PROJECT ANTON
A MEMORY EFFICIENT SOLVER FOR SOKOBAN

Venkatesh Kotwade (B18CSE023)
Project Report for Artificial Intelligence Course (CS314) by Dr. Yashaswi Verma

Introduction

Solving hard puzzles was one of the major inspirations behind research relating to search algorithms. Sokoban since its inception in 1982 by Thinking Rabbit Co.[1] has been a popular puzzle game across the globe. Its player community is still active today and even have their own wiki and discussion portals[2]. There have been multiple efforts for creating solvers for Sokoban. Currently, the most powerful solver is Festival[3], which uses FeSS (Feature Space Search algorithm)[4]. Most recent advancements are based on detecting patterns that are known to be difficult by preprocessing the level structure. These solvers are known as Pattern Database Based solvers. They are computation heavy and memory consuming algorithms.

Sokoban's goal is to move each box in one of the goal positions, using only push operation by the agent. The agent is normally referred to as "Sokoban" but throughout this report and also in code, the agent is referred to as the "Robo". Each goal is called a "hole" as it imagines that the boxes are being pushed into holes.

General rules of the game

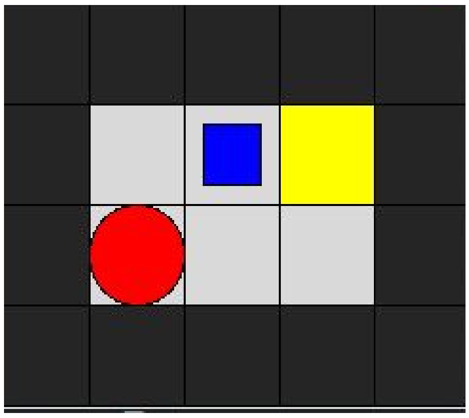
Fig. 1 Simple level in Sokoban

1. The Robo can not pass through walls.
2. The boxes can not be pushed through walls.
3. The Robo can only push the box. (never pull)
4. There is no such concept of scoring in the game, as long as you are able to push each of the boxes into one of the goal positions, it is considered as a win.
5. Two boxes can not occupy the same position.
6. The Robo and a box can not be in the same position.

Fig 1, represents a simple level in Sokoban. The red dot represents the Robo, the blue square is the box and the yellow square is a hole or the goal position. A solution to the above level will be simply going up and right, i.e. "UR" in the form of an action encoded string.

Problem Statement and Core Challenge

A Sokoban puzzle is solved when each box reaches any of the goal positions. The complexity of the problem lies in the astronomical size of the problem state. The simple level demonstrated in figure one has a possible search space of 30 states. This search space explodes as soon as we increase the size of the puzzle. Given,

A: Playable area in unit squares
K: Number of boxes
α: Accessibility factor [less than 1]
S: Size of Search Space

$$S = \alpha \times {}^{A}P_{K}$$

P is permutations operator.

Other than this, one more source of difficulty is from the dead ends that might form during the searching. Detecting Dead Ends like in Fig 2 is a difficult task. Pruning states like these as early as possible is key to achieving acceptable runtimes of the program.
The above challenges contribute majorly to the difficulty of this search problem. I have tried to come up with strong heuristics and methods to calculate deadlocks for pruning misleading states.

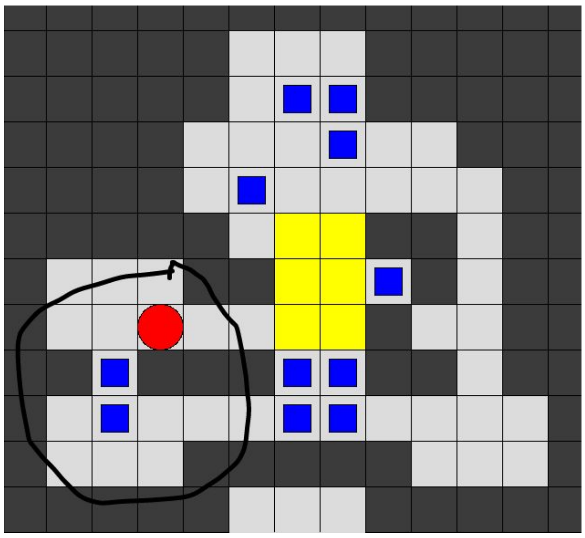
Fig. 2 Example of Complex dynamic Dead End

Problem Definition

The problem has been proved to be NP-Hard[5] and is PSPACE-COMPLETE[6].
We will define it as a standard search problem.
A search problem will consist of,
An abstract State Representation and a singleton class named Problem and a function for Searching.

Problem

It will have,
1. Start State: the root state of the search tree.
2. Level: Environment representation in the form of a 2D matrix, will only contain constant/static information, i.e. the unmovable elements like holes and walls.

Will have methods for,
1. *A Goal Test*: Checking if the given state reference is Goal State or not.

```
Bool Goal Test(State & S) :
  For pos in S→boxPositions:
      If pos not in S->holePositions:
            return false
  return true
```

2. A successor function, which will return a container having a set of non-dead-end successor states

```
State[] SuccessorStates(State & S) :
  SuccessorStates← empty States[]
    For newState in NeighbourStates(s):
      If not isDeadEndState(newState):
         Insert(SuccessorStates,newState)
    return SuccessorStates
```

State

It will have,
1. boxPositions : positions of all boxes
2.holePositions: positions of goals, (static henc common for all states)
3.roboPosition: the position of Robo in the Environment
4. actions: Actions taken till now to reach this state from root state.
5. hash : hash of this state to avoid revisiting the state

Data Set

We will use levels from data set MicroGon. Throughout the literature on Sokoban Solvers, a level is considered to be solved by the solver, if it finds the solution in under 10 minutes and under 1 GB of memory use.

Anton (the solver discussed in this report) however never took more than 150 MB of memory over multiple trials.

Background Survey

The problem has been extensively researched. The general trend in literature published recently has been detecting possible deadlock patterns in the level by preprocessing, using pattern detection techniques with the help of a domain-specific knowledge database and convolution. This method

although very fast requires a preprocessed database of possible deadlocks and resembles learning the solution of problems by playing multiple levels, and memoizing failing state patterns to build a domain-specific knowledge database, and hence does not fit for a typical search problem. Instead, it resembles a solution powered by Machine Learning over multiple failed attempts of solving the problem.

I will try to take on this problem as a pure search problem with a minimum amount of offline computation and focusing on minimizing memory use.

Discussion

Iterations over Search Algorithm

I tried iterating on multiple possible approaches to solve the problem. I started searching for the best search algorithm for solving the problem. Given the nature of the problem, as discussed in the introductory section, naive DFS and BFS will fail, simply due to the size of state space. Greedy search with considering the number of moves as the cost will mislead the search in the wrong direction, as the number of moves does not matter in solving the game. The only viable search algorithm left was A* and its variations. Among its variations applying Iterative Deepening A* search was not possible as the depth of solution could be as high as 200-400 and for IDA* To succeed, we will need efficient and consistent Heuristic Function. By doing analysis of extra time spent on computation of efficient consistent Heuristic Function and the actual time required for a solution using A*, it was observed that using IDA* was not feasible. Hence I finalized on A*.

Iterations over various Heuristic Functions

Sum of Closest Manhattan Distance to any Hole:

Nature: **Naive**, Consistent, Inefficient
In this approach, the heuristic just sums the distance to the closest hole for each box. To this sum, we finally add distance between Robo and any closest box. The distances are calculated using Manhattan Heuristic.
Complexity: $O(|B|^2)$
B: a set of the position of Boxes
|B|: the size of B

Minimum Weight Perfect Bipartite Matching among Box Set and Holes Set using Manhattan Distance as Weight of Edge

Nature: **Novel**, has both consistent and overestimating versions, moderately efficient

In this heuristic approach, we define a bipartite graph **G** having two sets of partite vertices sets namely H (holes) and B (boxes). There are $|B|^2$ edges in this graph, each edge is from one of the vertices in B to one of the vertices in G. The weight of each such edge is Manhattan distance between coordinates of vertices at ends.

Our aim is to find a min weight perfect matching with the Minimum Net weight of Edges included in the set. A perfect matching is one in which each vertex in the bipartite graph is incident to an edge in the matching. According to problem constraints, we have |B|=|G|. Hence the number of edges in such Minimum Weight Bipartite Perfect Matching will be |B|. This is because each edge from set B will satisfy one vertex in set H.

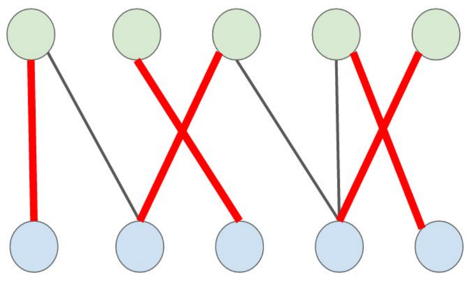

Fig 3: Example of Perfect Matching demonstrated in Red color.

Complexity: $O(|B|^3 \times \beta)$

β: it is the inverse of the average number of repetitions of set B over all instances of Heuristic Function call. It is always less than 1 and tends to 10^{-4}.

Novel Optimisation

One key observation was Min Weight Perfect Bipartite Matching (MWPBM) was exclusively a function of set B. This is because the set H is constant throughout all the states and hence we can avoid recalculating the heuristic for states having the same set B. To implement this, I created a static instance of std::unordered_map (hash map), which stored only the hash of set B as key and the value being the weight of MWBPM. This way the complexity is amortized down to a huge extent for the overall search algorithm.

Minimum Weight Perfect Bipartite Matching among Box Set and Holes Set using Exact Constrained Distance as Weight of Edge

Nature: **Novel**, has both consistent and overestimating versions, highly efficient for high depth solutions
Works in a similar fashion to the last discussed heuristic, the only difference is instead of Manhattan Distance we take actual constrained distance due to wall constraint and design of the level. This adds to extra computations, but it is observed that these computations never reach their worst-case complexity and are bound by a much lesser number of actual computations.

Complexity: $O(|B|^2 \times \bar{d} \times \beta)$

$\bar{d}$: the average of dimensions of the levels (h+w)/2

Novel Optimisation

We used the concept of linear conflict to better estimate the actual length of the path between Box and Hole. Linear conflict considers dynamic obstacles like boxes and adds a fixed weight to the net cost of path, each time such obstacles are detected. It gives a better bound for heuristic while still keeping it consistent. It is explained in a better way using the following example.

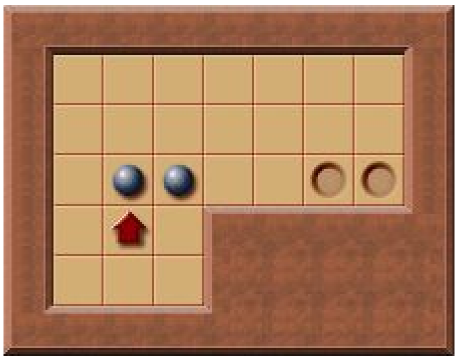

Fig 4 Here the pointed Box has a linear conflict with another box and it will need extra moves to bypass the other box. Such moves are added to the weight of the edge while calculating the Constrained Distance between Box and the nearest hole.

Iterations over Detection of DeadLocks

Dead Locks are positions in the level matrix such that if any box reaches that position, the state can never result in a solution of the level. There are broadly two types of DeadLocks, one being static Deadlocks caused by the position of Walls and Holes, and the other one being Dynamic Deadlock caused by the position of Boxes combined with the position of walls.

Detecting First order Static Deadlocks

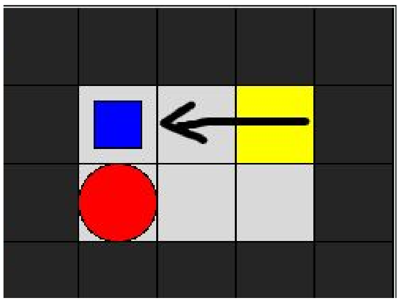

Fig 5 Here a First-order static deadlock is caused because of the box being obstructed by walls from horizontal as well as vertical positions

These are caused by a direct conflict between walls and boxes. These are easy to detect and just take 4 checks to determine their existence for each box

.

Detecting Second-Order Static Deadlocks

These deadlocks are caused by an indirect conflict with walls.

Novel Approach: We can detect these by doing a multi-sourced BFS from holes with forwarding checks to see if a box can be pushed from a given position. This multi-sourced BFS will give us a DeadLock table which will mark all the positions with Second order (and in turn first order also) Static Deadlocks. Initially all the positions are marked as deadlock except the position of holes. The BFS will mark a position as non-deadlock if it can be pushed into a previously discovered non-deadlock position using a valid move.

This is demonstrated in Fig 6

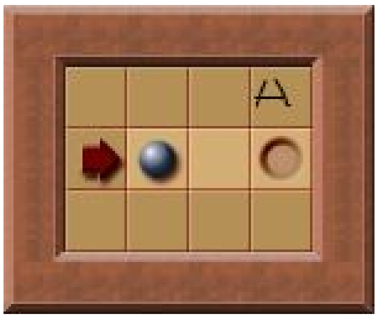

Fig 6: The BFS will start by marking all positions except the hole position as deadlock after that it will check its neighboring position A. As we can see a box on Position A can never be pushed into a non-deadlock position using a valid move because it is vertically constrained. The result of BFS will be marking the shaded squares as Deadlock Position. If a box is ever pushed to such deadlock position it is considered as a Dead End State and the subtree is pruned.

Detecting First order Dynamic Deadlocks

These deadlocks are caused by a direct conflict with partially constrained boxes and walls.

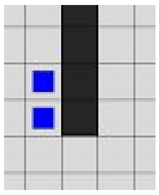

Fig 7: In the above fig both the boxes are in deadlock due to being vertically constrained because of each other and being horizontally constrained because of the wall.

This type of deadlock can be detected by a depth 2 DFS query for all possible constraining objects in all directions.

Detecting Second-order Dynamic Deadlocks

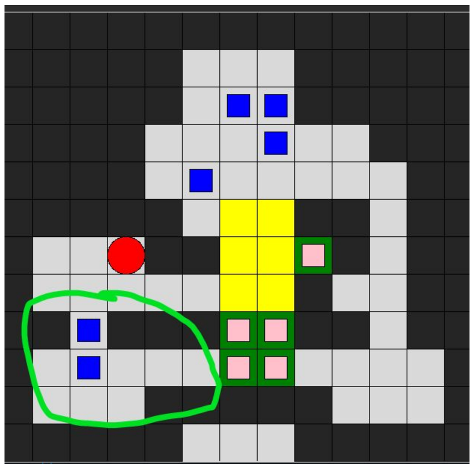

Fig 9: The area marked by the green encircle will never be reachable for the Robo agent shown by red solid circles.

This is the toughest type of deadlock detection and requires a high number of computations to effectively detect them without false positives.

Novel Approach: Detecting this type of deadlocks requires BFS from each Box. This BFS simulates reverse moves while transitioning to new positions. If a box is detected on a position, the BFS checks if this box can be removed from this position by Robo on currently untraversed squares, without conflicting with boxes in traversed squares or walls. This way we can reverse simulate moves. If the BFS is able to reach the Robo, that means the Box is not constrained, otherwise, the Box is deadlocked. If a single box is found deadlocked then the algorithm returns that the state is a Dead End State.

Average Complexity: $O(|B| \times \bar{d})$

$|B|$: the size of the set of box positions

$\bar{d}$: an average of dimensions of the levels i.e. (h+w)/2

Multiple novel approaches for pruning the Dead End States and memoization using caching to reduce redundant computations have been introduced. These improvements and optimizations have been implemented using Object Oriented and modular coding practices. The implementations have been done in C++ to minimize programming language overhead over the efficiency of algorithms. C++ also helped in having better control over the object properties of the problem state. One example of such control is the ability to define static members which are common to all the objects of the class. These static members help a lot in reducing redundant memory usage. Hence I feel that the solution to the given problem statement is an Engineering solution created using high standards for optimization at the implementation level itself.

The algorithm, Observations, and Results

Note: All the results are stored in standard_level_results.txt. All the results can be reproduced by running the test_runner.py script.

We mainly compiled two binaries, one with Second-order Dynamic Deadlock Checks and other without it. So basically our algorithm is A* with *Minimum Weight Perfect Bipartite Matching Heuristic* .

The program was run on a set of 155 levels created by David W Skinner. The set is named as Microban and is available here.

The program solved 149 out of 155 levels. It ran out of time in 4 levels and 2 levels mysteriously failed without detecting a goal. The algorithm must have encountered a false positive for dead end and pruned a subtree with the goal state. These 2 levels were later re-run without strict dead-end checks, the solution was found in 4 minutes and 5 minutes respectively.

On average, the algorithm took 12 seconds to calculate the solution for each level using the strongest MWPBM consistent heuristic. The average memory consumption for each solution was 40 MB. This was the main objective of Anton. It succeeded in keeping the memory consumption low through hashing.

On the other hand, we also ran the overestimating (non-consistent) version of the MWPBM heuristic, it ran with an average of 8 seconds to calculate the solution of each level. But at the same time, it failed on 15 levels. This was due to the potential misleading of the search algorithm due to overestimation.

The number of nodes expanded with strict dead-end checking was considerably low. Strict dead-end checks resulted in an overall reduction of 30% in the run time of the algorithm. It also resulted in a reduction of 63% in the number of nodes expanded.

Overall Anton Sokoban Solver succeeded in solving the given data set with good efficiency.

Limitations

There are some benchmarking data sets that have an absurd number of boxes (in the factor of 100s) per level. These benchmarking level set are meant for ML-powered solvers and hence we did not attempt to solve these sets. Microgon is a set suitable for purely search-based solvers.

Algorithm Complexity:

As discussed above the size of search space is,

A: Playable area in unit squares

K: Number of boxes

α: Accessibility factor [less than 1]

S: Size of Search Space

$$S = \alpha \times {}^{A}P_{K}$$

P is a permutations operator.

The search space grows in a factorial manner.

The branching factor for each state is 4. This can easily be brought down to 3 by hashing. Further level constraints and deadlock pruning can easily bring down the branching to 1.1 after a sufficient amount of nodes in the queue.

On running the algorithm with consistent heuristic the average depth of the solution was found to be around 70 for the Microgon problem set.

The overall complexity of the algorithm is similar to A*. The most efficient consistent heuristic has a complexity of $O(|B|^2 \times \bar{d} \times \beta)$

Hence the theoretical time complexity for the algorithm will be

$O(|B|^2 \times \bar{d} \times \beta \times (b^*)^m)$

In our heuristic the b* reaches close to 1 and m can reach up to 65.

b*: resultant branching factor

$\bar{d}$: the average of dimensions of the levels (h+w)/2

β: it is the inverse of the average number of repetitions of set B over all instances of Heuristic Function call. It is always less than 1 and tends to 10^{-4}.

|B|: a set of boxes

Summary and Conclusion

We explored solving the Sokoban puzzle using A*. The solver was able to solve 98% of the levels in the dataset. Each solution had an average depth of 70 for a moves-optimal solution and a depth of 100 for a suboptimal solution.

We observed that the key to solving this problem is successful early pruning of bad subtrees, either through overestimation or through strong deadlock checks. Overestimation gives fast but suboptimal results for low depth goal states,but overestimation also results in a high rate of failure in levels with high depth solutions. Giving more time to this algorithm will not result in the surety of results as difficult levels require very strong pruning methods. We also observed that although we gave a time limit of 10 minutes to the program, most solutions were found in under 3 minutes. This shows that once the number of nodes in the fringe increases, it is increasingly difficult to reach the goal. It is difficult to detect difficult deadlocks programmatically, they will eventually require preprocessing using ML models to detect possible dead ends. We need a pattern database for improved pruning. Pure search problems can only go so far.

Overall the solver performs better than most humans and fails only on absurdly difficult levels meant for professionals. It succeeds in its purpose and can also be used as an extension in Sokoban game to provide help to players.

Further Possible Improvements

We can improve the deadlock detection and avoid false positives. As discussed in the Results section, the algorithm exited on 2 levels out of 155, without finding a

solution. This must be caused by either hash collisions or false-positive given by deadlock detection function resulting in the pruning of subtree with goal node.
On primary investigation, I suspect that it must have been caused by a false-positive given by the deadlock detection function. The deadlock detection algorithm might have failed on some corner cases. The hashing function had a domain of size 10^{18} and the number of states hardly exceeded 10^{7}. Theoretically, the hash collisions shouldn't happen.

It needs further investigation to pinpoint the cause.

References

[1] https://en.wikipedia.org/wiki/Thinking_Rabbit
[2]http://www.sokobano.de/wiki/index.php?title=Main_Page
[3]https://en.wikipedia.org/wiki/Festival
[4] https://ieee-cog.org/2020/papers/paper_44.pdf
[5]M. Fryers; M. T. Greene (1995). "Sokoban". *Eureka* (54)
[6]Culberson, J. (1997). Sokoban is PSPACE-complete.
[7] http://www.sourcecode.se/sokoban/levels
Above is link to level database