Sokoban Problem Solver

Qiaowei Li

1. Abstract

The game of Sokoban[1] is an intriguing platform for algorithm and AI research. In this project, I am dedicated to solving a basic Sokoban problem with normal Breadth First Search, comparing the result with different searching strategies. By implementing different direction strategies to solve the Sokoban and comparing the number of expanded states respectively, we find there can be as much as 10% difference between two methods despite the map is not large. Since Java has significant advantages over other languages and environments that make it suitable for just about any programming task. The project is finished using java with eclipse IDE. Usually, we only used the Console or command line to output the result. In this project, a GUI[2] is necessary for us to observe the results in a direct way. After searching for different demos online, I decide to use AWT and Swing[3] packages in Java. These two seem to be the most popular tool to build a GUI. And when it comes to a GUI, even a simple program can be difficult.

2. Introduction

Sokoban is a game in which the player tries to push all the stones in a maze onto goal squares. Any stone can be placed on any goal square. The stones are moved by pushing them one square at a time by the player character. The player cannot move through walls or stones and can only push the stones along the four cardinal directions, not diagonally. Also, stones cannot go through walls or each other, and only one stone can be pushed at a time. The objective is to place all stones on the goal squares with a minimum number of pushes. Skilled human players rely mostly on heuristics; they are usually able to quickly discard futile or redundant lines of play, and recognize patterns and goals, drastically cutting down on the amount of search. Some Sokoban puzzles can be solved automatically by using a single-agent search algorithm, such as A*[4], enhanced by several techniques which make use of domain-specific knowledge. The more complex Sokoban levels are, however, out of reach even for the best automated solvers.

3. Related work

During the preparation time, I found some interesting project demo and related thesis in this field. One of the thesis, written by Helsingin Yliopisto, presents an overview of the techniques that have been applied to the domain of Sokoban. This master thesis has a very detailed introduction about this topic and its solutions theoretically. Especially it has a very specific comparison between BFS and DFS in both time complexity and space complexity. From the conclusion of this article, Breadth-First Search algorithm will solve puzzles faster and with less memory than a similarly naive Depth-First Search. That is why I choose BFS as our searching

algorithm.

4. Approach

We use two similar BFS algorithms to test for how many states are expanded and how many moves and pushes are taken. The first one is a very normal BFS algorithm, it begins at the player position and start to expand at up, right, down, left four directions. If any of the four directions are eligible, they will be put in a LinkedList. This LinkedList works like a queue. If all the nodes in one level are all expanded and still no solution, the oldest node will be popped out and expanded. Thus, the number of the expanded nodes is the number of nodes popped out from queue. When expanding the nodes, we choose to expand all nodes in the same level and then move on to the next level. After each push trial, we need to test whether the new node is the goal or not.

Then the question is which way among four directions is the best one. We use the direction strategies “Up, Right, Down, Left”, “Right, Down, Left, Up”, “Down, Left, Up, Right”, “Left, Up, Right, Down” as sequence. We do not mean that there are only four direction strategies. In fact, I just want to find out if the difference in the direction may cause obvious difference even in a Sokoban of small size 8*7.

5. Implementation

Beside the algorithms themselves, the performance of algorithms depends heavily on the detail of the implementation. The implementation used for all four experiments in this project is relatively unoptimized with focus on rapid prototyping and development. However, the program is fast and robust enough to provide reliable results for comparing different direction strategies.

5.1 Simple Deadlock Detection

When executing a solver program, the worst thing is stuck in a loop or deadlock. We use the LinkedList to check if it is empty every time we want to get the states expanded. If the queue is empty before we can find a solution, this means there is no solution in this particular searching algorithm.

5.2 Make a GUI Result

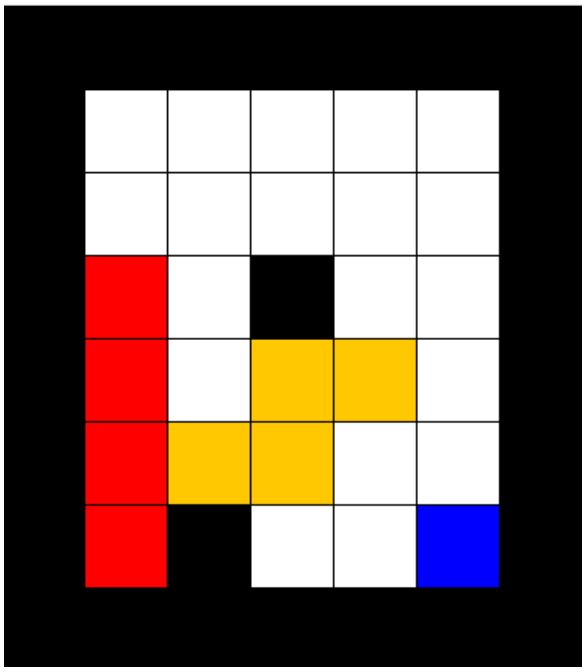
Build a GUI with java swing and awt packages. The problem is how to convert the String or Array into a visible window and make every move dynamically on that window. In java, a JFrame is a top-level window with a title and a border in Swing package. Then fill the frame with graph, grid and colors which is provided by awt package. Find different colors to fill the grids in order to represent different roles in Sokoban. We use red to represent goal spots, blue for agent, orange for boxes. Then record every moves and pushes made in solving the problem, so we can show the move one by one. Using event-based updates, i.e. updating the ball position and repainting the entire grid every time a user presses the left or right key. The reason why we need to record all the steps instead showing every state is there are too many states. And state cannot give you a consistent graph, because every time a state

popped out, the user will be confused by the sudden inconsistent change. So the best choice it find the path first and figure out how to display it.

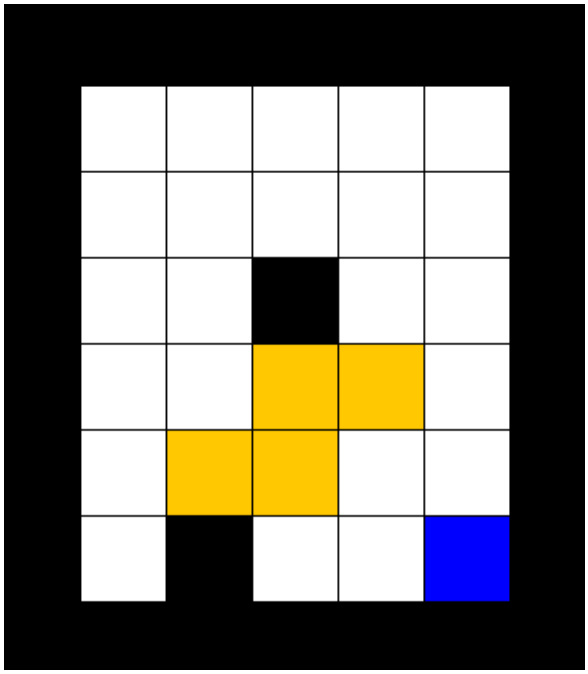
5.3 BFS Implementation

The implementations for the Breadth-First Search is written in an obvious way, translating quite directly from the pseudocode descriptions of the algorithms. The BFS implementation uses a LinkedList for the search frontier and a hash set for the explored set. Also there is no duplicate states allowed in this program. States are considered to be already tried if the player accessible area is identical, disregarding the actual exact player position.

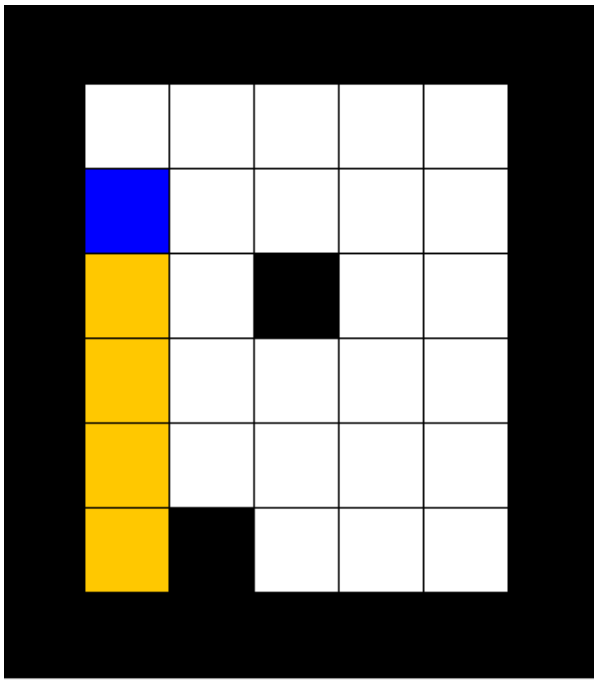
Results



This is the original state of the Sokoban. Red is for goals, blue is for agent, orange is for boxes. The moves and pushes will show as follows:



This is the original state without goals. User press the left and right key to see the previous and next step. After press right key for 34 times, we have the ultimate result.



If check with the first graph, we can see all the boxes are fit in the right positions.

Conclusion

Directions	Num Of States	Path
U R D L	168710	ulULLulDDurrrddlULrruLLrrUruLLLulD
R D L U	164945	luULLulDDurrrddlULrruLLrrUruLLLulD
D L U R	152812	luULLulDDurrrddlULrruLLrrUruLLLulD
L U R D	150812	luULLulDDurrrddlULrruLLrrUruLLLulD

From this table, the cost of path is nearly the same, but the number of expanded states are not identical. The fourth one is 10% less than the first one. So my conclusion is direction strategy is important in this project. But from this limited graph we cannot find a common strategy. Because some of them may be suitable for other graphs. That is why we need Heuristics function which can help adjust the direction any time. The program will not be limited to these four directions. So next step we can add some Heuristics functions to compare the results with these four.