A Sokoban Solver Using Multiple Search Algorithms and Q-learning

COMP 3211 Final Project - Group 6

Hong Kong University of Science and Technology

GAO, Huaxuan
hgaoab@ust.hk
20328408

HUANG, Xuhua
xhuangat@ust.hk
20329347

LIU, Shuyue
sliuao@ust.hk
20329256

WANG, Guanzhi
gwangaj@ust.hk
20328604

ZHONG, Zixuan
zzhongab@ust.hk
20328800

I. Introduction

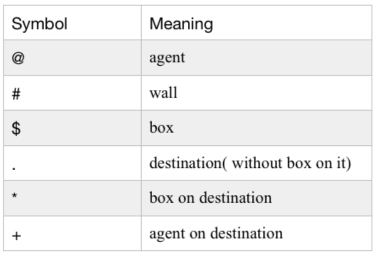
Sokoban(also called warehouse keeper) is a Japanese video game of a type of transport puzzle. In the game, player pushes boxes or crates in a warehouse and trying to get them to storage locations. The rules for this game are very simple to understand, but in some difficult levels, it is hard for human to complete due to the large sum of calculation. Therefore, we try to implement algorithms to solve this game. In fact, it is a challenging task and a number of algorithms have been used to reach the goal. In our project, we have implemented different searching algorithms, and compare their performance in order to find an effective one to solve the game. In addition to searching, we have also tried to solve it using Q-learning. The instruction of how to run our program will be given in the appendix

II. Problem Analysis

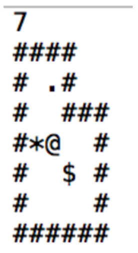
In sokoban, the goal of the game is to move all the boxes to the goals no matter which goals. However, there are serval constraints that player must follow:

1) The box cannot be in a deadlock state(For example, the box is in the corner but the position is not the destination).
2) The player is allowed to move only one box at a time, it cannot move two or more boxes simultaneously.
3) If the player want to move the box down, there must be a path reaching the upper side of the box and there should be a space on the lower side of the box.

The final score is related to the steps player moves. The more steps the player moves, the higher the cost and thus the lower score. Each step lead to 1 cost. So different ways may all lead to the goals but with different scores. The layout of the game is shown in figure1



Symbol	Meaning
@	agent
#	wall
$	box
.	destination(without box on it)
*	box on destination
+	agent on destination



```
7
####
# .#
#  ###
#*@  #
#  $ #
#    #
######
```

Fig. 1. sample layout

III. Challenges

A. Change the game rules into logic

Although the rules of Sokoban is not very complex, it is very hard to convert the rules into logic or algorithm function about which direction to take. Because the game includes many variables and the layout is changing all the time. The constraints of the game should also be considered. So the turning the logic of the game into a program is not a simple task.

B. The design of the State object

Because there may be boxes plus the player in the game, it is challenging to design an appropriate object to store the essential information of the game. These two elements′ positions are changing all the time and the position of the walls are also related to the solution. So if we respectively store the position of player and boxes, it will be complex and hard to use, and may be in deadlock because of the walls.

C. The deadlock detection

As we mentioned before, there may be a deadlock situation that′s impossible for the box to move to its target and the game is over if a box/boxes reach the deadlock situation. So we have to design a function to detect whether the box will go into the deadlock and combine it with the possible direction. In this case, the box during search will never go into the deadlock.

IV. Algorithm Analysis

Before we get into algorithm analysis, there are several concepts that need to be explained.

1) Map: A 2D array store the current level map
2) Open goal: The goal that havent been occupied. Represented by a dot .
3) Closed goals: The goal that havent been occupied. Represented by *
4) State: The states are represented by a string containing the whole layout of the map. Including the location of the walls, player, boxes, open goals, closed goals
5) Valid move: Four directions that the player can move, and if there is a box in front of the player, whether he can move the box
6) Cost: Whenever the player make a valid move, the cost increase by 1, initially set to 0

7) Solution: Store the solution with ‘u’,‘d’,‘l’,‘r’ representing four directions, the cost is the length of the solution

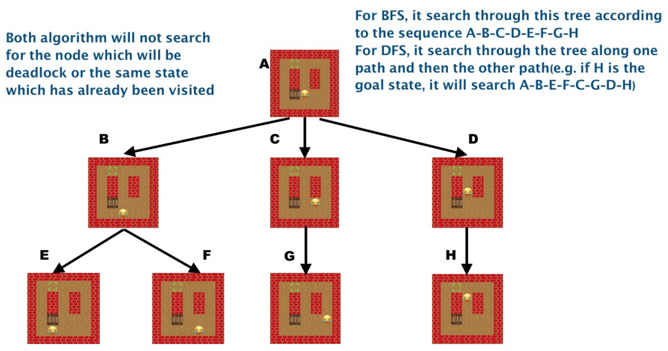
A. Breadth first Search [1]

Starting from the root node and put it in our fringe, we check every valid children (move direction = left/right/up/down) by a helper function isValid(), which returns true if the move is valid. If the child is in the final state(all boxes are in goal), we return the path to the final state (represented by a string recording directions). If we havent reach the goal and we havent been in this state before, we just add the child to our fringe and keep repeating popping from the fringe in a First In First Out(FIFO) manner.

B. Depth First Search [2]

Similar to BFS, but we pop the fringe in a Last In First Out(LIFO) manner. We use queue in this project to fulfill DFS, but actually it is just like a stack and every time we pop out the top object in the stack and do searching on that object.

Figure2 shows a simple search tree of BFS and DFS, and we can observe from the diagram how BFS and DFS search in this project.



Both algorithm will not search for the node which will be deadlock or the same state which has already been visited
For BFS, it search through this tree according to the sequence A-B-C-D-E-F-G-H
For DFS, it search through the tree along one path and then the other path(e.g. if H is the goal state, it will search A-B-E-F-C-G-D-H)
A
B
C
D
E
F
G
H

Fig. 2. a simple example search tree for BFS and DFS.

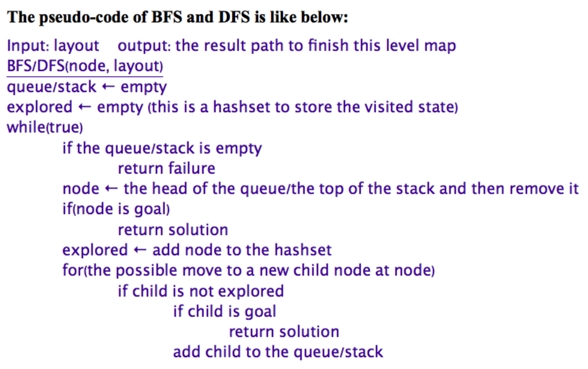
Figure3 shows the pseudo-code of BFS and DFS, in which we can clearly observe that we won’t search for the visited nodes and what the search sequence depends on.

C. Uniform Cost Search

We implement this by a priority queue, using the cost as the key. If the childs state is the same as some state stored in the fringe, we remove that state and add the child to the fringe. So every time we pop the queue depend on the priority(cost).

D. Greedy Search with Open Goal as Heuristic

Similar to UCS, but we use the number of open goal as the key. The less open goal a state have which means there are less goals to acheive, the earlier it will be popped.



The pseudo-code of BFS and DFS is like below:

```
Input: layout    output: the result path to finish this level map
BFS/DFS(node, layout)
queue/stack ← empty
explored ← empty (this is a hashset to store the visited state)
while(true)
        if the queue/stack is empty
                return failure
        node ← the head of the queue/the top of the stack and then remove it
        if(node is goal)
                return solution
        explored ← add node to the hashset
        for(the possible move to a new child node at node)
                if child is not explored
                        if child is goal
                                return solution
                        add child to the queue/stack
```

Fig. 3. The pseudo-code of BFS/DFS

E. Greedy Search with Manhattan Distance as Heuristic

Similar to UCS, but we use the Manhattan Distance as the key. Here the manhanttan distance refers to the sum of distance from boxes to the goals which means the smaller distance it is, the closer state to the goal. The smaller MD a state have, the earlier it will be popped.

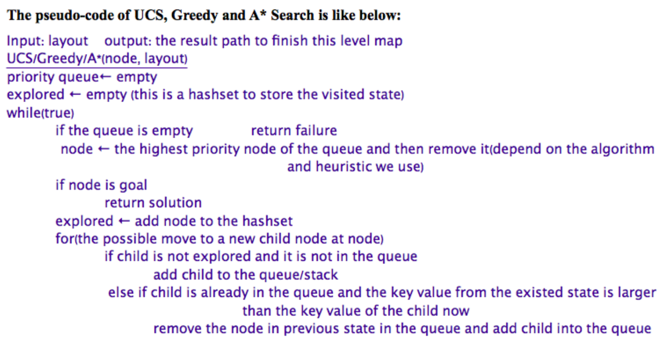
F. A* Search [3] with Manhattan Distance as Heuristic

Similar to UCS, use Manhattan Distance + cost as the key.

G. A* Search with Open Goal as Heuristic

Similar to UCS, use number of open goal + cost as the key.

Figure4 shows the pseudo-code of the priority queue search(UCS,Greedy and A*), it is a little bit different from the previous one which will compare the ”key” and move the larger one from the queue and add the smaller one if they have the same state.



The pseudo-code of UCS, Greedy and A* Search is like below:

```
Input: layout    output: the result path to finish this level map
UCS/Greedy/A*(node, layout)
priority queue← empty
explored ← empty (this is a hashset to store the visited state)
while(true)
        if the queue is empty          return failure
        node ← the highest priority node of the queue and then remove it(depend on the algorithm
                                and heuristic we use)
        if node is goal
                return solution
        explored ← add node to the hashset
        for(the possible move to a new child node at node)
                if child is not explored and it is not in the queue
                        add child to the queue/stack
                else if child is already in the queue and the key value from the existed state is larger
                                than the key value of the child now
                        remove the node in previous state in the queue and add child into the queue
```

Fig. 4. The pseudo-code of UCS, Greedy and A*

H. Q-Learning [4]

We have seen the power of learning in the game Pacman, so after we are done with the searching algorithms, we are interested to investigate how well Q-learning will perform for sokoban. We implemented a new class called QNode to store the state and action. The QNodes and their values are then stored in a HashMap. We designed a relatively simple reward function: when the player reach the goal state, it will receive a reward; when it enter a deadlock, it will get a penalty. A deadlock is the situation when the box in pushed to the corner, or it is push to the wall and it cannot reach any destination. While the player havent reach the goal state, it will keep picking actions randomly. Once the boxes are in deadlock or the goal state is reached, the q-value for states will be updated. But the player will keep performing random action, in order to explore unknown states. Once the training epochs are finished, a policy will be given based on the q-value that it learned.

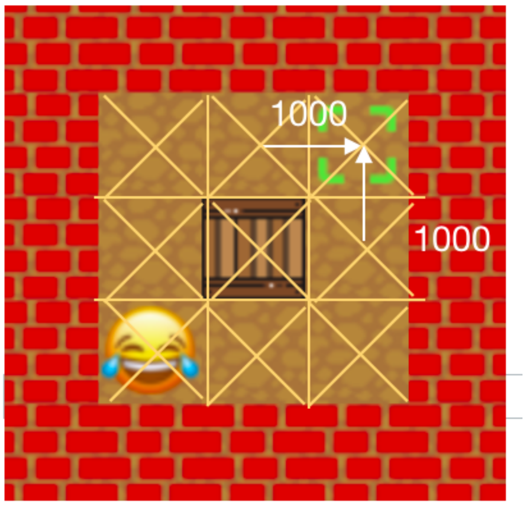


Fig. 5. sample of Q-learning

For example, as illustrated in figure5, the start point of player is on the bottom-left corner and the goal is on the top-right corner. Box is on the middle initially. When performing Q-learning, the player will randomly choose directions to go until it received a reward or enter a deadlock. As shown in the diagram, the Q-value of each box(four directions) will be updated. After the training is done, the player will choose the max Q-value direction to go and finally achieve the goal. If the training epoch is large enough, the Q-value will converge eventually and there should be an optimal solution.

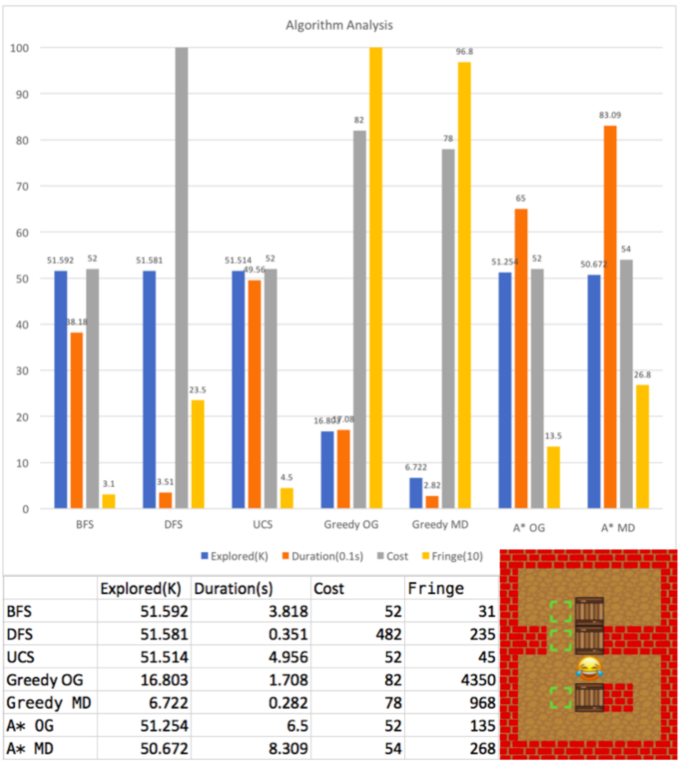
```
pseudo-code of Q-learning
Q-Learning(node, layout)
for(number of trainings smaller than the training epochs)
    while not get to the goal/deadlock
        get a random action from the possible actions and compute the next state
        if next state is deadlock
            update Q-Node value with penalty
            number of training ++
            create a new state to start the while loop
        else
            update Q-node value with the reward
            if next state is goal
                number of training ++
                create a new state to start the while loop again
compute the max Q-value and get the solution to the goal
```

Fig. 6. Pseudo code for Q-Learning

Figure6 shows the pseudo-code of the Q-learning, and we can observe from that it compute the Q-value randomly for a specific training epochs. It will train again until get to the deadlock or get to the goal. And ather the for-loop done, we will get the solution from the current Q-value stored in the Q-Node.

V. Result Analysis

In this project, we have tested a lot of different layouts. In order to illustrate the differences between various search algorithms, we are going to show the result of the same map and compare the results with respect to the number of nodes explored, time duration, fringe size, and the cost of the solution. In order to emphasize the advantages and disadvantages of different algorithms, we choose a significant one to show the result(expect of Q-learning which will be discussed later in the report). The layout is shown below. Besides, we draw a histogram about the serval performances of these algorithms, so that we will have a clearer understanding about the result.



	Explored(K)	Duration(s)	Cost	Fringe
BFS	51.592	3.818	52	31
DFS	51.581	0.351	482	235
UCS	51.514	4.956	52	45
Greedy OG	16.803	1.708	82	4350
Greedy MD	6.722	0.282	78	968
A* OG	51.254	6.5	52	135
A* MD	50.672	8.309	54	268

Fig. 7. Comparison for different search algorithms

A. *Breath First Search*

We can observe from the diagram that BFS has the lowest cost which means that it gives us the shortest path from start to the goal and it is always like this when running different layouts. At the same time, it stores a small amount of nodes in the fringe. Because when running BFS, it explores nodes which are first in the queue and once it finds the solution, it returns immediately. In this way, BFS can always find the optimal solution and explores a lot of nodes which returns smaller amount of nodes in fringe.

B. Depth First Search

In DFS, we get the longest solution among all the algorithms and stores a lot of states in the fringe but it takes a very short time to complete the calculation. Because when the DFS running, it explores the nodes at the top of the stack which is the last node into the stack. So it always keeps exploring the current path until there is no available direction to go and then it change a path to search. By doing this, DFS is prone to find a very long solution since it gives out the first solution it finds along its way down the search tree. It will also store a lot of nodes in fringe in this way.

C. Uniform Cost Search

This algorithm uses priority queue to store the nodes, and as we look at the result, the explored nodes is almost the same as before but with a little bit longer search time and performs other things between BFS and DFS. In this algorithm, it uses cost as a key to pop node off the queue, so we must calculate the cost of each node at first which may take a little bit longer time. In this way, UCS always search the path with the lowest cost so it will always find the smallest cost path to the goal which is an optimal algorithm. However, it search along many nodes before finding the solution so the nodes in fringe is much fewer than DFS.

D. Greedy Search

As we mentioned before, we use two different kinds of heuristic to do the greedy search. The two heuristics use different keys to determine which nodes to be popped off next. These two kinds of heuristic both point to the goalthat is, the smaller it is, the nearest state it is to the goal. So this kind of algorithms performance depends strongly on the layout we input. For some kinds of simple layouts, the greedy may be optimal and it may run in a very short time because the shortest path may be the solution to the goal and we dont need to search anyway. However, in some difficult layouts, greedy may not be such efficient since the solution may be a path with worse heuristic value.

E. A* Search

The solution of A* algorithm depends on how we design the heuristic. If the heuristic is perfect designed, A* must give out an optimal solution with the shortest path to the goal. In out project, we design open goal and manhattan distance as two heuristic. And observing the result above, we find that A* always costs longer time to find the solution since it must calculate a lot and the cost is or near the optimal solution. Besides, the explored nodes is a little bit smaller than BFS and the remaining nodes in fringe is a little bit more. However, according to all these performance of A*, it may be not suitable for us to use A* for searching since the heuristic may not be well designed and it takes longer time.

F. Q-Learning

For the same layout used in search algorithms, the Q-learning with 10000 epochs is not capable of finding out the solution since there are three boxes and it is very easy for the player to get into deadlock. Currently our design for Q-learning can only be used to solve very simple problems involving one box only, when there are two or more boxes in the layout the efficiency of the learning becomes very low. It is very hard for this algorithm to move boxes randomly into the goal. But for the single box situation, this algorithm can get final optimal solution within 1000 episodes. Here we choose a very simple layout to run and compare it with the BFS. According to the result, we can find that the Q-learning cannot perform better than the search algorithms which implies Q-learning may not suit for this problem.

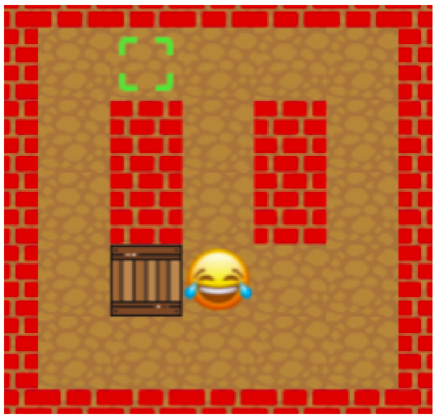


Fig. 8. The layout of Q-learning

For the layout in Figure8, we used Q-learning and BFS to solve it. The time duration of BFS is only 0.009 second. When we use Q-learning, we try to decrease the number of epochs and finally we find that when epoch equals 200, there is a chance for it to success, but it is not the optimal result. Below 200 there is little chance to success and above it will cost more time and space to calculate the result. However, when setting epoch to 200, the duration is 0.2 second which has been much longer than BFS already.
If a rational exploration function such as Epsilon-Greedy Exploration is used instead of a completely random one, the learning may be more efficient. Epsilon-Greedy Exploration Method will assign a small probability ϵ to explore randomly while giving a higher probability to explore on current policy. In order to force more random exploration at the very beginning and trust current policy more and more along the training process, we also introduced an adaptive decay rate on the probability ϵ of random exploration. If the decay rate is 1, it will act completely randomly. If the decay rate is 0, it will act completely on current policy. To investigate the influence of decay rate as well as learning rate α in Q-Learning, we made use of Control Variate Method and eventually worked out the conclusion shown in Figure9, where vertical axis represents ”Number of Success”. In our experiment, we will run 10000 training epochs for every setting of hyper parameters, so if the number of success is higher, it in fact implies that Q-Learning converges faster

where the results of later epochs will always be success based on current (optimal) policy. As shown in Fig. 8, with the decrease of decay rate, the dependence on random action will decrease slower, so it will do more random actions for exploration which lower the speed of convergence as a result. In addition, with the increase of Alpha from 0.2 to 0.3, its number of success also increases obviously, which indicates the improvement in convergence speed.

Besides, we also implemented a modified version of Q-Update:

$Q(s, a) \leftarrow \alpha R(s, a, s') + \gamma max f(Q(s', a'), N(s', a'))$

compared with the original formula:

$Q(s, a) \leftarrow \alpha R(s, a, s') + \gamma max Q(s', a')$

in order to explore areas whose badness is not (yet) established, eventually stop exploring.

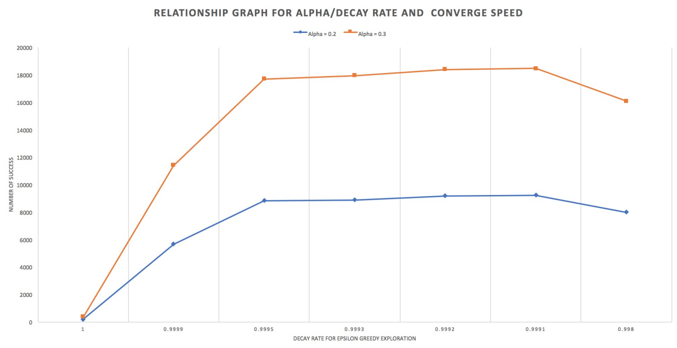


Fig. 9. The graph for different alpha and decay rate

VI. Conclusion

So far we have successfully implemented several search algorithms and one naive Q-learning algorithm to solve the Sokoban problem. Our searching algorithm can solve up to problem 7 in the problem set Microban(see references), which contains 3 boxes and a comprehensive layout. While the Q-learning can only solve problem with one box and simple layout. It seems that for this game, searching is a better approach than learning. But due to the limitation of time and other resources, our solution is not capable of solving this game completely. There are several possible improvement that can be made. For example, the heuristic in our searching only contains Manhattan Distance and Open Goal, both may not produce the optimal solution. Thus these heuristics are not consistent. It is very likely that with the help of a better designed heuristic, the searching algorithm will be able to solve much more complex problems. In addition to searching, the Q-learning for our project is just a very basic approach. We can actually make more improvement by trying some Feature-Based Representations, so that the player may be able to identify some patterns like human player do. Although we can now solve certain problems, more effort need to be made, and we believe that it is feasible for AI to solve the game of Sokoban.

VII. Future Work

Firstly, We successfully used several search algorithms to solve some Sokoban problems which are not too complicated. For some problems which contains more than 4 boxes, for example, the results are not very satisfying. So, in the future, we would try to find more sophisticated heuristic functions to make those problems be solved nicely.

Secondly, Q-learning can only solve naive problems(one box with simple map), as the number of states rises sharply if there are more than one box. We would try other reinforcement learning methods which has lower time complexity.

VIII. Related Work

1) In this paper Using an Algorithm Portfolio to Solve Sokoban distributed by Nils Froleyks(2016)[5], he was also working on the search algorithm of solving sokoban. He analysis the problem and then use graphs to show the process of all the search algorithms. From his result, we have a basic concept about the sokoban solver and get the ideas of designing search algorithms.
2) This is also a paper about different search algorithms designed in sokoban solver and they contract the performance between different search algorithms. This also gives us some ideas about designing the solver and which algorithm may be the best.
3) This link contains the Sokoban problem set called Microban

IX. Appendix

A. *source code*

https://github.com/zzhongab/AI_Sokoban

Here is our final code of the project. Please go into the github and clone the code. Then follow the instruction in Readme to run the project.

B. *javadoc of search classes*

http://hgaoab.student.ust.hk/javadoc3211/docs/package-summary.html

Here is the structure of Search classes and the interface. Enter the website to have a clearer look at the structure of different search algorithms.

C. *javadoc of Q-learning classes*

http://hgaoab.student.ust.hk/javadoc3211/reinforcement/reinforcement_learning/package-summary.html

Here is the structure of the Q-learning classes. Enter the website to see the detail structure of our design of Q-learning.

References

[1] E. F. Moore, “The shortest path through a maze,” in *Proc. Int. Symp. Switching Theory, 1959*, pp. 285–292, 1959.

[2] R. Tarjan, “Depth-first search and linear graph algorithms,” *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.

[3] P. E. Hart, N. J. Nilsson, and B. Raphael, “A formal basis for the heuristic determination of minimum cost paths,” *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[4] C. J. Watkins and P. Dayan, “Q-learning,” *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[5] N. Froleyks and T. Balyo, “Using an algorithm portfolio to solve sokoban,” 2016.