TungHo Lin

Txl429

EECS391

PA1 Write Up

1. **Code Design**

The whole program is organized into two parts. An EightPuzzle class and an EightPuzzleSolver class.   
  
**Class EightPuzzle:**  
This class represents a puzzle. There are two constructors for this class: one will take a String representation of the board and build an instance; the other will take a 2D array as the board and build it.   
  
Fields include:  
board: a 2D 3x3 String array that represents the board  
f\_value: represents the function value of this puzzle, which is g + h  
g\_value: represents the depth of this puzzle

h\_value: represents the heuristic value of this puzzle

parent: the parent EightPuzzle of this puzzle  
previousMove: the move in direction that results in this puzzle

Methods include:  
*toStringRep*:   
-converting this puzzle’s board representation to a String representation.  
*convertBoardToStringRep*:  
-helper function for toStringRep  
-take a 2D array as a board and convert it to a String representation  
*isValidStringCombination*:  
-takes a String combination and checks if it has the correct String regex and only contains one blank tile  
*isValidBoardCombination*:

-takes a 2D array as a board and utilizes isValidStringCombination to check if it is a valid board representation  
*containsOneBlank*:

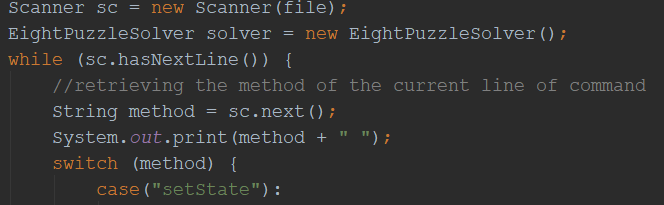
-helper method for isValidStringCombination  
-check if a String combination has only one blank tile  
*equals*:

-compare between two EightPuzzle using their 2D array board

**Class EightPuzzleSolver:**  
This class contains all the methods in the assignment and other helper methods that helps solving the puzzle.  
  
Field include:  
currentPuzzle: stores the current puzzle state  
nodeLimit: stores the limit for beam search

maxNodes: stores the maximum number of nodes that can be considered/created

nodeCtr: stores the current number of nodes generated  
goalState: the goal puzzle state, which is “b12 345 678”.  
  
Methods include:  
*main*:  
-takes a file path to the test text file  
-Scanner will scan the command one line at a time  
-the method name of each line will be extracted and evaluated for each case in the switch statement

  
-Before executing, presence of parameters will be checked, Exception will be thrown if there are no parameters for a method that requires parameters.  
-each case will execute the specified methods with the parameters  
-the “solve” case will check if it is A-star if beam and parse its corresponding parameter(heuristic for A-star and k-limit for beam).  
*setState*:  
-Set the current puzzle state to be the input String combination  
-creates a new EightPuzzle instance and assign it to be the currentPuzzle field

*randomizeState*:

-make n random moves from the goal state  
-create a new EightPuzzle with the goalstate combination and assign it to be the current state  
-determine the possible moves after each “move” using a helper method possibleMoves  
-using a random int every time to choose a random possible direction

*possibleMoves*:  
-return a list of all possible direction of movement using the current puzzle state  
-find the blank tile by parsing through the 2D array

-using the i and j coordinates of the blank tile and determine if it can move towards any of the four directions (e.g. if i >= 1 then it must be on the second/third row; and therefore can be moved upwards.)  
*printState*:

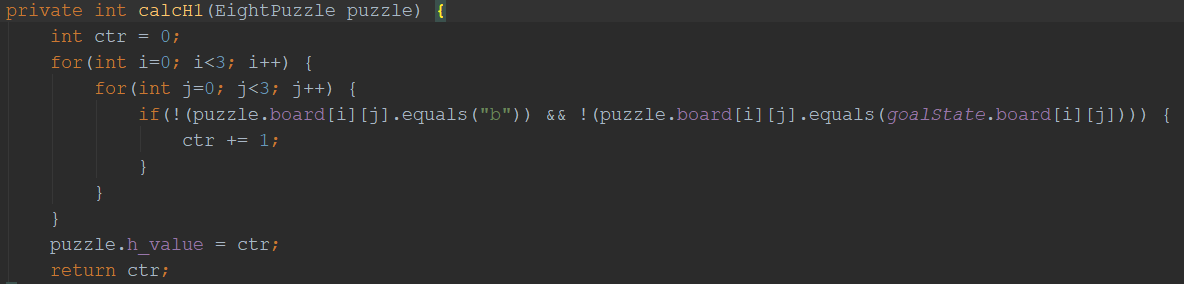
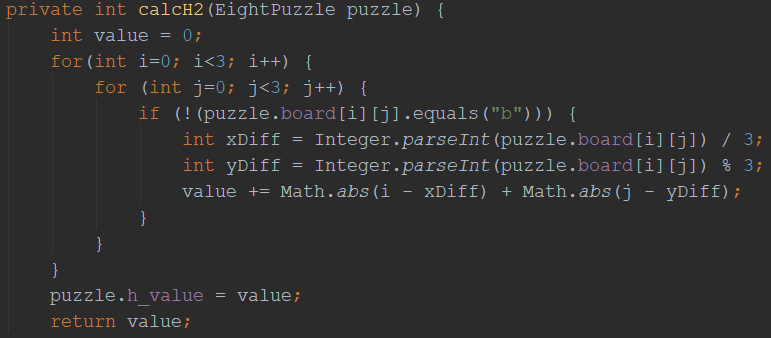
-print out the board of the current puzzle state

-parse through the 2D array, replace the “b” tile with an empty space and print out the whole row  
*move*:

-move the blank tile in the current puzzle state towards a specified direction  
-find the blank tile by parsing through the 2D array

-swap and move the blank tile according to the given direction

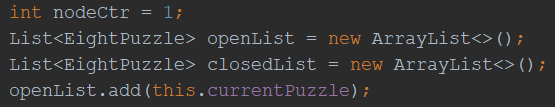
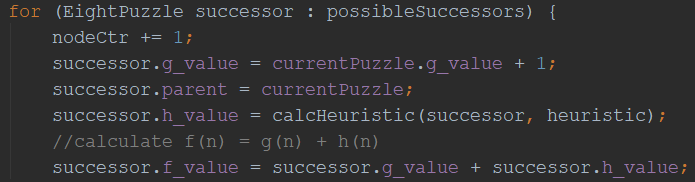
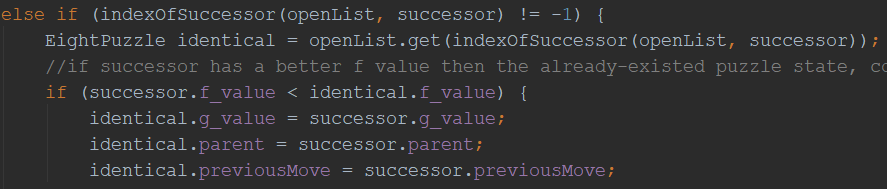
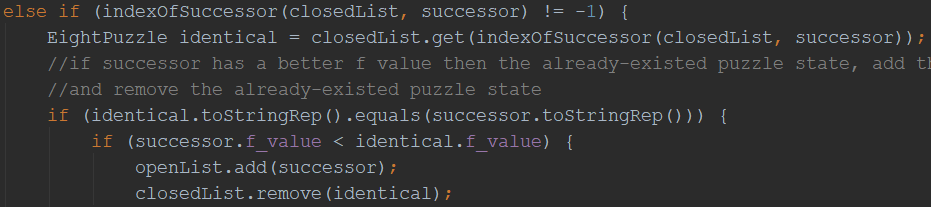
*printSolution*:  
-back-track the current puzzle state, which should be the goal state when the puzzle is solved, to retrieve all the previousMoves.  
-for each parent of the current puzzle state, retrieve its previousMove and assign current node to be its parent

*indexOfSuccessor*:  
-finding a given EightPuzzle in a given list of EightPuzzle and return its index  
-return -1 if not found  
*calcHeuristic*:  
-return the heuristic values of the given EightPuzzle  
-depends on the given heuristic (h1 or h2)  
  
*calcH1*:  
-return the heuristic of calculating the number of misplaced tiles  
-parse through the 2D array and if a tile does not match the corresponding tile on the goalState board then increment the counter  
  
*calcH2*:

-return the value of Manhattan heuristic  
-get the x coordinate difference by getting the integer value of the current tile’s coordinate and divide it by 3  
-get the y coordinate difference by getting the integer value of the current tile’s coordinate and mod 3  
-sum up the absolute values of the x difference and y difference

*maxNodes*:  
-assign the field maxNodes to the given one  
*isComplete*:  
-checks if the current puzzle state equals to the goal state  
*generatePossibleSuccessors*  
-generate a list of possible successors according to the possible moves  
*generateMovedPuzzle*  
-helper method of generatePossibleSuccessorsdsadsadddawdsa  
-create a new instance of a successor

**Class EightPuzzleTester:**  
-a class to run multiple tests on each search methods  
*testSearch*:  
-takes a search method and will test a randomized puzzle from (n=10 to n=150) fir every 10 moves.

**2) Code Correctness**  
*solve\_Astar*:  
-solve the puzzle using h1 or h2  
Pseudocode with screenshots:  
set the node\_counter to 1  
create an openList that stores unexplored nodes  
create an closedList that stores explored nodes  
add the current puzzle into the openList  
while(openList is not empty) {  
 if (node\_counter > maxNodes)  
 stop search  
 if (goal state is reached)  
 stop search and print solution  
 generate successors of the current puzzle  
 for each successor {  
 increment node\_counter  
 assign successor’s g\_value to be current puzzle’s g\_value + 1  
 assign its parent to be the current puzzle  
 assign its h\_value to be the result of calling calcHeuristic(h1/h2)  
 assign its f\_value to be g\_value + h\_value  
 if (the successor is not in the openList or closedList) {  
 Add it to the openList  
 }   
 Else if (the successor is in the openList and has lower f\_value than the already-existing state) {  
 assign the successor’s g\_value, parent and previousMove to the already-existing state  
 }  
 Else if (the successor is in the closedList and has lower f\_value than the already-existing state) {  
 add it to the openList and remove the already-existing state from the closedList  
 }  
 add the current puzzle state to closedList  
 sort the openList using their f\_values  
 }end for each loop  
}

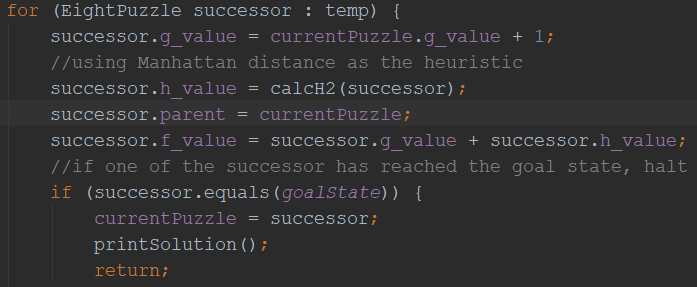
*solve\_beam*:

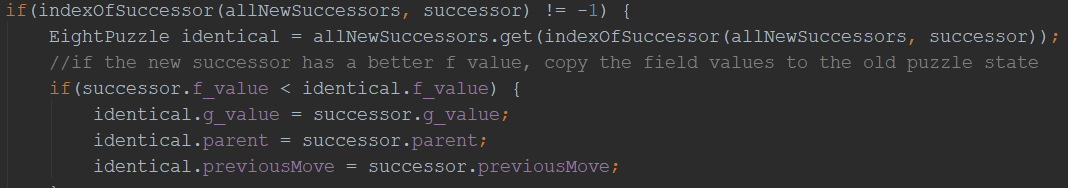
-solving the puzzle by local beam search limited by k number of nodes  
-PsuedoCode:  
if (the puzzle is complete)  
 terminate search  
node\_counter <- 1  
create an openList

while (node\_counter < an upper limit of nodes) {

Create a list for new successors

while (openList is not empty) {

remove the first puzzle state from openList  
  
 for each successor {  
 increment node\_counter  
 assign successor’s g\_value to be current puzzle’s g\_value + 1  
 assign its parent to be the current puzzle  
 assign its h\_value to be the result of calling calcHeuristic(h1/h2)  
 assign its f\_value to be g\_value + h\_value  
 if (this successor equals to the goal state) {  
 terminate search and print solution  
 }

  
 if (this successor already exists and has lower f\_value than the already-existing one) {

assign its g\_value, parent and previousMove to the already-existing one  
 }  
 else {

add it to the new successor list

increment counter

}

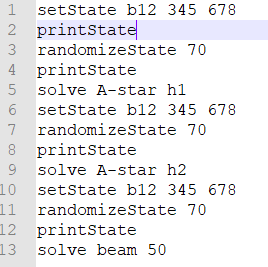
} end while loop  
 sort the new successor list  
 trim the new successor list to k nodes  
 openList <- new successor list with length k  
}end while loop

Basically A\* search starts with an open list that stores all the unexplored nodes and a closed list that stores all the explored nodes. We start off with the starting node in the open list, generate all the successors of it by trying to move the blank tile to all the possible locations; and for each successor, create a relationship between the current puzzle and its successor and calculate its heuristic values. Also for each successor, we check if it existed before in the open list or close list, if yes, then we either change the previous values in the closed list or move it from the closed list to the open list. If it has not existed before, we add it to the open list. After iterating through all the successors, we sort the list using their f values and run the loop again for the newly created open list.

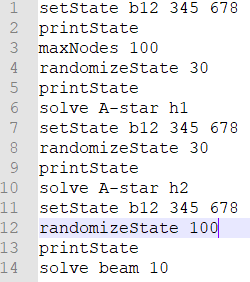
For the beam search, we start off with an open list only and a node counter. We add the current puzzle into the list and start a while loop. While the node counter is less than maxNodes, we create a list(L1) for all the new successors; and while the open list is not empty, we remove the first one, generate all the successors, create the relationship and assign the values to them, just like in A\*-search. After that, we check if each successor is the goal state and terminate the search if it is. If not, we check if it is in the open list and change the previous values if it existed before. We then add this successor to the L1 list and increment the node counter. The main difference is that we sort the L1 list using their f values and then trim it down to k nodes (we will call the k-length list L2). We then replace the current open list with L2, the current best-k-nodes. The loop continues with L2 being the open list.

The similarity between them is basically having a node counter that limits the search to a certain number of nodes. The only time the code fails is when it reaches the maxNodes, at which an error message will print. I have the default nodes at 99999 so that I can test the time length for a higher randomized starting state.

Another thing that the program depends on is the number of randomized moves. I tried to use 50 to 100 moves for randomizing the puzzle and the program takes very very long to run or usually hit the maximum nodes before solving it. I usually use 20-25 moves to test my programs and they should output a solution within 2 seconds. Anything above that is going to cause slowness.

If you run the **test1**file, which is:  
  
It should output the same solution for all 3 search, a 14-step solution. You should see the following output:  
Solved!

Number of Steps: 14

[down, right, up, left, left, up, right, down, down, left, up, right, up, left]  
  
If you run the test2 file, which is:  


It has a small maxNodes and a small k for local beam search, therefore all 3 searches will not return a solution, the output should be:  
The system has exceeded the maximum number of nodes to be considered!  
The system has exceeded the maximum number of nodes to be considered!  
Solution not found!

**3) Experiments**  
  
Table 1 shows the **average solution length** generated of tests run on h1, h2 and beam using randomize moves from increments of 10 from n=10 to 150, maxNodes = 99999, nodeLimit = 1 million

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| No. of randomize moves | A\* search with h1 | A\* search with h2 | Beam search with k=5 | Beam search with k=30 | Beam search with k=100 |
| 10 | 8 | 8 | 8 | 8 | 8 |
| 20 | 8 | 8 | 8 | 8 | 8 |
| 30 | 14 | 14 | No Sln | 14 | 14 |
| 40 | 16 | 16 | No Sln | 16 | 16 |
| 50 | 14 | 14 | No Sln | 14 | 14 |
| 60 | 20 | 20 | No Sln | 20 | 20 |
| 70 | 24 | 24 | No Sln | No Sln | 26 |
| 80 | No Sln | 26 | No Sln | No Sln | No Sln |
| 90 | 24 | 24 | No Sln | No Sln | 36 |
| 100 | 22 | 22 | No Sln | No Sln | 22 |
| 110 | 22 | 22 | No Sln | No Sln | 22 |
| 120 | 20 | 20 | No Sln | 20 | 20 |
| 130 | 24 | 24 | No Sln | No Sln | 24 |
| 140 | 22 | 22 | No Sln | No Sln | 30 |
| 150 | 24 | 24 | No Sln | 26 | 24 |

Table 2 shows the **average nodes generated** generated of tests run on h1, h2 and beam using randomize moves from increments of 10 from n=10 to 150, maxNodes = 99999, nodeLimit = 1 million

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| No. of randomize moves | A\* search with h1 | A\* search with h2 | Beam search with k=5 | Beam search with k=30 | Beam search with k=100 |
| 10 | 44 | 30 | 59 | 177 | 222 |
| 20 | 44 | 30 | 59 | 177 | 222 |
| 30 | 733 | 198 | >nodeLimit | 513 | 1343 |
| 40 | 1434 | 252 | >nodeLimit | 594 | 1606 |
| 50 | 598 | 172 | >nodeLimit | 498 | 1251 |
| 60 | 8539 | 887 | >nodeLimit | 836 | 2385 |
| 70 | 48013 | 4098 | >nodeLimit | >nodeLimit | 2566 |
| 80 | \* | 10620 | >nodeLimit | >nodeLimit | >nodeLimit |
| 90 | \* | 4283 | >nodeLimit | >nodeLimit | 5406 |
| 100 | 20484 | 1941 | >nodeLimit | >nodeLimit | 2779 |
| 110 | 20772 | 2235 | >nodeLimit | >nodeLimit | 2776 |
| 120 | 10382 | 788 | >nodeLimit | 893 | 2514 |
| 130 | \* | 6141 | >nodeLimit | >nodeLimit | 3236 |
| 140 | 27019 | 3988 | >nodeLimit | >nodeLimit | 4351 |
| 150 | \* | 3766 | >nodeLimit | 1226 | 3188 |

\*unknown due to really long execution time.

a)   
During the testing, A\* search will generally find the solution without exceeding the maxNodes limit, however, h1 has generated a significant higher number of nodes than h2. The lower the maxNodes, the higher fraction of puzzles will not be solvable due to exceeding the maxNodes limit, the easier it will be for A\* h1 to exceed maxNodes. Although beam search is not limited by maxNodes, I had to limit it with nodeLimit = 1million since the solution can be pruned in trimming the result list down to length k at each iteraction. I have to stop the search and claim there is no solution after nodeCtr reaches nodeLimit.

b)   
For A\* search, h2 is way better than h1 as we can see it generates a lot less number of nodes than h1. H2 has an advantage because it can further variate 2 similar states by giving them a bigger difference. This allows the program to select the better state to continue. H1 also takes a lot longer to solve each puzzle due to the less efficient heuristic values that did not well-define two very similar states. For Table 1, H1 took XX mins to run while H2 took less than 3 seconds.

c)   
The solution lengths are the same for A\* search because they both have identical structure of finding the best solution and returning it. Beam search, on the opposite, will not produce an optimal solution every time. For smaller k values, most of the puzzles cannot be solved because the solution is more likely to be pruned, as you can see in Table 1. If we increase k to a much larger number, more solutions will be found but most of them are longer than those in A\* searches.

d)   
For A\* h1, all puzzles should be solvable but I did not have time to test out the one marked with an asterisk in table 2 to calculate their no. of nodes generated but they should be a lot bigger than that of A\* h2. However, A\* h1 takes a relatively long time to find the solution.  
For A\*h2, all problems are solvable and they are solved within 3 seconds.   
For beam search, with k = 5, only 2/15 problems I generated are solvable. 13/15 problems cannot be solved because it has reached the nodeLimit, which indicates that the solution might have been pruned and the search is endless without a possible solution showing up.   
With k=30, 7/15 problems are solved since more nodes are saved and explored at each iteration when limiting k states.   
With k=100, 14/15 problems are solved. Therefore, as k gets larger, the more puzzles can be solved but the more space it will use at the same time.

**4) Discussion**  
a) A\* search with h2 seems to be the best search algorithm for this problem. It will find an optimal solution since it continuously compares with previous puzzle states. I chose A\* h2 over A\* h1 because of the amount of space and time A\* h1 will take; and because A\* h1 generates a lot more nodes than h2, it will more prone to reach the maxNodes limit and do not output a solution. A\* h1 and h2 will both generate an optimal/shorter solution for this problem. For beam search, a small k can lead to no solution found since the solution might be pruned; a large k can use up a lot of space. Therefore, in terms of time and space, shorter solutions, A\* h2 is the clear optimal choice.

b) I had difficulties when I try to implement the method “move”. I wasn’t so sure if it should directly change the currentpuzzle to the moved state. However, I found out that I could not do that because I need to use it for generating possible successors and if I changed the current puzzle then there would be no memory of the current parent (the current puzzle will turn into the successor). I can’t generate multiple successors doing that. Therefore, I wrote a method called generateMovedPuzzle that has basically the same functionality of “move” but not only it will not change the current puzzle, it will do a deep copy of it and generate a new puzzle, it will also automatically assign the previousMove to the successor.

As for implementing A\* search, I used quite some time figuring out what should I check with the current successor (does it exist in openList and closeList and what do I do if it does?). I read the book again and again till I found out that it should either replace the previous values or be added to the openList. Also, the Manhattan heuristics is a bit tricky to implement. I had to do some research online in order to get the idea of it.

As for implementing beam search, there isn’t much resources from the book or the Internet. For some reason, nobody talks about local beam search that much. The book doesn’t even have a Pseudocode for it and it only explains it for half a page. I tried my best to understand what the book said and implement it. It wasn’t too difficult though.

Other than the search methods, the required methods for the assignment are pretty straight forward to implement so there are no difficulties encountered there.