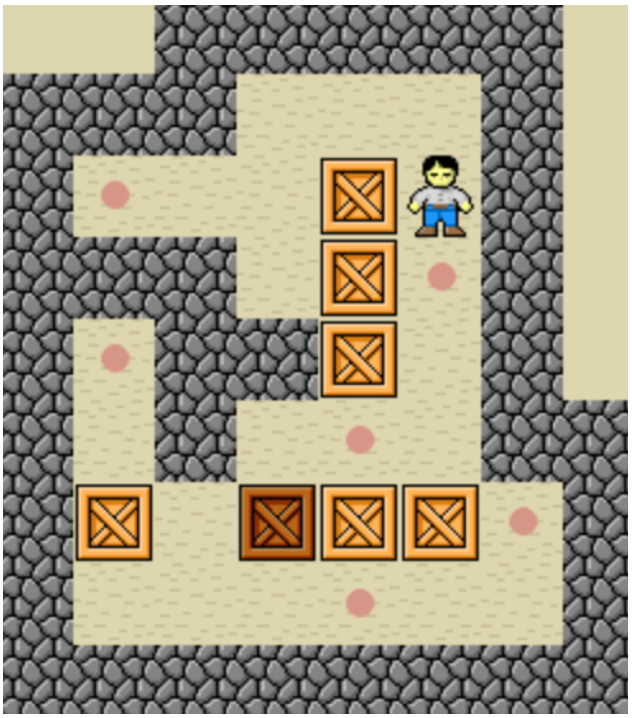
Sokoban Solver using A^* Search

Final Report for CS271 Introduction to Artificial Intelligence

Kalyani Asthana - 55889159
Jason (Zesheng) Chen - 79361335
Hamza Errahmouni Barkam - 22766303



Department of Computer Science
University of California, Irvine
Fall 2020

Contents

List of Figures

1 Problem Statement

1.1 Sokoban

Nowadays, Artificial Intelligence (AI) has become a hot topic in Science and one of the fields that will determine our future. Generating robust algorithms in AI, especially for complex problems is essential. It's even better to test these robust algorithms on setups that are easy to explain and comprehend in terms of space and time complexity. One of such easy setups is the game of Sokoban and one of the several robust AI algorithms is A* search.

In this project, we have made an AI Solver using A^* Search that is able to solve different Sokoban puzzle games. Sokoban is a game played on a board of squares, where each square/coordinate of the board can be either a wall, a box, a player, a storage location or just the empty floor. The player is allowed to move vertically or horizontally to empty floor squares and storage locations only. The player can move to these locations either while pushing a box or without pushing a box. The goal of the player in this game is to move all boxes to storage locations.

1.2 Sokoban Properties

Research shows that the game of Sokoban has a PSPACE-complete in terms of Computational Complexity. Advancing and finding ways to solve Sokoban could help the AI community in many ways. Sokoban is a challenging domain for computers, mainly due to the following reasons:

1. Some moves are not reversible, which can even make you lose a game and or have to start it from scratch.
2. Sokoban has a complex lower-bound estimator O(N^3), when we have N store locations. However, even with this lower-bound, it is not enough to make the problem significantly easier.
3. Some Sokoban problems require a chain of moves that depend on each other (sequential) to get a solution. Many of these moves interact with each other, making it complex and hard to divide the problem into independent sub-goals.
4. The graph underlying a Sokoban problem is directed[1]. Moreover, there are certain zones of the search spaces that do not contain solutions. Both these characteristics cause that this game has deadlocks (states that have no solution). Examples of the simplest deadlocks are presented on the third section of this report.

[1] From this source, A directed graph is graph, i.e., a set of objects (called vertices or nodes) that are connected together, where all the edges are directed from one vertex to another. A directed graph is sometimes called a digraph or a directed network. In contrast, a graph where the edges are bidirectional is called an undirected graph.

2 Approach

2.1 Methodology

Preparation: Before actually starting the project we tried to understand what exactly we are dealing with. Do understand what we're dealing with, we had weekly meeting for discussing the following points:

- Tools required
- Defining inputs, outputs, Class and function definitions
- Discussions on division of work
- Coming up with timeline for the project and sticking to it
- Discussing incremental changes once we had a working version of A^*
- Testing
- Results and Conclusions

2.2 Milestones and Checkpoints

This is the list of tasks that we had for each other when we started the project:

1. Define a basic setup to play Sokoban manually
2. Implement a basic BFS as a Skeleton
3. Develop an A^* algorithm over the BFS Skeleton
4. Create a Heuristic and Deadlock class
5. Improvements on A^* (deadlock detector, pruning techniques, etc.)
6. Results and Performance
7. Final Report writing

Our group was formed roughly around the time when the course had covered Search algorithms and Games. We came to the conclusion that Game Playing Algorithms such as Alpha-Beta Pruning are not exactly suitable for a game like Sokoban, because Sokoban has no obvious adversary to the player. This is why we decided to adopt a search algorithm for solving the game.

To summarise, we started with building an implementation of the Sokoban game, which can generate levels using user-defined specifications and can carry out user-defined instructions. Having such a widget allowed us to monitor the performance of our algorithm in a step-by-step fashion, and it also helped us to fine-tune it later on. Once example of this fine tuning is that in the later stages of development, we discovered new types of Deadlocks (defined in later sections) that we had previously ignored.

In the earlier stages of development, we noticed the structural similarity between Uninformed and Informed Search algorithms. Noticing this similarity allowed us to start with a quick prototype of Breadth-First-Search algorithm which served as a foundation on which we build and the current A^* algorithm. The breadth-first-search algorithm (without pruning) was able to solve several trivial and non-trivial instances of the game, however, it was too slow.

Before we started modifying BFS into an A^* algorithm, we decided to modularize our code, which was previously written in a class-free fashion. Our code was subsequently broken down into:

1. A Board class: Whose objects contain information about the game states
2. A Game class: Which allows manipulation of a Board and also contains our search algorithm
3. A Heuristics class: Which calculates various heuristics
4. A PriorityQueue class: For implementing frontiers

3 Algorithms

3.1 A^*-Search

To solve the game of Sokoban, we implemented an A^* algorithm. Recall that A^*-search algorithm is a variant of Best-First-Search which uses an evaluation function $f(n) = g(n) + h(n)$ to order its search space. Intuitively, we understand $f(n)$ to be the estimate value of node n to the goal state, and we understand $g(n)$ to be the cost of going from the starting state to state node n, and $h(n)$ the estimate cost of going from state node n to the goal state (the *heuristic* value). Following [2], in selecting A^* as our algorithm, we considered specifically the following properties:

1. Completeness: the algorithm finds a solution when there is one, and correctly reports failure otherwise.
2. Cost-Optimality: if the algorithm finds a solution, it finds the shortest one.

The main advantage of A^* is that, with a consistent heuristic, it is complete and cost-optimal. Moreover it is optimally efficient in the sense that it will not be less efficient than other informed search algorithms using the same heuristic.

Recall that ([2]) a heuristic is consistent if and only if for every node n and every successor node n' generated by an action a, we have $h(n) \leq c(n, a, n') + h(n')$, where $c(n, a, n')$ is the cost of taking action a at node n resulting in n'. The heuristic we have chosen is the wall-free version of Manhattan heuristic. That is, for each box, and for each coordinate of the box, we first sum over the absolute difference between it and the corresponding coordinate of each goal. For each box we will get such a value, and then we sum over all such values.

To give an overall view of our A^*-search algorithm, we provide the following pseudocode

Algorithm 1: `playAStar()`

input : a Board object, *board*
output: a sequence of action leading to the goal from the starting state, or failure

visited ← the empty list;
frontier1 ← a priority queue, initially empty;
frontier2 ← a priority queue, initially empty;
path ← a priority queue, initially empty;
rootNode ← a deep copy of *board*

check if the input files are well-formed (missing the player location or if the game is already in goal state, for example);
add *rootNode* to *frontier1*, with priority value the heuristic value of the current state;
add *(rootNode.playerCoordinates, rootNode.boxCoordinates)* to *frontier2*, with priority value the heuristic value of the current state;
add `null action` to *path*, with priority value the heuristic value of the current state;

while *True* **do**
  if *frontier1 is empty* **then**
    return "Solution not Found"
  end
  currentNode← POP(*frontier1*);
  (currentPlayer, currentBoxCoordinates)← POP(*frontier2*);
  currentActionSequence←POP(*path*);
  foreach *move* ***in*** *possibleMoves from the current position* **do**
    childNode ← a deep copy of *currentNode*;
    update *childNode* with *move* ;
    if *(childNode.playerCoordinates, childNode.boxCoordinates) not in visited* **then**
      if *childNode is goal state* **then**
        return sequence of moves leading to childNode
      end
      if *childNode is deadlock* **then**
        continue
      end
      Cost ← length of currentActionSequence;
      Heuristic ← `calculate_heuristic`(childNode.playerCoordinates, childNode.boxCoordinates);
      add *childNode* to *frontier1* with heuristic value *Cost+Heuristic*;
      add *(childNode.playerCoordinates, childNode.boxCoordinates)* to *frontier1* with heuristic value *Cost+Heuristic*;
      add *currentActionSequence* + [*move*] to *path* with heuristic value *Cost+Heuristic*
    end
  end
end

The `playAStar()` function starts by creating 3 priority queues, *frontier1*, *frontier2*, and *path* and add the starting board (the current node) to *frontier1*, and the player and box coordinates to *frontier2*, with the estimated value of the current node. Similarly, we push the null action [''] to *path* with its estimated value.

Assuming *frontier1* is not empty (otherwise we return failure), we iteratively search for a solution as follows. We first pop a node (*currentNode*) from *frontier1* and pop a pair *(currentPlayer, currentBoxCoordinates)* from *frontier2*, and pop an action sequence (*currentActionSequence*) from *path*. Then, we assign the variable *possibleMoves* the list of legal moves from the current position and add *(currentPlayer, currentBoxCoordinates)* to *visited*.

Next, we run through the moves in *possibleMoves*. For each move, we create a deep copy of *currentNode* and assign it to the variable *childNode*, on which we will update the game board with the move. If the updated *childNode* has not been visited, then we first check if it's a goal state. If so, we return the sequence of actions leading up to this situation. Then, we perform a deadlock check to see if we are in an unsolvable state of the game (more on this later). If so, we skip this move and go on to the next. If the updated *childNode* is neither a goal state nor a deadlock, then we calculate its heuristic value $h(n)$ and cost $g(n)$. Then we append *childNode* to *frontier1* with priority value $g(n) + h(n)$. Similarly, we append the pair containing player and box coordinates of *childNode* to *frontier2* with priority value $g(n) + h(n)$. Finally, append the sequence *currentActionSequence* + [*move*] to *path* with priority value $g(n) + h(n)$. This finishes the description of our algorithm. In sum, we iteratively try each legal move in a current state, estimate the value of the result, then go on to set current state to the next "most promising" state.

3.2 Heuristics

With a consistent heuristic, A^* is the "optimal" search algorithm in the sense of the previous subsection. Since we chose A^* specifically to exploit this feature, it is important that the heuristic we used, the Manhattan distance, is consistent. We show that it is, assuming uniform cost of each move.

Claim. Manhattan distance heuristic is consistent.

Proof. If an action results in no boxes being moved, then the successor node n' has the same heuristic value as the current node n, and since actions have uniform costs (say 1), $c(n, a, n') + h(n') = 1 + h(n) > h(n)$. If a box is moved as a result, then without loss of generality we may assume its Manhattan distance is decreased as a result (otherwise the inequality holds trivially), but since a box can only be moved one unit distance at a time, this decrease is at most 1. That is, $h(n) \leq h(n') + 1$. But our cost is uniformly 1, so $h(n) \leq h(n') + 1 \leq c(n, a, n') + h(n')$. This finishes the proof.

For the sake of completeness, we note that in the actual code, we have allowed the choice of heuristics as a parameter. That is, the user can specify whether to use the Manhattan distance as a heuristic, or the Euclidean distance. In our submission we have opted for Manhattan distance as the default heuristic. A rough justification might be this: we aim to have a heuristic that is as close to the actual cost as possible, and since Euclidean distance between a box and a goal is always shorter than the Manhattan distance, and the actual move sequence required to move the box to the goal will be longer than the Manhattan distance (because boxes cannot be moved diagonally in 1 step). Therefore, Manhattan distance will be closer to the actual number of moves required. This intuition is confirmed, for example, by the results tabulated in [1], where Manhattan distance consistently outperforms Euclidean distance in their A^* implementation.

3.3 Deadlock Detector

In the description of our algorithm, we referred to an operation of checking whether a state is a deadlock or not. To be precise, a deadlock refers to a state of the game where it is no longer possible to move all the boxes to storage positions. If such a situation occurs, then it is no longer useful for the search to continue down its successors, for the obvious reason that the game becomes unsolvable from that point on. For this reason, we have built in deadlock detector functions to check if a node is a deadlock. The deadlock detectors are hand-coded to recognize a few common deadlock patterns. This prunes the search tree.

The most common deadlock is when a box is cornered by walls. If a box is cornered by walls, then there is no way to rescue that box, so this counts as a deadlock. In our code, this situation is called a *corner deadlock*. This is checked in the obvious way, by checking, for each box, whether it has two walls next to it, in orthogonal directions.

Meanwhile, some boxes which are not currently cornered might inevitably become cornered. For example, if a box has hit a horizon of walls and therefore can only be pushed along the direction that horizon, and yet the only objects on that path are walls. This is called *pre-corner deadlock* in our code. This is checked by seeing if a box has hit a horizon of walls, and if so, whether there's a storage location available along that horizon. If not, this counts as a deadlock. This deadlock detector was later modified to handle to more general

situation, where a box has hit a horizon of walls and there is a storage location available along the same horizon, but there is a wall in between that storage location and the box. In the implementation submitted in the draft design, this situation was considered to be solvable by our pre-corder deadlock detector, when it is in fact not. This observation only occurred to us later, and in our final submission we have updated the detector to prune such a situation.

A third possibility is when more than four boxes form a rectangle. In this case, the rectangular group of boxes is "rigid" in the sense that none of them can be moved. This results in another deadlock, which we called *square deadlock*. This is checked by considering, for each box, whether there are boxes to its right, lower right, and bottom. Below we have included a few illustrations of some of the deadlock scenarios.

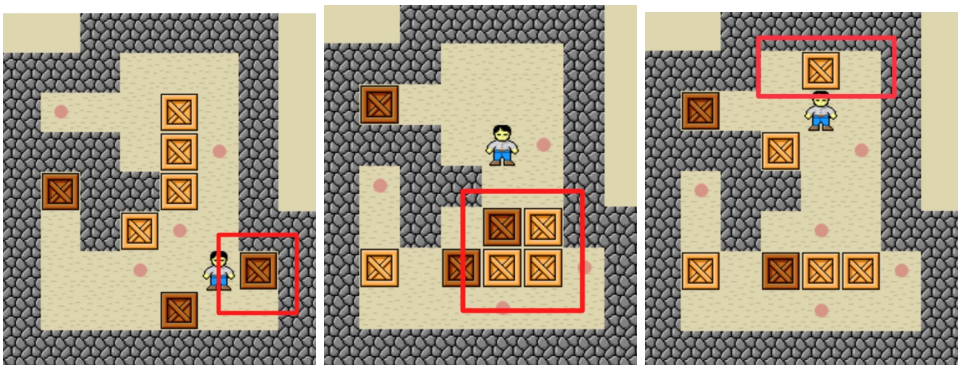


Figure 1: Example of a corner, square and pre-corner deadlocks

4 Results

4.1 Testing on different inputs

We have a total of 22 inputs on our project. To make the process of testing easier, we generated a file that converts files that have the board drawn to the *sokobanXX* that was asked to be the input for our agent. Inputs from 02 to 10 were not considered as they were intractable for our A^* algorithm (same happened for inputs 11, 13 and 14).

The ones considered for our results are the following:

Figure 2: Inputs 00, 01, 12, 16 and 17 respectively

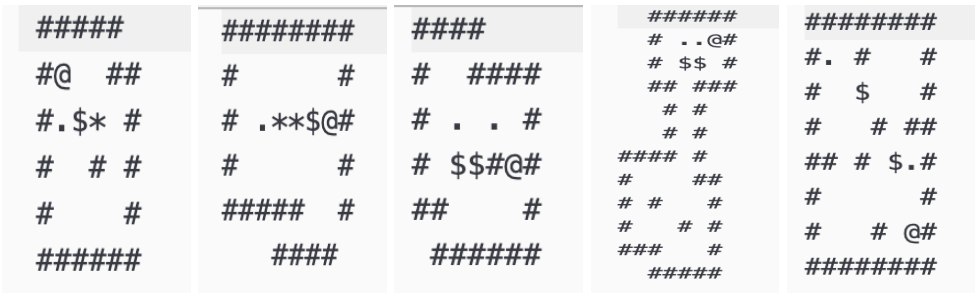


Figure 3: Inputs 18, 19, 20, 21 and 22 respectively

4.2 Results for Manhattan Distance

Sokoban Board	Number of Boxes	Solution found?	Time	nMoves	Generated Nodes	Repeated Nodes
00	1	True	0.001 s	1	1	0
01	3	False	X	0	X	X
12	2	True	0.0488 s	9	131	47
16	1	True	0.17 s	19	576	326
17	2	True	0.41 s	23	1134	501
18	2	True	0.16 s	34	541	269
19	3	True	4.12 s	23	8757	4099
20	2	True	0.13 s	17	399	189
21	2	True	4.21 s	97	6515	3426
22	2	False	X	X	X	X

Sokoban Board	Frontier Length	Deadlocks found	Branching Factor
00	1	0	1
01	X	X	X
12	36	3	3
16	10	3	3
17	256	8	4
18	12	26	3
19	1271	294	3
20	25	17	3
21	163	109	3
22	X	X	X

4.3 Results for Euclidean Distance

Sokoban Board	Number of Boxes	Solution found?	Time	nMoves	Generated Nodes	Repeated Nodes
00	1	True	0.001 s	1	1	0
01	2	True	240 s	0	30000	25000
12	2	True	0.13 s	9	172	68
16	1	True	0.18 s	19	598	341
17	2	True	11.85 s	21	16358	8771
18	2	True	0.18 s	34	567	284
19	3	True	44.59 s	23	38177	20365
20	2	True	0.17 s	17	535	265
21	2	True	4.54 s	101	7676	4095
22	2	False	X	0	X	X

Sokoban Board	Frontier Length	Deadlocks found	Branching Factor
00	1	0	1
01	45000	15000	3
12	36	8	3
16	9	3	3
17	1641	164	3
18	11	27	3
19	2552	1560	3
20	20	23	3
21	110	117	3
22	X	X	3

5 Conclusions

5.1 General Conclusions

As far as **heuristics** are concerned, if we focus on Manhattan versus Euclidean distance.The former gave us better results in terms of number of nodes generated (which also means less repeated nodes and deadlock conditions) and time to solve. The number of moves is the same for both heuristics in most cases as they are both admissible heuristics and should find the optimal solution. This makes sense (it happens the same for puzzle games) since the Manhattan distance is a better heuristic because it represents the minimum number of moves to place one box to a store Location (and as it does not care for walls, it is even lower than the number of pushes) compared to the euclidean distance which is realistic as we cannot move the boxes diagonally.

We experimented with several other heuristic options. We generated two heuristics called "manhattan2" and "euclidean2" which are described on our Algorithms section. They did not work as expected (their time results were worse on both cases to their original versions), mainly because there are sometimes where two boxes have the smallest distance for the same storage location. Moreover, we created a third modification of Manhattan, on our code is the function "compute_manhattan_modified" where we used the normal Manhattan distance but we also added the distance of the player to the closest box, which made the agent to get closer to boxes instead of wandering around the game board. The results of this heuristic were worse than normal Manhattan but on input01 it made great advancements compared to the normal version, as we found the following state:

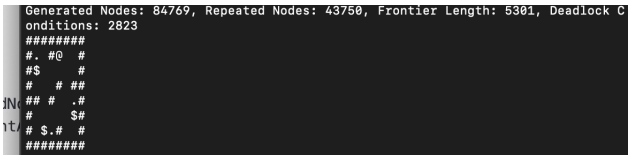


Figure 4: Internal Node of Input01: very close to solving it

But it ended up giving us the output of 'SOLUTION NOT FOUND'. This means that probably some of the deadlock checking conditions are making the code skip some important nodes, and trying to check so would mean taking the deadlock options out and that would cause longer time to go through the search space which makes very difficult to find the root cause.

A key aspect that we realized while checking the internal nodes of the solvable boxes was that our A^* algorithm does not prefer states where a box is already on a store location (probably an heuristic that was better existed) because those boxes were moved. That made us realize that computing the Manhattan distance of one box to all of the store locations was a problem and "manhattan2" should have solved it but it did not as we commented previously.

Sokoban is a very complex game, because the player dooes not only have walk through the board, but also needs to push the boxes. This means we need to go back to different positions of the board after moving a box, and for every possible step, we evaluate its directions which makes the search space enormous.

Furthermore, we also tried to generate a heuristic that computes the number of pushes. This heuristic in itself is equivalent to solving the game of Sokoban so we dropped the idea. We considered another heuristic that computes the distance of one box to the closest store location, meaning one box will have the distance to the closest store location and no other box will have that position. Time constraints did not give us time to finish it but it could be a promising solution.

5.2 Future Directions

Some future considerations to improve this project are to finish the push heuristic and find more complex deadlock detectors. Another idea we had to improve this project even further was to solve the problem of with just one box and one store location while treating the rest of the boxes as walls.

References

[1] Anand Venkatesan, Atishay Jain, and Rakesh Grewal. "AI in Game Playing: Sokoban Solver". In: *ArXiv* abs/1807.00049 (2018).

[2] Stuart Russell, Peter Norvig, et al. *Artificial intelligence: a modern approach*. Pearson Education Limited, 2020.