Project: Eight Puzzle

Akshay Kumar, 8 January 2023

Puzzle Background: For this project, students were required to construct a program that is able to solve an 8-puzzle. The 8-puzzle problem was invented by Noyes Palmer Chapman and includes a set of 9 tiles arranged into 3 columns and 3 rows. All of the tiles are numbered 1-8 from left to right and top to bottom. There is one blank space in the puzzle and the objective is to slide the tiles such that the goal state is eventually reached. Players can only move numbers into the blank space and no two numbers may occupy the same tile at any time. Below, Figure 1 shows a completed 8-puzzle.

1	**2**	**3**
4	**5**	**6**
7	**8**	

Figure 1: Example of a completed 8-puzzle

Uniform Cost Search Background: Uniform cost search is a blind search algorithm which always expands the cheapest cost node. In this way, the goal node is reached with the lowest cumulative cost where g(n) is the path cost. This particular search has a hardcoded heuristic cost, h(n) of 0. In the 8-puzzle, each move has the same cost of 1 which is not always the case for other problems.

Misplaced Tile Heuristic Background: A* search with the Misplaced Tile heuristic is an informed search algorithm which expands the node with the cheapest f(n) cost. A heuristic of the number of misplaced tiles is used, not including the blank. Once we count the number of misplaced tiles, this gives us an h(n) cost that is added to the path cost g(n) to obtain f(n) where f(n) = h(n) + g(n).

Figure 2 below shows an 8-puzzle with four tiles misplaced (highlighted in yellow) which gives an h(n) of 4.

1	**3**	**2**
8	**5**	**6**
7	**4**	

Figure 2: 8-puzzle with misplaced tiles

Manhattan Distance Heuristic Background: A* search with the Manhattan Distance heuristic is an informed search algorithm which expands the node with the cheapest f(n) cost. A heuristic is used where the difference between the goal position of the tile and the current position of the tile is calculated. Once again the blank is not included in the calculation. The difference is calculated for each tile and all the values are summed to obtain heuristic cost h(n). This cost is added to the path cost of g(n) to get f(n). Figure 3 below shows the number of moves it takes for tile 7 and tile 3 to reach their goal position which is four for both. Thus, the Manhattan Distance is 4 + 4 = 8.

1	**2**	**7**
4	**5**	**6**
3	**8**	

		3
		↑
3	→	↑

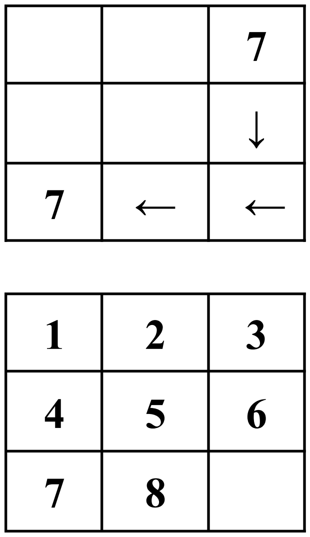


Figure 3: 8-puzzle with arrows showing Manhattan Distance

Algorithm Comparison: As Dr. Keogh explained in his slides, Uniform Cost Search is complete, but is usually not optimal when compared to heuristic algorithms. It occupies a time complexity of O(b^d) and a space complexity of O(b^d). On the other hand, A* search is both complete and optimal, with the time complexity depending on the heuristic, but usually being much less than Uniform Cost Search. However, the space complexity is still O(b^d).

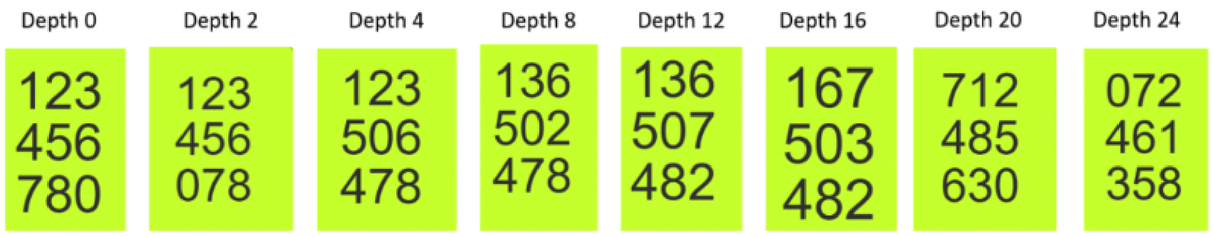
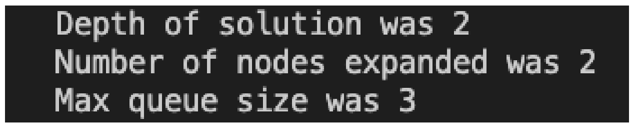
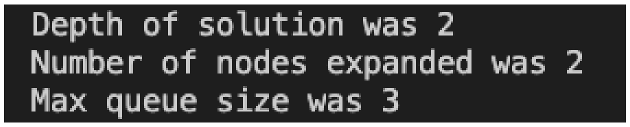


Figure 4: 8-Puzzle samples with given depths for testing purposes



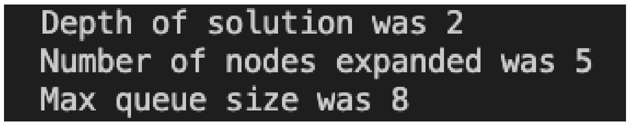
```
Depth of solution was 2
Number of nodes expanded was 2
Max queue size was 3
```

Figure 5: Information shown for Manhattan Distance Heuristic on the depth 2 puzzle



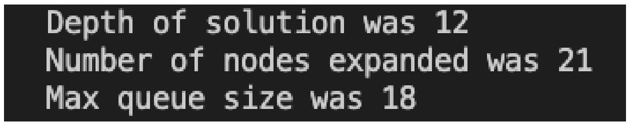
```
Depth of solution was 2
Number of nodes expanded was 2
Max queue size was 3
```

Figure 6: Information shown for Misplaced Tile Heuristic on the depth 2 puzzle



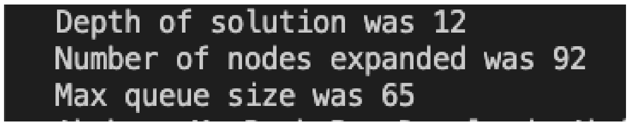
```
Depth of solution was 2
Number of nodes expanded was 5
Max queue size was 8
```

Figure 7: Information shown for Uniform Cost Search on the depth 2 puzzle



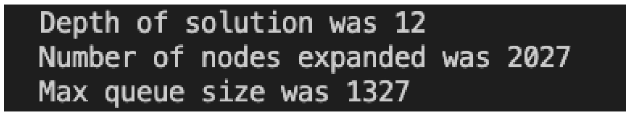
```
Depth of solution was 12
Number of nodes expanded was 21
Max queue size was 18
```

Figure 8: Information shown for Manhattan Distance Heuristic on the depth 12 puzzle



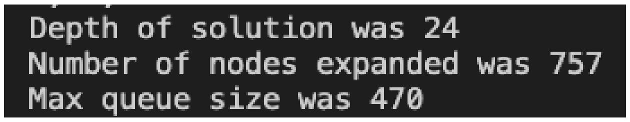
```
Depth of solution was 12
Number of nodes expanded was 92
Max queue size was 65
```

Figure 9: Information shown for Misplaced Tile Heuristic on the depth 12 puzzle



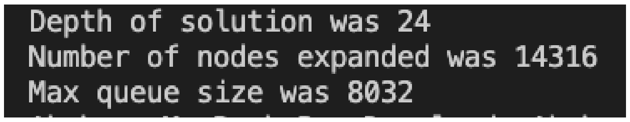
```
Depth of solution was 12
Number of nodes expanded was 2027
Max queue size was 1327
```

Figure 10: Information shown for Uniform Cost Search on the depth 12 puzzle



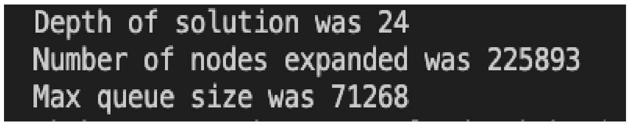
```
Depth of solution was 24
Number of nodes expanded was 757
Max queue size was 470
```

Figure 11: Information shown for Manhattan Distance Heuristic on the depth 24 puzzle



```
Depth of solution was 24
Number of nodes expanded was 14316
Max queue size was 8032
```

Figure 12: Information shown for Misplaced Tile Heuristic on the depth 24 puzzle



```
Depth of solution was 24
Number of nodes expanded was 225893
Max queue size was 71268
```

Figure 13: Information shown for Uniform Cost Search on the depth 24 puzzle

Consider the sample puzzles shown in Figure 4. These were used in testing the performance of each of the three search algorithms. After completing the assignment, it was discovered that the difference between the Uniform Cost Search, the A* Search with Misplaced Tile Heuristic, and the A* Search with Manhattan Distance Heuristic increased with depth. For example, take the puzzle at depth 2 shown in Figure 4. Figure 5 shows the information for the Manhattan Distance heuristic and Figure 6 shows the information for the Misplaced Tile heuristic. As seen, there is no difference between these two algorithms in terms of the number of nodes expanded or max queue size. However, Figure 7 shows that Uniform Cost Search expands more nodes and has a larger max queue size for the same puzzle. The difference between the three algorithms increases when the depth 12 puzzle is solved from Figure 4. Figure 8 shows 21 nodes expanded and a max queue size of 18 for the Manhattan Distance heuristic. Figure 9 shows 92 nodes expanded and a max queue size of 65 for1 the Misplaced Tile heuristic. Figure 10 shows 2027 nodes expanded and a max queue size of 1327 for Uniform Cost Search. Now, the Manhattan Distance heuristic algorithm has become more efficient than the Misplaced Tile heuristic algorithm with the Uniform Cost Search performing much worse than the rest. The difference grows even larger when the depth 24 is solved from Figure 4. Figure 11 shows 757 nodes expanded and a max queue size of 470 for the Manhattan Distance heuristic. Figure 12 shows 14,316 nodes expanded and a max queue size of 8,032 for the Misplaced Tile heuristic. Figure 13 shows a staggering 225,893 nodes expanded with a massive queue size of 71,268 for the Uniform Cost Search. The difference in the number of nodes expanded and max queue size is now in the tens of thousands between the Manhattan Distance A* search and Uniform Cost Search. The same difference is now in the thousands for Manhattan Distance A* search vs. Misplaced Tile Search.

Conclusion: As seen from Figures 5-13, the difference between the three search algorithms grows with depth. Initially, there was only a small difference between the number of nodes expanded as well as max queue size. However, that difference grew immensely as the depth increased from 2 to 24. The importance of heuristic choice can also be seen as the Misplaced Tile algorithm expanded significantly more nodes and had a much larger queue size than Manhattan Distance algorithm . Overall, A* search with the Manhattan Distance heuristic performed the best, closely followed by A* search with the Misplaced Tile heuristic, followed by the Uniform Cost Search.

Resources: To complete this assignment, I used:

- The Blind Search and Heuristic Search lecture slides for the main driver program and for concepts to help me understand the assignment
- https://www.cplusplus.com/ for c++ documentation on various methods and expressions
- https://www.cs.princeton.edu/courses/archive/spr10/cos226/assignments/8puzzle.html for background regarding the 8-puzzle
- The sample project posted in dropbox for report format

(code on following pages)

main.cpp

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <map>
#include <algorithm>
#include <string>
#include "puzzleCase.h"
using namespace std;

vector<int> PuzzleInput(string &myString) // Used for gathering user input for custom
puzzle
{
   getline(cin, myString);
   vector<int> puzzleRow;
   for (unsigned i = 0; i < puzzleSize; ++i)
   {
       puzzleRow.push_back(stoi(myString.substr(i, 1), nullptr, 10)); //Convert each
user entered row from a string to int and then add it to the row
   };
   return puzzleRow;
};

int main()
{
   vector<vector<int> > puzzle; //puzzle is kept as a vector of vector of ints with
each inner vector representing a row in the puzzle
   string userInput;
   bool takingInput = true; //Used to test whether we are done taking input
   string searchType = "";
   priority_queue<puzzleCase> cases;             //Used for selecting the next best
case
   map<vector<vector<int> >, bool> visitedCases; //Used to keep track of already
visited puzzle cases
   unsigned int numNodesExpanded = 0;
   unsigned int maxQSize = cases.size();
   unsigned int solutionDepth = 0;
   while (takingInput)
   {
       cout << "Enter 1 for first preset puzzle, 2 for second preset puzzle, 3 for
third preset puzzle and 4 for custom puzzle " << endl;
       cin >> userInput;
```

```
        if (userInput == "1")
        {
            puzzle = {{1, 2, 3}, {5, 0, 6}, {4, 7, 8}};
            takingInput = false; //We are done taking input
        }
        else if (userInput == "2")
        {
            puzzle = {{1, 3, 6}, {5, 0, 7}, {4, 8, 2}};
            takingInput = false;
        }
        else if (userInput == "3")
        {
            puzzle = {{0, 7, 2}, {4, 6, 1}, {3, 5, 8}};
            takingInput = false;
        }
        else if (userInput == "4")
        {
            cin.ignore();
            cout << "Please enter your 8-puzzle where 0 represents the blank " << endl;
            cout << "Please enter the first row without any spaces: " << endl;
            puzzle.push_back(PuzzleInput(userInput));
            cout << "Please enter the second row without any spaces: " << endl;
            puzzle.push_back(PuzzleInput(userInput));
            cout << "Please enter the third row without any spaces: " << endl;
            puzzle.push_back(PuzzleInput(userInput));
            takingInput = false;
        }
        else
        {
            cout << "Input is invalid. Please either enter 1, 2, 3 or 4." << endl;
            userInput.clear();
        }
    }
    while (searchType != "U" && searchType != "MIS" && searchType != "MAN") //Used for
detecting which algorithm to use, keep going until valid input
    {
        cout << "Please enter your search type (U for Uniform Cost Search, MIS for A*
Misplaced Tile Search, and MAN for A* Manhattan Distance Search:" << endl;
        cin >> searchType;
    }
    puzzleCase newPuzzle = puzzleCase(puzzle); //puzzleCase is a class that handles
each puzzle case and includes the necessary operators for each case
```

```
   newPuzzle.setHCost(searchType);            //Set h(n) cost depending on algorithm
which is passed in as a string
   cases.push(newPuzzle);
   while (cases.size() > 0) //main driver algorithm
   {
       if (cases.size() > maxQSize) //If the queue size is bigger than our set max, we
must update our max
       {
           maxQSize = cases.size();
       }
       puzzleCase currentCase = cases.top();
       cases.pop();
       visitedCases[currentCase.getMyPuzzle()] = 1; //We have now visited this case so
change its bool map value to 1
       cout << "The best state to expand with a g(n) of " << currentCase.getGCost() <<
" and h(n) of " << currentCase.getHCost() << " is:" << endl;
       currentCase.printPuzzleCase();
       if (currentCase.isGoalCase())
       {
           //currentCase.printPuzzleCase();
           cout << "Depth of solution was " << currentCase.getGCost() << endl;
           cout << "Number of nodes expanded was " << numNodesExpanded << endl;
           cout << "Max queue size was " << maxQSize << endl;
           return 0;
       }
       currentCase.expandCase(currentCase, cases, visitedCases, searchType);
//Expanding the current case to get options for next best case
       numNodesExpanded += 1;                                                //We have
now expanded one more node
   }
   cout << "Unsolvable 8-Puzzle" << endl;
   cout << "Number of nodes expanded was " << numNodesExpanded << endl;
   cout << "Max queue size was " << maxQSize << endl;
   return 0;
};
```

puzzleCase.h

```
#ifndef PUZZLE_CASE_H
#define PUZZLE_CASE_H
#include <iostream>
#include <vector>
#include <queue>
#include <map>
#include <utility>

const unsigned puzzleSize = 3; //Used for 8 puzzle since 3 rows and 3 columns, can be
easily changed

class puzzleCase
{
private:
   std::vector<std::vector<int> > myPuzzle;
   unsigned int gCost;
   unsigned int hCost;
   unsigned int blankRowNumber;
   unsigned int blankColNumber;
   std::vector<std::vector<int> > goalPuzzle;
   std::map<puzzleCase, puzzleCase> parent;

public:
   puzzleCase();
   puzzleCase(std::vector<std::vector<int> > aPuzzle);
   void printPuzzleCase();
   unsigned int getGCost() const;
   unsigned int getHCost() const;
   void setGCost();
   void setFCost();
   std::vector<std::vector<int> > getMyPuzzle();
   void setHCost(const std::string &algType);
   unsigned int getFCost() const;
   unsigned int getBlankColNumber();
   unsigned int getBlankRowNumber();
   bool operator<(const puzzleCase &aCase) const;
   bool isGoalCase();
   bool isValidMove(int rowNum, int colNum);
   unsigned locateBlankRowNumber();
   unsigned locateBlankColNumber();
   puzzleCase moveBlankUp(puzzleCase);
```

```cpp
   puzzleCase moveBlankDown(puzzleCase);
   puzzleCase moveBlankLeft(puzzleCase);
   puzzleCase moveBlankRight(puzzleCase);
   bool isGoalState();
   unsigned getManhattanCost();
   unsigned getMisplacedCost();
   std::vector<puzzleCase> getChildCases(puzzleCase firstCase, puzzleCase secondCase,
puzzleCase thirdCase, puzzleCase fourthCase);
   void expandCase(puzzleCase myCase, std::priority_queue<puzzleCase> &cases,
std::map<std::vector<std::vector<int> >, bool> &visitedCases, std::string &algType);
};

#endif
```

puzzleCase.cpp

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <map>
#include <algorithm>
#include <cmath>
#include <cstdlib>
#include "puzzleCase.h"

using namespace std;

puzzleCase::puzzleCase(vector<vector<int> > aPuzzleCase) //Constructor initializing
all the necessary member variables
{
   myPuzzle = aPuzzleCase;
   gCost = 0;                                   //Keeps track of the g(n) cost
   hCost = 0;                                   // Keeps track of the h(n) cost
   blankRowNumber = locateBlankRowNumber(); //Keeps track of where the blank row
number is in order to find it when necessary
   blankColNumber = locateBlankColNumber(); //Keeps track of where the blank col
number is in order to find it when necessary
   goalPuzzle = {{1, 2, 3}, {4, 5, 6}, {7, 8, 0}};
}

unsigned int puzzleCase::locateBlankColNumber() //Used to find the blank column number
{
   int blankCol;
   for (unsigned i = 0; i < myPuzzle.size(); ++i)
   {
       for (unsigned j = 0; j < myPuzzle.at(i).size(); ++j)
       {
           if (myPuzzle.at(i).at(j) == 0)
           {
               blankCol = j;
           }
       }
   }
   return blankCol;
}

unsigned int puzzleCase::locateBlankRowNumber() //Used to find the blank row number
{
   int blankRow;
   for (unsigned i = 0; i < myPuzzle.size(); ++i)
   {
       for (unsigned j = 0; j < myPuzzle.at(i).size(); ++j)
       {
           if (myPuzzle.at(i).at(j) == 0)
           {
               blankRow = i;
           }
       }
   }
   return blankRow;
}

unsigned int puzzleCase::getBlankColNumber() //getter function for grabbing the blank
column number
{
   return blankColNumber;
}
unsigned int puzzleCase::getBlankRowNumber() //getter function for grabbing the blank
row number
{
   return blankRowNumber;
}
```

```
std::vector<std::vector<int> > puzzleCase::getMyPuzzle() //getter function for
grabbing the puzzle
{
   return myPuzzle;
}

puzzleCase puzzleCase::moveBlankUp(puzzleCase myCase) //Used to move the blank tile up
by swapping the blank and numbered tiles
{
   if (myCase.blankRowNumber != 0) // meaning we can move up
   {

       int valueSave = myCase.myPuzzle.at(myCase.blankRowNumber -
1).at(myCase.blankColNumber); // save numbered tile
       myCase.myPuzzle.at(myCase.blankRowNumber - 1).at(myCase.blankColNumber) = 0;
// set value of original position of numbered tile to 0
       myCase.myPuzzle.at(myCase.blankRowNumber).at(myCase.blankColNumber) =
valueSave;          // insert the value of the numbered tile where the original blank
was
       myCase.blankRowNumber--;
// blank is now one more tile up
   }

   return myCase;
}
puzzleCase puzzleCase::moveBlankDown(puzzleCase myCase) //Used to move the blank tile
down by swapping the blank and numbered tiles
{
   if (myCase.blankRowNumber != puzzleSize - 1) // meaning we can move down
   {
       int valueSave = myCase.myPuzzle.at(myCase.blankRowNumber +
1).at(myCase.blankColNumber);
       myCase.myPuzzle.at(myCase.blankRowNumber + 1).at(myCase.blankColNumber) = 0;
       myCase.myPuzzle.at(myCase.blankRowNumber).at(myCase.blankColNumber) =
valueSave;
       myCase.blankRowNumber++; // blank is now one more tile down
   }
   return myCase;
}
puzzleCase puzzleCase::moveBlankLeft(puzzleCase myCase) //Used to move the blank tile
left by swapping the blank and numbered tiles
{
   if (myCase.blankColNumber != 0) // meaning we can move left
   {
       int valueSave =
myCase.myPuzzle.at(myCase.blankRowNumber).at(myCase.blankColNumber - 1);
       myCase.myPuzzle.at(myCase.blankRowNumber).at(myCase.blankColNumber - 1) = 0;
       myCase.myPuzzle.at(myCase.blankRowNumber).at(myCase.blankColNumber) =
valueSave;
       myCase.blankColNumber--; // blank is now one more tile to the left
   }
   return myCase;
}
puzzleCase puzzleCase::moveBlankRight(puzzleCase myCase) //Used to move the blank tile
right by swapping the blank and numbered tiles
{
   if (myCase.blankColNumber != puzzleSize - 1) // meaning we can move right
   {
       int valueSave =
myCase.myPuzzle.at(myCase.blankRowNumber).at(myCase.blankColNumber + 1);
       myCase.myPuzzle.at(myCase.blankRowNumber).at(myCase.blankColNumber + 1) = 0;
       myCase.myPuzzle.at(myCase.blankRowNumber).at(myCase.blankColNumber) =
valueSave;
       myCase.blankColNumber++; // blank is now one more tile to the right
   }
   return myCase;
}

unsigned int puzzleCase::getMisplacedCost() //Used to calculate h(n) for Misplaced
Tile heuristic
{
   unsigned int cost = 0;
   unsigned int comparisonValue = 1; // Used to check if the tile value matches the
tile value of the goal case
   for (unsigned int i = 0; i < myPuzzle.size(); i++)
   {
       for (unsigned int j = 0; j < myPuzzle.at(i).size(); j++)
       {
           if (myPuzzle.at(i).at(j) == 0) // Skip the blank tile
           {
               comparisonValue++;
               continue;
           }
```

```
            if (myPuzzle.at(i).at(j) != comparisonValue) //If a tile is misplaced
then...
            {
                cost++; //We have one more misplaced tile
                comparisonValue++;
            }
            else
            {
                comparisonValue++;
            }
        }
    }
    return cost;
}

unsigned int puzzleCase::getManhattanCost() //Used to calculate h(n) for Manhattan
Distanceheuristic
{
    unsigned int overallCost = 0;
    unsigned int currentCost = 0;
    unsigned int tileValue;
    int goalX;
    int goalY;
    int incrementor = -1; // Used for calculating the current position; has to be -1
because incremented in beginning to 0
    int currentX;
    int currentY;
    for (unsigned int i = 0; i < myPuzzle.size(); i++)
    {
        for (unsigned int j = 0; j < myPuzzle.at(i).size(); j++)
        {
            incrementor++;
            tileValue = myPuzzle.at(i).at(j);
            if (tileValue == 0) //Skip blank space
            {
                continue;
            }
            tileValue--;                                                    // Subtract
because each tile's value is the index + 1
            goalX = tileValue % puzzleSize;                                 //Calculates
the goal column
```

```
            goalY = tileValue / puzzleSize;                              // Calculates
the goal row
            currentX = incrementor % puzzleSize;                         // Calculates
the current column
            currentY = incrementor / puzzleSize;                         // Calculates
the goal row
            currentCost = abs(currentX - goalX) + abs(currentY - goalY); //Calculates
displacement of tile from goal position (Manhattan Distance) for this tile
            overallCost += currentCost;
        }
    }
    return overallCost; //Returns sum of all the distances
}
void puzzleCase::setGCost() // Used to increase g(n) cost when expanding case
{
    gCost++;
}
unsigned int puzzleCase::getGCost() const // getter function for grabbing g(n) cost
{
    return gCost;
}

void puzzleCase::setHCost(const string &algType) //Used to distinguish between which
algorithm as h(n) changes based on algorithm type
{
    if (algType == "MIS")
    {
        hCost = getMisplacedCost();
    }
    else if (algType == "MAN")
    {
        hCost = getManhattanCost();
    }
    else
    {
        hCost = 0; // We have Uniform Cost Search in this case
    }
}
unsigned int puzzleCase::getHCost() const //getter function for grabbing the h(n) cost
{
    return hCost;
}
```

```cpp
unsigned int puzzleCase::getFCost() const //getter function for grabbing the f(n) cost
{
   return getGCost() + getHCost();
}

bool puzzleCase::isGoalCase() //Used to test whether current case is indeed the goal case
{
   for (unsigned i = 0; i < myPuzzle.size(); ++i)
   {
       for (unsigned j = 0; j < myPuzzle.at(i).size(); ++j)
       {
           if (myPuzzle.at(i).at(j) != goalPuzzle.at(i).at(j)) //if current element is not the same as the goal element
           {
               return false;
           }
       }
   }
   return true;
}

void puzzleCase::printPuzzleCase() //Used to print puzzle case for debugging and testing purposes
{

   for (unsigned i = 0; i < myPuzzle.size(); ++i)
   {
       for (unsigned j = 0; j < myPuzzle.at(i).size(); ++j)
       {
           if (j == myPuzzle.size() - 1)
           {
               cout << myPuzzle.at(i).at(j); //Edge numbers don't require commas
           }
           else
           {
               cout << myPuzzle.at(i).at(j) << ", ";
           }
       }
       cout << endl;
   }
}

bool puzzleCase::isValidMove(int rowNum, int colNum) //Tests whether new blank
position is on the puzzle board
{
   if (rowNum < 0 || rowNum >= puzzleSize || colNum < 0 || colNum >= puzzleSize) //If
we are not on the board anymore then...
   {
       return false;
   }
   else
   {
       return true;
   }
}

bool puzzleCase::operator<(const puzzleCase &aCase) const //Used to override priority
queue comparison
{
   int firstCost = getFCost();
   int secondCost = aCase.getFCost();
   return firstCost < secondCost ? false : true; //Used to find the element in queue
with the lowest f(n) cost
}

vector<puzzleCase> puzzleCase::getChildCases(puzzleCase firstCase, puzzleCase
secondCase, puzzleCase thirdCase, puzzleCase fourthCase) //Grabs all the possible
       childCases.push_back(secondCase.moveBlankDown(secondCase)); //Move the blank
down and add the new case to the child cases
   }
   if (isValidMove(thirdCase.blankRowNumber, thirdCase.blankColNumber + 1)) //If we
can move right then...
   {
       childCases.push_back(thirdCase.moveBlankRight(thirdCase)); //Move the blank
right and add the new case to the child cases
   }
   if (isValidMove(fourthCase.blankRowNumber, fourthCase.blankColNumber - 1)) //If we
can move left then...
   {
       childCases.push_back(fourthCase.moveBlankLeft(fourthCase)); //Move the blank
left and add the new case to the child cases
   }
```

```
    return childCases;
}

void puzzleCase::expandCase(puzzleCase aCase, priority_queue<class puzzleCase> &cases, map<vector<vector<int> >, bool> &visitedCases, string &algType) //Handles the expansion of new cases that may lead to solution
{
    vector<puzzleCase> children = getChildCases(aCase, aCase, aCase, aCase); // passes the current case to grab all the possible neighboring cases
    for (int i = 0; i < children.size(); i++)
    {

        if (visitedCases[children.at(i).myPuzzle] == 0) //If we haven't seen this case before...
        {
            children.at(i).setGCost();        //Calculate g(n) for this case
            children.at(i).setHCost(algType); //Calculate h(n) for this case given the algorithm type that we received as a string parameter
            cases.push(children.at(i));       //Add this case to the priority queue as a possible way to reach the solution
        }
    }
}cases where 0 can move to
{
    vector<puzzleCase> childCases;
    if (isValidMove(firstCase.blankRowNumber - 1, firstCase.blankColNumber)) //If we can move up then...
    {
        childCases.push_back(firstCase.moveBlankUp(firstCase)); //Move the blank up and add the new case to the child cases
    }
    if (isValidMove(secondCase.blankRowNumber + 1, secondCase.blankColNumber)) //If we can move down then...
    { childCases.push_back(secondCase.moveBlankDown(secondCase)); //Move the blank down and add the new case to the child cases
    }if (isValidMove(thirdCase.blankRowNumber, thirdCase.blankColNumber - 1)) //If we can move left then...
    { childCases.push_back(thirdCase.moveBlankDown(thirdCase)); //Move the blank left and add the new case to the child cases
    }if (isValidMove(fourthCase.blankRowNumber, fourthCase.blankColNumber + 1)) //If we can move right then...
```

```
   { childCases.push_back(fourthCase.moveBlankRight(fourthCase)); //Move the blank
right and add the new case to the child cases
   }
}
```