Jiaqi Wu

Programming Assignment 1

Design Document

**Introduction**

The 8-puzzle is a toy problem used to exercise search algorithm techniques. For my implementation, I used a breadth first search to find the solution to a puzzle given an input and a defined goal state. In this implementation, a breadth first search was used to search for the goal state of the game. Path length was kept track in order to determine if a better solution was found.

**Design**

The design of this assignment followed the suggested algorithm in the assignment prompt. A tree search algorithm is used to find the solution. Given an initial node, an expansion of this node is generated which represents all of the possible moves that can be made from that given state. Each of these possible moves is appended to a list. Each of the items in the list is then expanded for possible moves from that point. This iterative expansion process eventually leads to the goal state.

Several optimizations are applied in order to facilitate the search of the solution. First of all, if cleansing of the list of paths to explore does not occur, then redundant searches will occur. Upon appending of the expanded states of the present node, all of the previous nodes in this list are checked to see if the state is repeated. If a repeat state is found the cost it took to reach that same node will be compared to see which path is shorter. The shorter path node will be kept while the other will be discarded.

Since the implementation is very simplistic, I will not provide a full class diagram. The only class defined in the solution is the Node, which defines the current state of the search. The implementation was written in python to simplify the deliverables. Here is the class definition and method declarations.

*#Node class which defines the state of the game*

class Node:

#ctor

def \_\_init\_\_(self, state, parent = None, cost = 0):

self.state = state

self.parent = parent

self.cost = cost

*#produces a child node from this node. Provided the arguments for which element to swap with relative to this node and which node is empty,*

*#the return value is a child node with these two indexes swapped and the parent node defined as self and an increased cost*

def \_\_makeChildNode(self, swap\_index, empty\_index):

*#print the current node on the console*

*def print\_node(self):*

*#Returns an expansion of nodes. Checks the path for duplicates and removes them*

def expand(self):

#prints the series of states which leads to the solution

def get\_path(self):

**Discussion**

As part of the assignment, there are two implementation suggestions to consider.

1. *Use “InsertAll” to maintain a sorted “fringe” (possibly using an “insert-sort”) and have “RemoveFront” take the head (or tail) element.*
2. *Build the fringe in an arbitrary way and search for the smallest element when extracting a node to explore.*

The first implementation suggests keeping a sorted fringe. There is a significant advantage to this because a heap can be used to maintain a sorted collection of nodes. This would result in an O(log n*)* operation per every inserted node and a constant time retrieval of the smallest node. However the second implementation would result in an O(n) because a linear search for the smallest element would have to occur for every retrieval of the next node. The first implementation, maintaining a sorted fringe, is clearly must more efficient.

The second item of discussion involves eliminating loops in the search path. In the initial implementation, this step was not checked for. This result in the first and second move looping infinitely and no solution ever being reached. It is essential to eliminate loops because without this step, it is likely that the solution found will not be optimal and will also increase the algorithm running time significantly. Looking at the fringe growth by depth diagram, the complexity of the solution grows exponentially. Minimizing the depth helps with solution running time.

Regarding the how generic my implementation is, this solution is specifically designed for a 3x3 coordinate system. A lot of abstraction needs to occur in order to make the solution generic for a wider variety of problems. The expansion algorithm was not written generically for a wide variety of solutions but specifically for the 8-puzzle. Many parts of the program needs to change in order to be a generic search algorithm solution.