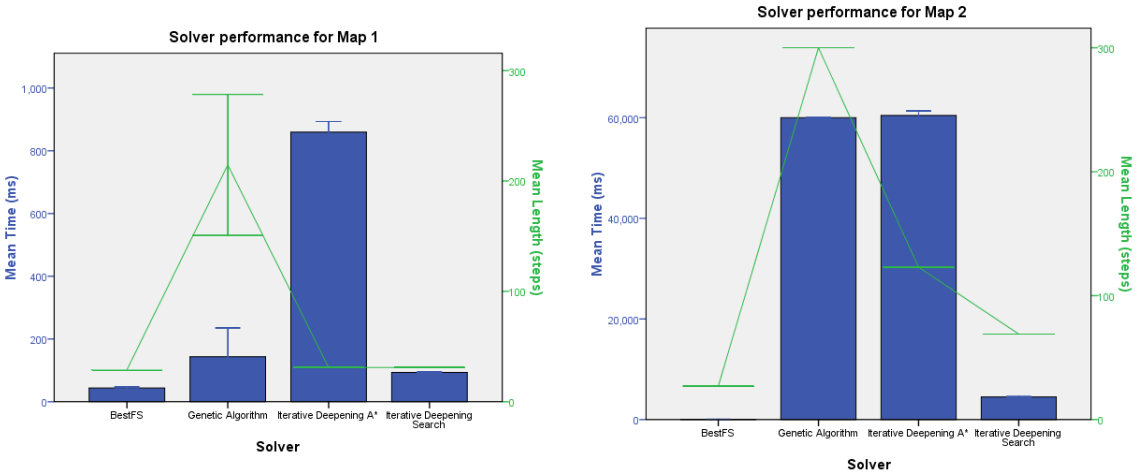


Figure 6: Results Map 1 and Map 2

the time they require to obtain those solutions. In general, BestFS performs best here, as it finds an efficient solution fast, compared to the other algorithms. Interesting to note is the great deviation in solution length, found by the Genetic Algorithm solver: as is to be expected, the randomness of this approach creates great variation, and longer individuals are more likely to solve the problem.

Interestingly enough, Iterative Deepening A* performs worst of all four solvers. There is no easy explanation for this: the solution it finds appears to be close to optimal, yet it is very slow. This hints at an inefficiency in the algorithm that might be resolved in the future.

4.2 Performance on Map 2

A similar result can be seen for Map 2: BestFS again performs fast, and finds the shortest solution. Iterative Deepening Search is comparatively close, while the Iterative Deepening A* algorithm reaches the 60 second time limit. The Genetic Algorithm does in fact reach that limit and did not find a solution on any trial. Likewise, its proposed solution length is the maximum individual size of 300 steps.

4.3 General Performance

From the given data, it should be clear that the solvers may all be capable of solving simple maps, yet are not equally effective at doing so. The BestFS algorithm performs well under any circumstance, though performance of the Genetic Algorithm solver varies greatly. IDA* performance is an order of magnitude worse than BestFS performance, even though both algorithms share many common assumptions: this can be cause for further investigation. Surprisingly enough, the generic Iterative Deepening Search algorithm keeps performing adequately in these maps, even though it is an uninformed search method.

The algorithms are also tested on additional maps. However, only BestFS was capable of solving more complex maps within the given time limit. General observations from these results indicate that solution length and time requirement are correlated; however, as the solution length for Map 2 is lower than the solution length for Map 1, yet the time requirement is greater, it is quite likely that this correlation will prove to be a complex one.

5 Conclusion

The naive approach of implementing a Sokoban solver based on pure brute force search has proven to be inefficient, even for simple board configurations. In more complex scenarios, no solution is obtained within a reasonable time limit.

Huge performance boosts are observed when the traditional search methods are augmented with

heuristics, to prioritize nodes with higher values over those which are redundant. However, different types of Sokoban boards require different strategies. For this reason, using a single heuristic to solve every map will not yield optimal results. The Sokoban problem is not beyond current computational power, but the current heuristics that help us explore the search tree are static, incapable of adapting to different maps. Thus, we have to find a way to mimic human behaviour in the machine, and make it capable of changing its strategy at runtime.

A multi-faceted approach which utilizes multiple solving strategies, should in theory have an advantage over specialized solutions, optimized for a select number of puzzles. In practice however we found that a good solver will almost always be significantly better than a bad solver, irrelevant of the puzzle. Great attention should be paid to the heuristics, as these are of utmost importance for the performance of all informed search methods. This should come as no surprise, as previous research has led to the same conclusion [4].

An interesting piece of future work would be to optimize the heuristic function. Heuristic functions can totally affect the solvability and the complexity. For example, heuristics based on pattern detection, move ordering or macro moves can be used to boost agent's performance. However, which is the best heuristic is a matter of debate.

References

[1] A. Junghanns and J. Schaeffer. Sokoban: A challenging single-agent search problem. In *In IJCAI Workshop on Using Games as an Experimental Testbed for AI Reasearch*. Citeseer, 1997.

[2] J. Culberson. Sokoban is pspace-complete. In *Fun With Algorithms*, volume 4, pages 65–76. Citeseer, 1999.

[3] D. Lichtenstein and M. Sipser. Go is polynomial-space hard. *Journal of the ACM (JACM)*, 27(2):393–401, 1980.

[4] A. Junghanns and J. Schaeffer. Sokoban: Enhancing general single-agent search methods using domain knowledge. *Artificial Intelligence*, 129(1-2):219–251, 2001.

[5] T. Nishihara and Y. Minamide. Depth first search. *The Archive of Formal Proofs. http://afp. sf. net/entries/Depth-First-Search. shtml*, 2004.

[6] R.E. Korf. Linear-space best-first search. *Artificial Intelligence*, 62(1):41–78, 1993.

[7] D.E. Goldberg. Genetic algorithms in search, optimization, and machine learning. 1989.

[8] B. Stout. Smart moves: Intelligent pathfinding. *Game Developer*, pages 28–35, 1996.

[9] J. Pearl. Heuristics: intelligent search strategies for computer problem solving. 1984.

[10] M. Amos and J. Coldridge. A genetic algorithm for the zen puzzle garden game.

[11] J. Coldridge and M. Amos. Genetic algorithms and the art of zen. In *Bio-Inspired Computing: Theories and Applications (BIC-TA), 2010 IEEE Fifth International Conference on*, pages 1417–1423. IEEE, 2010.