AI Sokoban Solver: CS271 Project Report

Zhenhan Li (44129905), Yue Zeng (36487716), Yang Jiao (59085058)
University of California, Irvine

***Abstract*—Artificial Intelligence is developing in an unbelievable speed in this decade. Game AI, which relies heavily on AI algorithms is also improving significantly in this trend. With the assistance of deep learning techniques, AlphaGo and its successor AlphaZero conquered the boardgame—Go, which was dominated by human for centuries. In this paper, we would like to focus on a simpler traditional game, Sokoban. We will discuss the ways to model this problem and try to explore the possibilities to solve this problem using different AI algorithms.**

***Keywords*—AI, Sokoban, graph search algorithms, neural networks,**

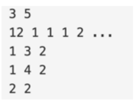
I. Introduction

Sokoban is a type of puzzle game which was created by Hiroyuki Imbayashi in Japan in 1981. In this game, the player controls a warehouse keeper and tries to push boxes to expected positions. The design of minimum moves and how to avoid dead ends are two imperative factors that determines the difficulty of the game. To analyze this puzzle, we will first discuss how it is modeled as a graph searching problem and how a status node are formed in section II. In section III, we will introduce available searching algorithms in this puzzle. In section IV, we will discuss pruning techniques that can be applied to this problem. In section V, we will compare the results of different algorithms and give possible explanations. In section VI, we will discuss the application of convolution neural networks and the analysis of its respective results. In section VII, we will display our results for the benchmark problems. Lastly, in section VIII, we will summarize the work and give comments on different approaches.

II. Problem Modeling

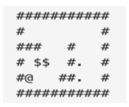
A. Problem Statement

A formal input (*Fig. 1*) of a Sokoban puzzle is a text file which consists of a few lines indicating the coordinates of walls, boxes and storage positions. One can transfer a formal input to a 2D map (*Fig. 2*) for an intuitive visualization, where '#' stands for walls, '$' stands for boxes, '.' stands for storages and '@' stands for the warehouse keeper.



```
3 5
12 1 1 1 2 ...
1 3 2
1 4 2
2 2
```

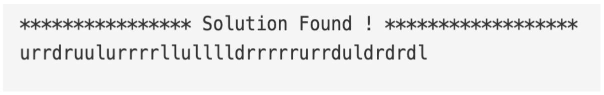
Figure 1: formal input of a Sokoban puzzle



```
############
#          #
###   #    #
# $$  #.   #
#@    ##.  #
############
```

Figure 2:2D map of a formal input

For every puzzle, the program is supposed to return a string (*Fig. 3*) that represents the sequence of movements that helps the warehouse keeper to push all boxes to the expected positions.

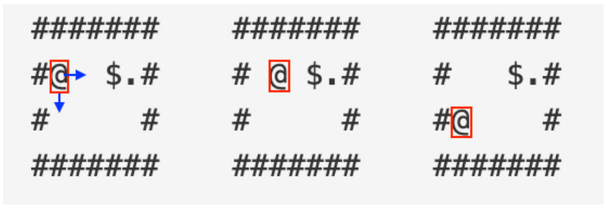


```
**************** Solution Found ! ******************
urrdruulurrrrllullllldrrrrrurrduldrdrdl
```

Figure 3: sample output of a Sokoban puzzle

B. Constructing States of a Probelm

For each game, we may notice that the solution of the puzzle is a collection of consecutive single-step moves. The difference between two states is the position of player and boxes. Hence, we define a state of a problem as the collections of player and boxes positions. Since every state is determined by a move of the warehouse keeper, a state potentially has four neighbors/successor nodes that can explore in the next move. A valid move contains situations: (1) only the keeper moves, and he moves to an empty place within the walls limit. (2) The keeper moved and pushed a box to a new place. Both the keeper and the box do not violate walls and box restrictions.



```
#######     #######     #######
#@   $.#    # @ $.#     #   $.#
#     #     #     #     #@    #
#######     #######     #######
```

Figure 4: A state of problem and its successors

C. Modeling as a Graph Search Problem

Generally, a graph search problem is composed by an initial state, actions, end state, goal test, and path cost. For a Sokoban puzzle: (a) the initial state provides the positions of the keeper and boxes; (b) an action is represented by the changes of the positions of the keeper and boxes; (c) a goal test will ask the system to execute a matching between boxes and their targets; (d) the path cost is the number of moves starting from the original state to current state. These fit the principles for constructing a Sokoban puzzle. Starting from the initial state, we will be able to explore its successors and execute recursions to reach the final goal. Comparing with human's intuitive solutions while viewing the puzzle, this model has some drawbacks in respect to space consumption. Inspired by this point, we adopted different heuristic functions in algorithm trials.

III. Algorithms

A Sokoban puzzle may have multiple solutions. Based on the properties of a map, solutions can vary significantly in the length of steps or searching time. To reach an optimal algorithm, several factors need to be taken into account. (1) The length of total move steps. For some algorithms, even though the box is close to the warehouse keeper, it may still generate a solution which orders to keeper to go around a big circle before he makes an effective move. Such inefficient outcome matters more when the map is large, which makes it easier for the system go out of memory space. For some games, there are restrictions on move length or time. For this reason, it is imperative for an algorithm to finish the task within fewer moves. (2) The search time for a solution. For all graph search problems, search time is a typical and important factor that measures the quality of an algorithm. It is true that people can use tree search to find a solution and tree search relatively saves much space. However, the time complexity will be a disaster and it will fall into infinite loops if there's no available solutions. It is also meaningful in reality because this-real time game expects the player to finish the puzzle using as less time as possible.

To solve this puzzle and analyze for an optimal algorithm, we applied various algorithms to different sizes of maps and different numbers of boxes. We tracked the result of move steps and respective running time for later analysis. Successively, we applied Breath First Search (BFS), Depth First Search (DFS), Iterative Deepening Search (IDS), Uniform Cost Search (UCS), Greedy Search and A* search algorithm.

A. Breadth First Search (BFS)

Breadth First Search (BFS) is an algorithm for traversing or searching a tree or a graph structure. Generally, a search begins at a node which marked as root, and start exploring all its neighbors/successors at the same level before it moves on to the states on the next level. A BFS search is always complete in the sense that applying its level-by-level check, it will never miss a solution. However, since the time and complexity for BFS is $O(b^d)$, the program may take much time to process and can easily go out of heap memory space. In this problem, the branch factor of a state is 4 (up, down, left, right). On every level, it will check all the successors of a state and keep traversing until a solution is found or all nodes have been checked.

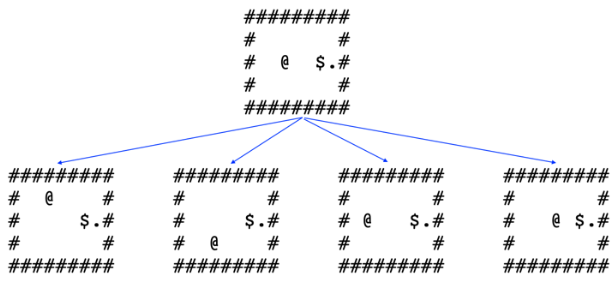


Figure 5: A sample BFS search

B. Depth First Search (DFS)

Similar to BFS, the Depth First Search (DFS) algorithm starts searching at a root node and explores as far as possible along one of its successors before backtracking. This algorithm consumes $O(b^m)$ time but occupies only $O(bm)$ space, which is linear. In practice, DFS results in much faster searching compared with BFS. However, it usually generates a long solution which contains many moves. Moreover, the reason for DFS performs faster than BFS is that m (maximum search depth) is usually smaller than d (graph depth). A DFS search may fall into infinite loop in one branch even though the solution is at the root of the next branch. The following figure gives a rough idea about how DFS works for a Sokoban puzzle.

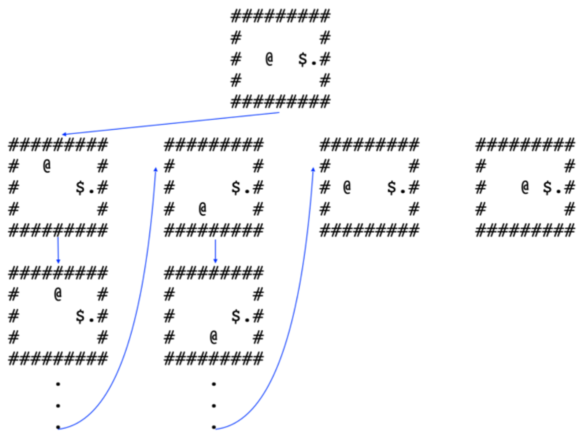


Figure 6: A sample DFS Search

C. Iterative Deepening Search (IDS)

Iterative Deepening Search (IDS) is a depth-limited version of depth-first search with gradually increasing depth until the maximum depth is reached or the goal is found. It is generally the preferred uninformed search because it inherits the memory advantage of depth-first search and meanwhile has the completeness property of breadth-first search. With a limit in depth, IDS avoids the situation that DFS keeps searching in a very deep status and avoids returns a solution with long moves. In practice, we set the default maximum depth for searching as 50 steps. It will gradually increase the maximum depth by 10 every time it reaches the limit.

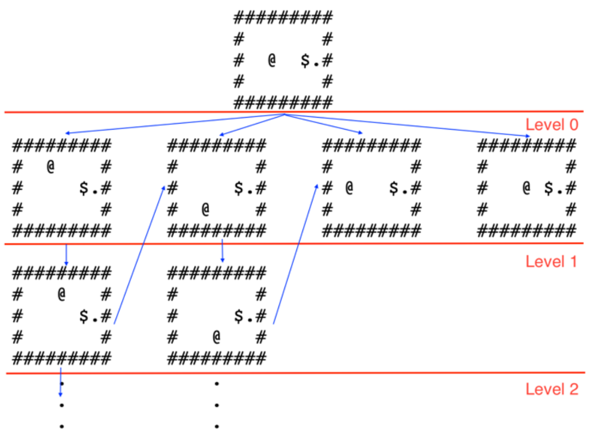


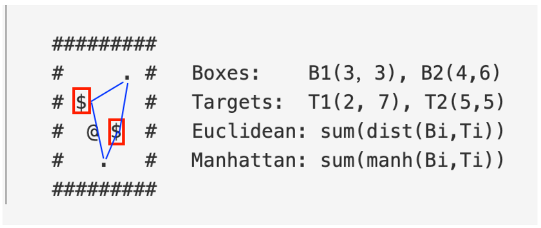
Figure 7: A sample IDS search

D. Uniform Cost Search (UCS)

Uniform Search is a tree/graph search algorithm related to breadth first search. However, different from BFS which adds a state's successors sequentially, a uniform cost search keeps a priority queue which puts states with fewer costs in front. BFS will yield a path to the goal with least number of steps, whereas UCS will result in a path that has lowest cost. However, for this Sokoban puzzle, UCS is equal to BFS since the cost for every edge is the same. We used a priority queue to keep a set of states and sort the states in the order of their movement length. In practice, the result testified our conjecture. We noticed that actual running time of UCS is slightly shorter than that of BFS, this may due to the inner structure of a LinkedList and a Priority Queue.

E. Heuristic Functions

A heuristic function is a function that calculates an approximate cost from initial state to the goal state. Depends on the setup of the games, heuristic functions are different from case to case. It is common for one game to have more than one heuristic functions. A good heuristic would tend to precisely estimate the distance from a state to the goal state, indicating the direction of the optimal path and reduce the complexity of a searching algorithm. For Sokoban puzzle, we adopted two possible heuristics: (1) Euclidean distances and (2) Manhattan distances. Euclidean distances heuristics calculates the sum of Euclidean distances between boxes and targets, whereas the other calculates the absolute values of the differences between boxes and their targets.



```
#########
#      . #    Boxes:     B1(3, 3), B2(4,6)
# $      #    Targets:   T1(2, 7), T2(5,5)
#   @ $  #    Euclidean: sum(dist(Bi,Ti))
#    .   #    Manhattan: sum(manh(Bi,Ti))
#########
```

Figure 8: Heuristic Functions

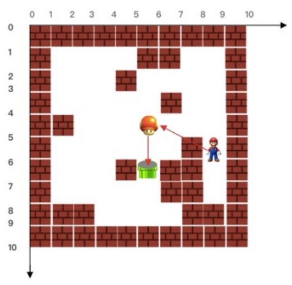


Figure 9: Euclidean heuristic

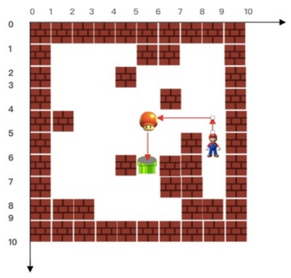


Figure 10: Manhattan heuristic

F. Greedy Best First Search

Greedy Best-first Search utilizes the information that we have from the heuristic functions, maintains a priority queue as its frontier and expands the nodes that is closest to the goal node in the frontier. The major drawback of Greedy Best-first Search is it can be easily stuck in infinite loops. For this reason, it is neither complete nor optimal. And its performance depends on the heuristic function that it works with. If the heuristic function is not accurate, its worst case is of a complexity $O(b^m)$, whereas in the best cases, it can reach a time complexity of $O(n)$. In the Sokoban puzzle, we adopted two heuristics for greedy algorithms, and they have their advantages respectively in different maps. One is Euclidean distance, which is a consistent heuristic for 1-box puzzle, always gives the minimum distance between two tiles. Another one is the Manhattan distance, which gives reasonable estimate assuming there are no obstacles between a box and its target.

G. A Algorithm*

A* is a graph traversal and path search algorithm which is often used in search problem due to its advantages including completeness and optimality. As another informed search algorithm, A * search not only considers the path cost from a node to the goal but also sums it up with the actual cost to the current node. The A* search utilizes f(n) = g(n) + h(n) as an estimate for the cost towards a goal, where h(n) is the estimated cost of the cheapest path from n to the goal, g(n) is the path cost to reach the current goal and f(n) is the value of the heuristic function. Similar to the UCS, A* has a sorted frontier based on the cost but the difference is that A * has a n estimation of where the goal is located so it has a better performance if the heuristic function is good. In order for the A * to be optimal and complete, the heuristic function has to be both optimal and constant.

IV. PRUNING TECHNIQUES

Generally, the difficulty of Sokoban puzzles increases with maps size and the number of boxes. As for some algorithms which grows exponentially in both space and time, it is of great importance to prune the search space. In order to minimize the search space, we designed deadlock detections which executes before a search of a solution. The detector takes initial games state as an input and crosses out all box positions that will lead to a dead end. The idea behind the deadlock detection is similar to Minimum Remaining Value (MRV) from the constraint satisfaction problem.

Our deadlock detection function mainly deals with two types of deadlocks. (1) Corner deadlocks. A deadlock corner is a non-available storage gird which have at least 2 adjacent walls (e.g. up and left). As figure 1 shows below, the three corner spots marked with red "X" is corner deadlocks. Intuitively, if a box reaches these positions, it will never come out in the following moves.

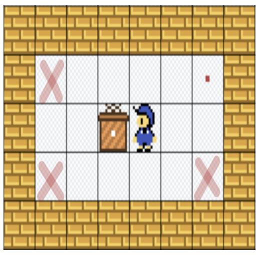


Figure 11: Corner detection

(2) Boundaries deadlocks. Boundary grids are the grids that have only one wall next to it. If a box falls onto such grid, it will only be able to move horizontally or vertically. However, not all boundary grids are deadlocks. If a storage grid is on the same line as a box, it is still possible for the box to reach the target. An instance of how we define the boundary deadlocks and non-deadlocks is shown in figure 2. Grids with red "X" are boundary deadlocks, non-deadlocks are marked as green "O". From the figure, we can tell that if there is a storage grid next to the boundary, all the boundary grids in the same direction as the walls will be eliminated from the deadlock list. In addition, some walls convex to the outside (as the left wall shown is figure 2), and points on those boundaries cannot directly fail the game or even may be useful for the player to find the solution, so we also take them out from our deadlock list.

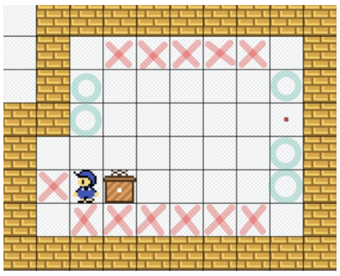


Figure 12: Boundary deadlock detection

V. Performance Analysis

For all applied algorithms, we adopted two metrics for analyzing their performances. We first compared the number of states each algorithm explored, and then compared the time each method used when searching for the solution. We designed a Sokoban map generator that can spawn available Sokoban puzzles in large numbers, and tested the algorithms in size n map with 1 to 3 boxes attached. Although intuitively the complexity and consumed time is supposed to increase with the size of the map, the actual running time does not always follow this trend, because for a map in same size, the complexity can still be very different. Another issue occurred because some algorithm fails to provide solutions when the size of a map increases over 12 when there are more than two boxes. To ensure a intuitive and consecutive observation, we finally generated maps ranging from 9*9 to 15 * 15 with one single box to compare the performances and trends of different algorithms.

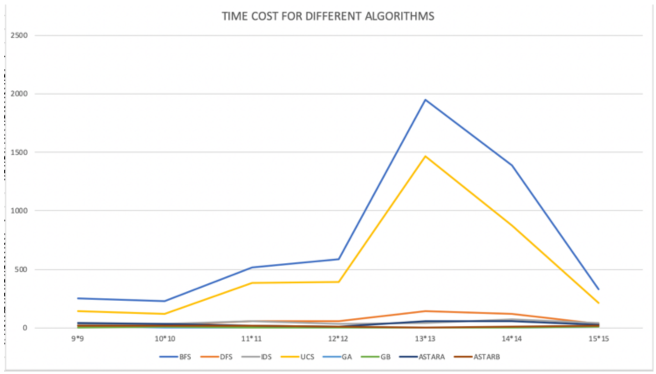


Figure 13: time cost graph for algorithms

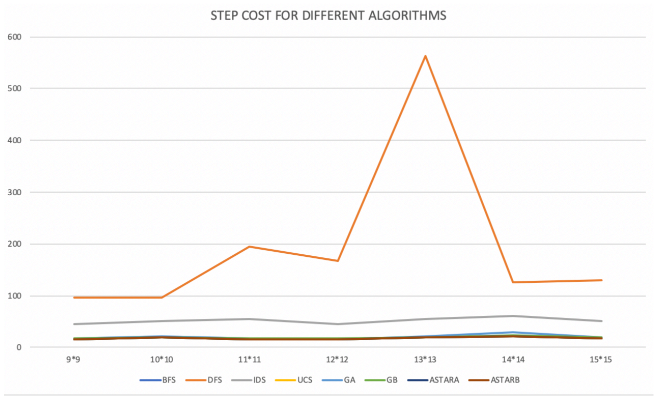


Figure 14: move length for different algorithms

As shown in the above two figures, the comparison s between algorithms other than BFS and DFS are rather ambiguous, and the between step size/time and map size are not obvious.

In order to compare the performance of each algorithm and heuristic functions more intuitively and efficiently, we used 15 10x10 maps with 2 boxes to collect the data. We choose 10x10 because it is a relatively large map which can be challenging in some cases but still overall solvable. In the 15 maps that we tested on, BFS and UCS are able to solve 94% of them (we terminate the program after 15 seconds). And other algorithms are able to find at least find a solution for all the maps. We used the “steps”(it takes to solve the problem) and “time” to measure the performance.

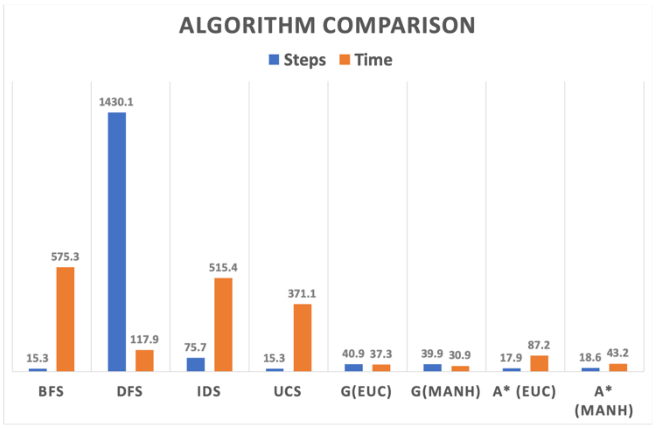


Figure 15: Performance Comparison

As expected, uninformed searches, including Greedy best-first search and A*, is better than uninformed searches. In general, A* has the best performance. However, Euclidian takes slightly less steps and time than Manhattan. Greedy Best-First search is the fastest search among all algorithms. As a trade-off, it fails to find the optimal or “closed-to-optimal” solutions. On average, the step cost of the greedy best-first search doubles the step cost of the A*, which is reasonable since g(n) is not involved.

A* search is expected to provide both complete solutions in graph search algorithms if the heuristic functions never overestimate the true costs. Moreover, if the sum of the true cost and heuristic is always non-decreasing along any path towards the goal (i.e. consistent), the algorithm is supposed to yield an optimal solution. At the initial stage, we expected the A* algorithm to provide optimal solutions since both Euclidean and Manhattan distances offers consistent heuristics. However, in practice, it is testified that these heuristics are not consistent and thus not optimal. This is due to the fact that when the number of boxes increases, the heuristics becomes a summation of every single heuristic between boxes and their respective storage targets.

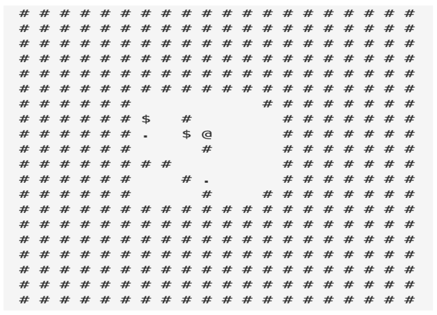
As for uninformed search algorithms, we expect BFS and UCS to always find the optimal solution (if there is one) because the true cost of each step is constant and this has been proved by our test cases. DFS has the worst performance as the step cost is extremely higher than other algorithms. However, even though it requires more computation based on the number of the steps, the running time is in fact lower than BFS and UCS. As an improved version of the DFS, it combines the advantages of BFS and DFS together. Therefore, IDS is both complete and space efficient.

VI. Application of CNN on Sokoban Puzzle

Neural networks have developed rapidly these years due to the boosting of hardware upgrade and optimization. It tends to have tremendous advantage in regression and classification problems. The structure of a Sokoban problem offers us the possibility to utilize neural network to solve this problem. At every stage, a warehouse keeper can simply moves towards four different directions, this can be modeled as a classification problem. Moreover, since the structures of a 2D map has critical influence on the result of moving, the input is supposed to be a 2D matrix representing the types and locations of different tiles. After training and feeding the trained model, the system will finally a 4-dimensional matrix indicating the predicted direction of next move. Due to the limit of our hardware and algorithm efficiency, we limit the input of a Sokoban map within a square with length 20 bricks. For a map that is smaller than 20*20, we filled the blank spaces with walls. We assigned the tiles with following numerals:

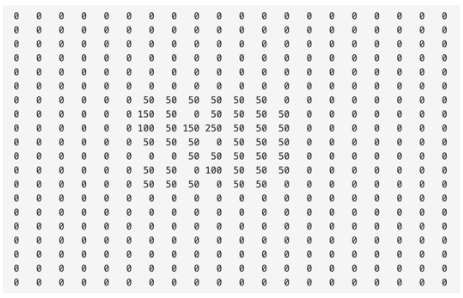
walls	boxes	storages	player
0	150	100	250

Figure 16: Sokoban map value assignment



```
# # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # # # # # #
# # # # # #             # # # # # # # #
# # # # # # $   #         # # # # # # #
# # # # # # .   $ @       # # # # # # #
# # # # # #       #       # # # # # # #
# # # # # # # #           # # # # # # #
# # # # # #     # .       # # # # # # #
# # # # # #       #     # # # # # # # #
# # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # # # # # #
```

Figure 17: 2D map normalization



```
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0  50  50  50  50  50  50   0   0   0   0   0   0   0   0
   0   0   0   0   0   0 150  50   0  50  50  50  50   0   0   0   0   0   0   0
   0   0   0   0   0   0 100  50 150 250  50  50  50   0   0   0   0   0   0   0
   0   0   0   0   0   0  50  50  50   0  50  50  50   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0  50  50  50  50  50   0   0   0   0   0   0   0
   0   0   0   0   0   0  50  50   0 100  50  50  50   0   0   0   0   0   0   0
   0   0   0   0   0   0  50  50  50   0  50  50   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
```

Figure 18: CNN sample input matrix

We first ran A* algorithm to get a solution, and at every step of the solution, we combines the move with the map matrix to form a input-output pair. We passed the input into a CNN model with following configurations:

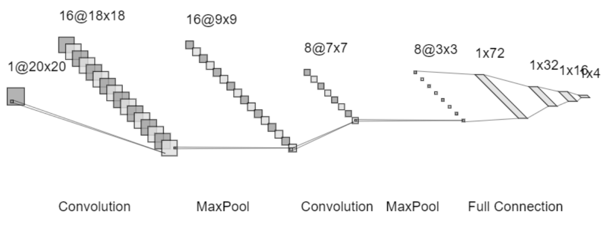


Figure 19: neural networks structure

We generated 50,000 samples and splitted 30,000 as training data, 10,000 as validation data, and 10,000 as test data. During training, we get following results:

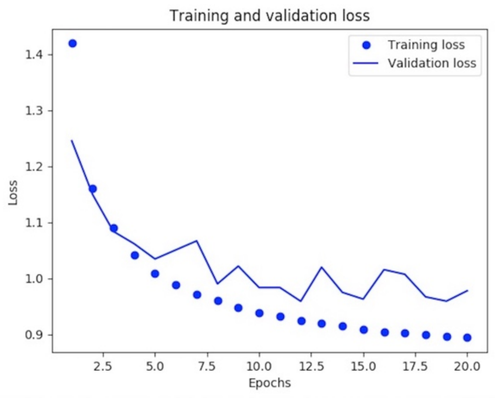


Figure 20: training & validation loss

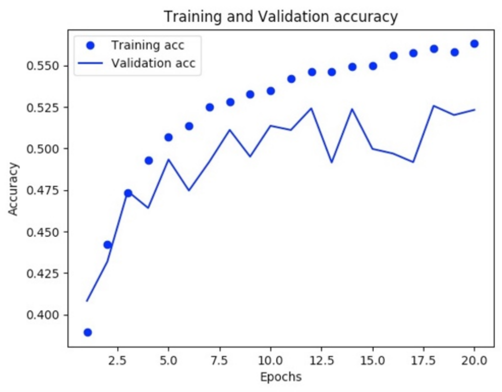


Figure 21: training & validation accuracy

For test data, we get an average prediction accuracy 0.53. This is not much better than random guess which also has an accuracy equals 0.25. Hence, we have to give up this idea.

VII. Results for Benchmarks

Input	00	01	02	03	04	05a	05b	06a
Steps	1	23	30	34	40	19	101	38
Time(*ms*)	0	105	3	3	5	2	222	2

Input	06b	06c	07a	07b	08	09	10
Steps	75	NA	75	NA	NA	NA	NA
Time(*ms*)	6	NA	58	NA	NA	NA	NA

VIII. Conclusion

Sokoban is a typical and accessible puzzle model for testing AI algorithms. Our practices have proved that it can be well-formed as a graph search algorithm, or a machine learning based classification problem. Among all the search algorithms we have tested, greedy search tends to provide fastest search and A* algorithm provides more optimal efficient solutions. BFS and DFS works well for small and simple tasks, however, BFS will go out of memory heap easily and DFS usually generates very long moves which is rather inefficient. A* algorithm, which is expected to provide the optimal and fastest search, fails to meet our expectation because of poor design our heuristics. It is expected to behave better if more reasonable heuristics are used.

In addition to algorithms, pruning is of great importance in the Sokoban graph search problem. The performance after applying deadlock detectors improved greatly. Although our way to compute deadlocks is inefficient, when compared with searching, the time can be omitted. If we want to pursue further optimization, it is necessary to decrease the search space further to decrease the pressure for every search algorithm.

Finally, although we tried to convert the puzzle to an image classification problem using CNN, it fails to provide accurate predictions. The small size of the figure and bad value assignments prevent the model from learning useful graph structures from a matrix. Even though, we still believe that neural networks can have great impact on Sokoban puzzles.