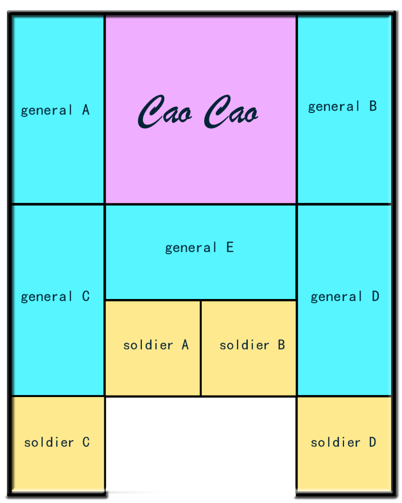
Huarong Dao Puzzle Solution(Search)

Wen Xiao(g4xiaowe), Zhi Li(g3lizhi), Zhen Ren(c3renzhe)

December 6, 2015

1 Introduction

Klotski (Huarong Dao) is a Chinese traditional sliding block puzzle game. It aims to move the largest block out of the puzzle, regardless the other smaller blocks. The game was inspired from a Chinese historical story. A Chinese famous warlord, Cao Cao was surrounded by four enemy generals and soldiers, and he had to think of a way to escape. The game is consisted by one largest block, which represents Cao Cao, and 4 smaller rectangle blocks (represents enemy generals) and 4 smallest blocks (represents soldiers). The number and the size of each blocks are fixed; however, the position can be various. In addition, there are also two empty positions with the smallest block size so that all blocks are able to move until the largest block get out of the puzzle by passing through the hole at the bottom. The detail visualization of Klotski is the following.



The picture above shows an example position of each block. Our program takes the above state as initial state, and expand to multiple states by moving only one block each step.

The variability of the position of each block results different versions of this game. It is apparent that each version has its own unique way to solve it and has its own optimal solution. The picture above describes one of the most famous versions of this game and we will take this version as an example to demonstrate our solution towards this problem. (we just take the above version as an example, but our solution is designed to work on all versions of this game).

Because of the similarity between this game and the "8-Puzzle" game introduced in lecture, by the suggestion from Professor Fahiem Bacchus, we decided to choose **Breath First Search** to solve this problem. We took the above implementation of each blocks as initial state and generate Breath First Search to expand into multiple states by moving one block one position each time; then the shortest path to goal state should be the optimal solution of the game. The detail implementation of each functions and class (including state space, hash code, etc...) will be illustrated in Section.2, Build Up the Model in this report.

In order to achieve our design in this project, the first problem came up to us is how do we choose to represent the Klotski game in Python. It is no doubt that BFS will generate uncountable number of nodes and each node is a Klotski board with different position of blocks. Besides worrying about the size of each node, thousands of nodes has already become a huge data storage problem and it will undoubtedly affect the run time of program. As the result, the design of our implementation of Klotski game board should be minimize but also include essential data to ensure the the best running effort. The graph below shows our **first** attempt to implement the game board but we changed soon because its size and instability (all data in list, which is too easy to be changed). Finally, we choose list of tuples to represent the Klotski board in python program and use a function to print the list to a table version so people can easily determine which block moves in each step. The detail of the date structure we use and the implementation of print function will be demonstrated in the following sections.

Our first version of Klotski game board:

```
game_board =
[[B, A, A, C]
 [B, A, A, C]
 [D, E, E, D]
 [D, F, G, D]
 [H, '','',I]]
```

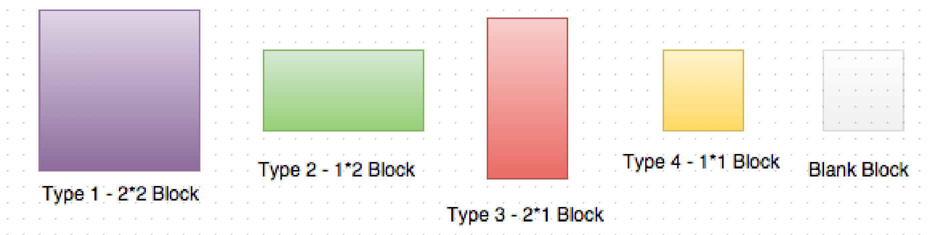
2 Build up the Model

2.1 Idea of the Model

As introduced above, the Huarongdao puzzle game is made up of a size 5*4 board, which consists 4 different types of blocks - one of size 2*2 block (representing Cao Cao), 4 of size 2*1 rectangle blocks (representing general A, B, C and D respectively), 1 of size 1*2 rectangle blocks (representing general E) and 4 of size 1*1 smallest blocks (representing soldier A, B, C and D respectively). And also contains 2 of size 1*1 empty positions.

The idea of our Huarongdao model is that we consider the 4 different types of blocks, together with 2 empty positions, as 5 types of the blocks. And the Huarongdao game board is made up by all those blocks and has coordinates. The board itself is a state space. The model has the function that will produce all possible new boards by moving one of the block. And it also has the function to make the board visualized to users.

More detailed, the 5 different types of blocks are shown below. Each block has three attributes, the vertical coordinate of its upper left vertex, the horizontal coordinate of its upper left vertex and its name. The information are stored in a tuple, for instance, (2, 1,"A") represents a block named "A", whose upper left vertex is at the coordinate (2, 1) on the board.



The board of Huarongdao in our model is a size 5*4 rectangle with the origin coordinate on the upper left corner and covered by all of those blocks.

(add board image)

We used a list of ordered sublists to represent such board. And each sublist contains the tuple of block(s) which have the same type, and the elements of those sublists are sorted in a way that first sorting based on their horizontal coordinate values and then the vertical values. The helper function **sort()** are built for simplicity. The reason we sort those blocks is that we consider two boards with the different positions of the blocks with the same size are same. Therefore, such two boards will have the same hashable states, which is useful for path/cycle checking. The following is an example of how we use list of sublists to represent the sample board from above.

```
[[(0,1,"A")],
 [(0,0,"E"),(0,3,"F"),(2,0,"C"),(2,3,"D")],
 [(2,1,"B")],
 [(3,1,"G"),(3,2,"H"),(4,0,"I"),(4,3,"J")],
 [(4,1,"X"),(4,2,"X")]]
```

The first element of the list, [(0, 1, “A”)], is always contains the largest block - the block we want to move out. The second element of the list contains all the blocks with size 2*1 - type 3 in the above image, which is [(0,0,”E”),(0,3,”F”),(2,0,”C”),(2,3,”D”)] in this example. The third element of the list, [(2,1,”B”)], is the block with size 1*2 - type 2 in the above image. In this case, [(3,1,”G”),(3,2,”H”),(4,0,”I”),(4,3,”J”)] is the fourth element of the list contains all the blocks with size 1*1 - type 4 in the above image. The last element of the list contains two blank blocks - type 5, say,[(4,1,”X”),(4,2,”X”)].

2.2 Implementation of the Model

The file HuaRongDao.py is class that builds the Huarongdao model described above and provides some methods to solve Huarong Dao puzzle.

1. The class initializer method **__init__**: This method initializes a Huarongdao object an initial board, each block on the board is assigned by an initial position and a name, which is a list of sublists of the blocks.

2. the **successors** method: This method takes a state("self") as an input, and return a list of all its successors. Note that each Huarongdao object represents a state in the Huarondao state space and its successors are Huarongdao objects of which are reachable from current board("self") by a single movement of any block. Also, every Huarongdao object contains the name of action used to obtain it (a string indicates which direction the block is moved), the reference to its parent, the cost of getting to this state (zero in this case), the Huarongdao specific data structure (the board and the blocks information) and the blocks with the same type are sorted by calling **sort()**.

The logic of the successors() method is to walk through every block on the board, check whether or not this block is eligible to move in each of the four directions (up, down, left and right). If the block is eligible to move, then create a new Huarongdao object which is obtained from a single movement of that block and store the new object in a list, we called it State in our project. Note that one particular block may have multiple ways to move, in such case, it will produce multiple successors from current state by moving the same block in the different directions. Then repeating this procedure until checked the eligibility of all blocks and return successors which is the list State. Note that we do not need to check the blank blocks. Moreover, to consider a block is eligible to move or not we check is there enough blank space that allows the block to move one unit distance in that direction. In our model, firstly, we store the blocks of the current board in the lists such that each list contains the same type of blocks. Secondly, we looped through every element of different type. For each element, we checked whether or not this block is able to move in all four directions. For instance, The Type 1 block are able to make a single movement if and only if the two blank blocks are both next to one side of the Type 1 block. The possible movements in all four directions of Type 1 block are similar. The following is the code checking whether the Type 1 block are able to move downwards, which checks if there is a blank block lies in two unit distances below the upper left position of Type 1 block and there is another blank block lies next to the previous one on the right side.

```
if (((size_4[0][0]+2, size_4[0][1], "X") in blank) and
    ((size_4[0][0] + 2, size_4[0][1] + 1, "X") in blank)):
```

For the Type 3 blocks, there are two kinds of scenarios of a single movement: one blank block lies right below or above the Type 3 block or two blank blocks both lies next to the right or left side of the Type 3 block. The following code are two examples of checking these two scenarios. The first one checked the availability of type 3 moving one unit distance downwards by checking if the coordinate of any blank block is at right below this block. And the second one checked if there are two blank blocks lies next to the right side of the Type3 block in order to make a rightward movement.

```
if (((size_2[i][0]+2, size_2[i][1], "X") in blank)):


if (((size_2[i][0], size_2[i][1]+1, "X") in blank)
    and ((size_2[i][0] + 1, size_2[i][1] + 1, "X") in blank)):
```

For the other two types of blocks, Type 2 and Type 4, the Type 2 blocks have the similar two scenarios as the Type 3 blocks, we discussed above, but are in the different directions. And the Type 4 blocks are able to move one unit distance in any direction if there is a blank block lies next to it.

3. The **hashable_state** method: This method returns the current board (state) using a tuple of sub-tuples, which can uniquely represent the current board(state). Since python dictionary is a good way to implement path/cycle checking in our model, which is critical of saving time and space, we are using a tuple of tuples as indexes of the dictionary. The way to implement it is that basically casting the coordinate_list(the board infomation list), which was a list of sublists, to a tuple of subtuples and remove the name of each block. Remember that the first subtuple contains the Type 1 block, the second subtuple contains the Typle 3 blocks, Typle 2 block is in the third sublist, Typle 4 blocks are in the fourth sublist and two blank blocks are in the last sublist. The following shows an example of the hashable state of the board showed above.

```
(((0,1)),
((0,0),(0,3),(2,0),(2,3)),
((2,1)),
((3,1),(3,2),(4,0),(4,3)),
((4,1),(4,2)))
```

4. The **print_state** method: This method print out the current state in the user-friend way. We implement it by creating an empty 5:4 board which has 5*4=20 empty positions. We use a list of 20 sublists representing this empty board, and each sublist representing a position. The sublist with index zero is linked to the first position from left side in first row, and sublist with index one is the second postion in the first row and so on. Each sublist has 5 attributes, first four are booleans which shows whether this positon has border on the top side, right side, bottom side and left side respectively, the fifth attribute is the name of the relating block. Note some of the blocks may cover multiple positions. While we walk through every block on the current board, we check which posotion(s) this block cover and store those infomation into the corresponding sublist(s). The following shows a sample result of this method.

```
|----|---------|----|
|    |         |    |
|  B |    A    |  C |
|    |         |    |
|----|---------|----|
|    |  F      |    |
|  D |----|----|  E |
|    |  G |  H |    |
|----|----|----|----|
|  I |  X |  X |  J |
|----|         |----|
```

5. The **get_goal** method simply return the coordinate of the largest block on the current board.

3 Algorithm

3.1 Select algorithm: Breath First Search

Up to now, we have learned several searching algorithms, and I will analyze if these algorithm can be used in this problem.

1.**Uniform-Cost Search**

It will expand the node with the least cost in the open list. Since the cost of every move equals one in our model, it is the same as BFS.

2.**Depth First Search**

It will place the new paths at the front of the open list, which means we will go through one path until no way to go. Then we will change to another path. It can work well in this problem, that is to say, we can always find a solution to the puzzle if exists. And as we already knows that the worst running time of this algorithm is $O(b^m)$. Which seems better than BFS. But its solution is always the 'first' one approached, not the 'shortest'. We also implement this algorithm so that we can compare the results.

3.**Iterative Deepening Search**

It just like the combination of BFS and DFS. Start at L=0, We iteratively increase the depth limit, and then perform a DLS for each depth limit. It can always find the optimal solution, and the worst running time of it is less than BFS if BFS check the status of the node when it is expanded. But we can make the BFS better, and we can include it in the latter section.

4.**A* Search**

The idea is to build a heuristic function which guesses the cost to get to the goal from node n, and calculate the cost from start to n, then get an evaluation function by adding them up. Every time we need to expand the open list, we just expand the node with the lowest evaluation function. It can work very well if we have a perfect evaluation function. But the problem here is that we can not get a good evaluation function from the puzzle. We do not know the distance between 'solution' and the current state.

5.**Backtracking Search(CSP)**

It provides a good way to solve the problems with constraints(i.e the sudoku problem we solved in the assignment2). But there is no constraints in this problem, the blocks can be in any place in the board as long as there is space. So we can not use this model.

6.**Breadth First Search**

It place the new paths that extend the current path at the end of the open list. By which we means just search the solution step by step, after reached every possible state with the same number of step, we move on to the next step. Thus it can always find the 'shortest' solution, which is exactly we want. As for the running time, which seems worth than IDS, but we can make it better by **checking it when its parent is expanded**.

3.2 Path/Cycle Checking

Since we can reach the same state by the different steps, path checking and cycle checking are very important to save time and save space.
Here in this problem, we simply used a dictionary to store all states. Each time when we expand a state, we just add all its successors into the dictionary, and if it is already in the dictionary, we just ignore it.
And as we mentioned in the previous part, two states with the same position of the blocks with the same size is considered to be the same.
For example, the following two states are considered to be the same .

```
|----|---------|----|      |----|---------|----|
|    |         |    |      |    |         |    |
| B  |    A    | C  |      | D  |    A    | C  |
|    |         |    |      |    |         |    |
|----|---------|----|      |----|---------|----|
|    | F       |    |      |    | F       |    |
| D  |----|----| E  |      | B  |----|----| E  |
|    | G  | H  |    |      |    | G  | H  |    |
|----|----|----|----|      |----|----|----|----|
| I  | X  | X  | J  |      | I  | X  | X  | J  |
|----|         |----|      |----|         |----|
```

(The positions of 'B' and 'D' are exchanged.)

3.3 Implement of BFS

We have following variables defined in the algorithm:
1. searched: a dictionary to do the path checking and cycle checking.
2. bfs: the open list to contain the states we need to check.
3. current_step: record the current step, since we are doing BFS, we search the solution step by step, which means we need to check all states with step i before checking the states with step i+1. Once we finish all the states with the same step, we will move on, so just add this attribute by 1 to inform which step are we in.
4. start: the start time of this algorithm, it is used to calculate the whole running time of searching using this algorithm.

Then I will introduce how this algorithm works.
Start from the initial state, we just loop in the bfs. Every time we reach a state in bfs, we will first check if its step is the same as the current_step, if

not, we will add the current_step by 1 to indicate we reached the next level in the searching tree. And then we will get all the successors of the current state, first check if there is one state satisfied our goal function, if so, just print out the final step number and the whole process. Then we will let the step of all the successors equal to one plus the current state to indicate it is on the next level.Next we will append them to the bfs and add them to the searched dictionary. If after we search all the bfs and still can't find a solution, just print 'no solution'.
One particular thing to notice is that we **check the satisfaction of the state when its parent is reached instead of it is reached**, which saves lots of time and space and dominate the IDS algorithm, that is also a reason why we choose bfs.

3.4 BFS and DFS

We also implement the DFS algorithm in this problem, here I am going to compare these two algorithm.

Algorithm Name	BFS	DFS
Runtime	2.6	1.2
Step	116	3185
Number of States	24029	11948

We can see that BFS can find the optimal solution – the solution with the shortest step from the initial state. But the running time of DFS is better than BFS, and the states it reached is much less than BFS.
This is very interesting because as we learned in the class, the worst running time of BFS and DFS are $O(b^{d+1})$ and $O(b^m)$ where m is the longest path in the searching tree. And in this puzzle game, m should be much larger than the best solution(116 steps), so BFS is supposed to be much faster than DFS. In my opinion, we get this result because the solution in our search tree is not unique. There are many different solutions with different steps; once we go along a path, it is not hard to reach one solution if we don't repeat doing the same thing(we did the path/cycle checking to avoid repeating). So DFS here is faster but unfortunately it can not find the optimal solution, so we can not use it.

4 Results and Evaluation

4.1 Results

In accordance with the algorithm analysis and comparison above, we implement Breath First Search in our program to ensure getting an optimal solution. The program takes 2.6s and 24029 states to find the optimal solution of our example version of Klotski Game in this report. The following two pictures shows our print version of initial and final state in our program.

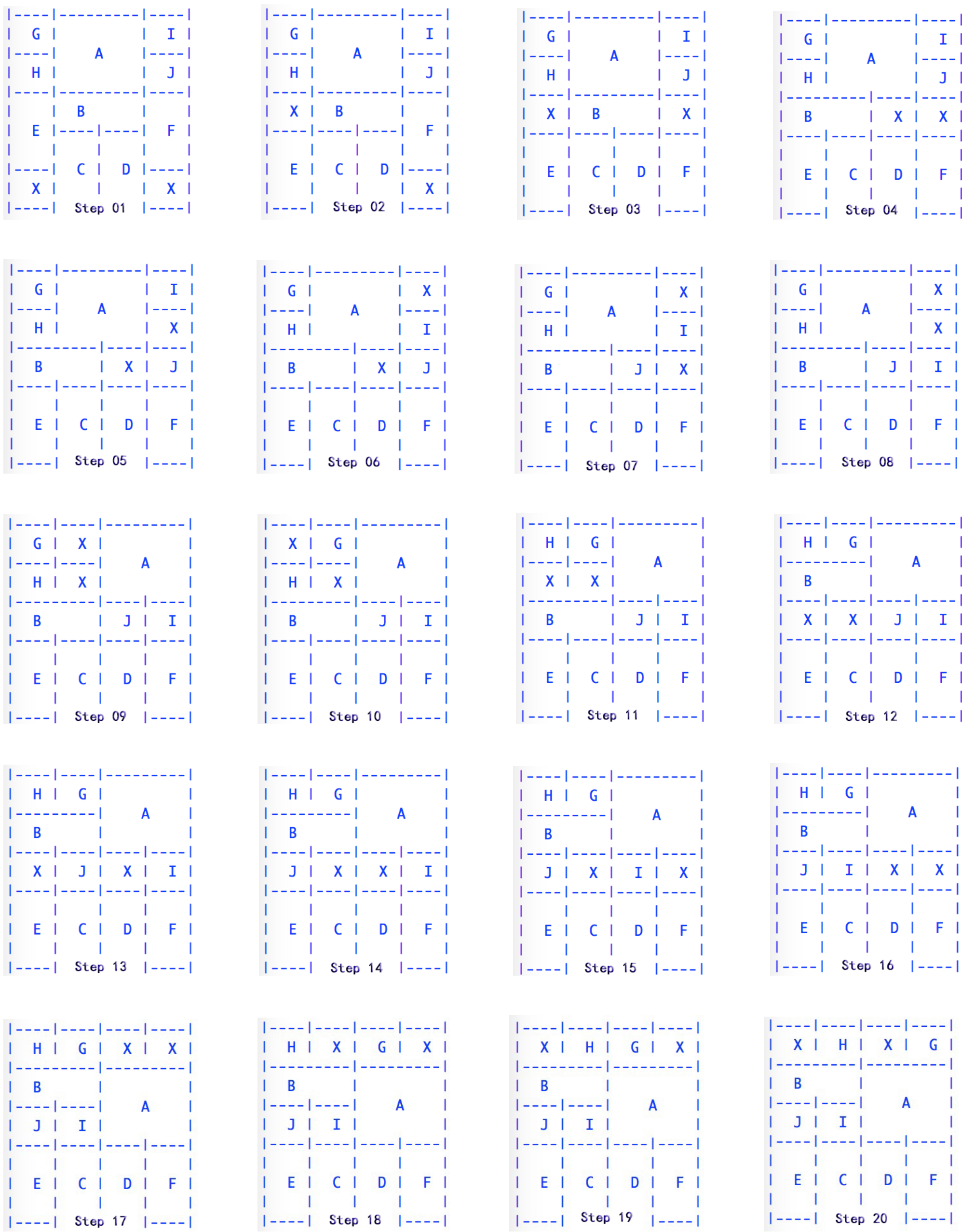
initial state:

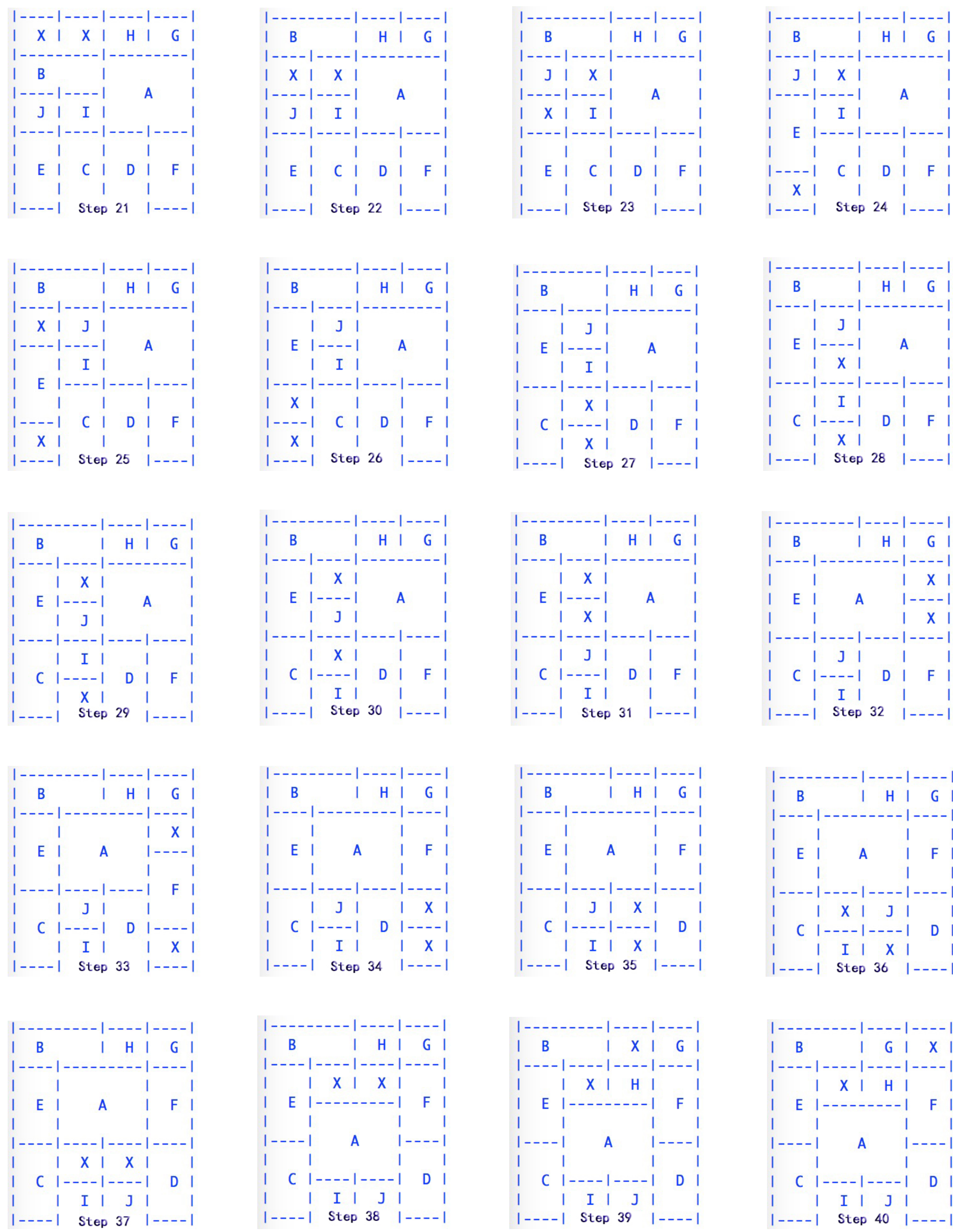
```
|----|---------|----|
|    |         |    |
|  E |    A    |  F |
|    |         |    |
|----|---------|----|
|    |  B      |    |
|  C |----|----|  D |
|    |  G |  H |    |
|----|----|----|----|
|  I |  X |  X |  J |
|----|         |----|
```

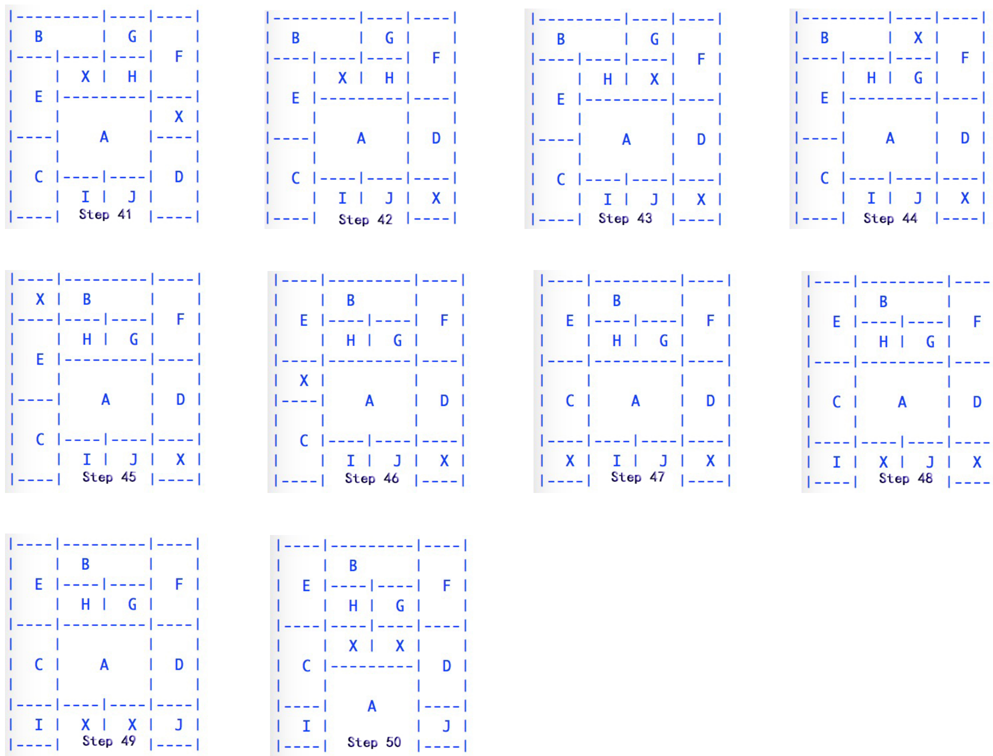
final state:

```
|----|----|----|----|
|    |    |    |    |
|  D |  C |  F |  E |
|    |    |    |    |
|----|----|---------|
|  J |  G |  B      |
|----|---------|----|
|  X |         |  I |
|----|    A    |----|
|  X |         |  H |
|----|         |----|
--------------------
total steps: 116
```

In order to clearly present the detail solving steps of our program, instead of using the above initial state, we will take an easier Klotskil Game board as our initial state so that each step (each move of block) can be clearly demonstrated.


Step 01
Step 02
Step 03
Step 04
Step 05
Step 06
Step 07
Step 08
Step 09
Step 10
Step 11
Step 12
Step 13
Step 14
Step 15
Step 16
Step 17
Step 18
Step 19
Step 20


Step 21
Step 22
Step 23
Step 24
Step 25
Step 26
Step 27
Step 28
Step 29
Step 30
Step 31
Step 32
Step 33
Step 34
Step 35
Step 36
Step 37
Step 38
Step 39
Step 40



4.2 Evaluation and Improvement

Compared with other results on the internet, our algorithm can always find the optimal solution, but the speed is a little slower than them. And since we use a dictionary to store all the visited states, we waste much more spaces. As for the steps, it will be better if we can think one more step when finding the successor, which means when a block can move two steps, we can just add the state after two moves in the successor list of the original one.