**California State University, Fresno**

**Computer Science Department**

**CSCI 264**

**Artificial Intelligence**   
Assignment 1

**To**

**Prof. David Ruby**

**Submitted by**

Sri Ramya Nimmagadda 109492883

Jeevana kala Bommannagari 109504843

Leeladhar Reddy Munnangi 109510225

Yasasvi Yeleswarapu 109475008

Table of Contents

[**Introduction** 3](#_Toc443288400)

[**Breadth First Search(BFS)** 3](#_Toc443288401)

[**Depth First Search (DFS)** 5](#_Toc443288402)

[**A\* Search Algorithm** 6](#_Toc443288403)

[**Program Code** 6](#_Toc443288404)

**Introduction**

Sliding tile puzzle is a [tour puzzle](https://en.wikipedia.org/wiki/Tour_puzzle) that challenges a player to slide (frequently flat) pieces along certain routes (usually on a board) to establish a certain end-configuration. The pieces to be moved may consist of simple shapes, or they may be imprinted with colors, patterns, sections of a larger picture (like a jigsaw puzzle), numbers, or letters.

Sliding puzzles are essentially two-dimensional in nature, even if the sliding is facilitated by mechanically interlinked pieces (like partially encaged marbles) or three-dimensional tokens. As this example shows, some sliding puzzles are [mechanical puzzles](https://en.wikipedia.org/wiki/Mechanical_puzzles). However, the mechanical fixtures are usually not essential to these puzzles; the parts could as well be tokens on a flat board that are moved according to certain rules.

Unlike other tour puzzles, a sliding block puzzle prohibits lifting any piece off the board. This property separates sliding puzzles from [rearrangement puzzles](https://en.wikipedia.org/wiki/Rearrangement_puzzle). Hence, finding moves and the paths opened up by each move within the two-dimensional confines of the board are important parts of solving sliding block puzzles.

**Breadth First Search(BFS)** is an [algorithm](https://en.wikipedia.org/wiki/Algorithm) for traversing or searching [tree](https://en.wikipedia.org/wiki/Tree_data_structure) or [graph](https://en.wikipedia.org/wiki/Graph_(data_structure)) data structures. It starts at the [tree root](https://en.wikipedia.org/wiki/Tree_(data_structure)#Terminology)(or some arbitrary node of a graph, sometimes referred to as a 'search key) and explores the neighbor nodes first, before moving to the next level neighbors.

BFS was invented in the late 1950s by [E. F. Moore](https://en.wikipedia.org/wiki/Edward_F._Moore), who used it to find the shortest path out of a maze, and [discovered independently](https://en.wikipedia.org/wiki/Multiple_discovery) by C. Y. Lee as a [wire routing](https://en.wikipedia.org/wiki/Routing_(electronic_design_automation)) algorithm (published 1961).

**BFS Code:**

Breadth-First-Search (Graph, root):

for each node n in Graph:

n.distance = INFINITY

n.parent = NIL

create empty queue Q

root.distance = 0

Q.enqueue(root)

while Q is not empty:

current = Q.dequeue()

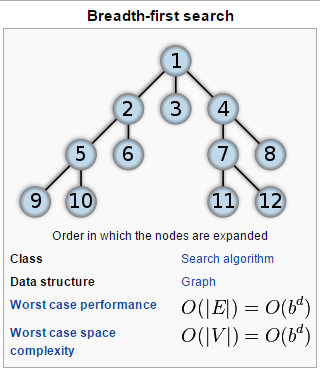
for each node n that is adjacent to current:

if n.distance == INFINITY:

n.distance = current.distance + 1

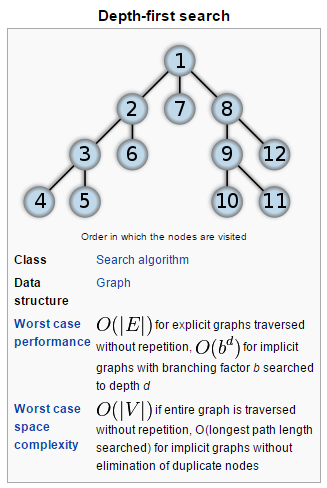
n.parent = current

Q.enqueue(n)



# **Depth First Search (DFS)** is an [algorithm](https://en.wikipedia.org/wiki/Algorithm) for traversing or searching [tree](https://en.wikipedia.org/wiki/Tree_data_structure) or [graph](https://en.wikipedia.org/wiki/Graph_(data_structure)) data structures. One starts at the [root](https://en.wikipedia.org/wiki/Tree_(data_structure)#Terminology)(selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before [back tracking](https://en.wikipedia.org/wiki/Backtracking).

A version of depth-first search was investigated in the 19th century by French mathematician [Charles Pierre Trémaux](https://en.wikipedia.org/wiki/Charles_Pierre_Tr%C3%A9maux) as a strategy for solving mazes.



# **A\* Search Algorithm**

**A\*** (pronounced as "A star" is a [computer algorithm](https://en.wikipedia.org/wiki/Computer_algorithm) that is widely used in [pathfinding](https://en.wikipedia.org/wiki/Pathfinding) and [graph traversal](https://en.wikipedia.org/wiki/Graph_traversal), the process of plotting an efficiently traversable path between multiple points, called nodes. Noted for its [performance](https://en.wikipedia.org/wiki/Computer_performance) and accuracy, it enjoys widespread use. However, in practical travel-routing systems, it is generally outperformed by algorithms which can pre-process the graph to attain better performance, although other work has found A\* to be superior to other approaches.

# **Program Code**

class tiles

{

     public static int newTable [] ;

     public static int table [] = {2,3,1,4,5,0,6,7,8};

     public static int goalTable[] ={0,1,2,3,4,5,6,7,8};

     public static int blankSpot;

     public static int a=0, b=0, c=0, x;

     public static int numCorrect=0;

     public static int origNum, leftNum,rightNum,upNum,downNum;

        public static void main(String args[])

              {

               puzzSolver();

       }

//Solves puzzle

     static void puzzSolver()

     {

       int a;

       printinitvalues();        //Prints the puzzle

       for (a=0; a<5; ++a)

             {

               countTiles();

               branch();

               System.out.println(numCorrect);

             }

     }

//Solves puzzle

     static void branch()

     {

       if (numCorrect<9)

             {

               countTiles();

               locateSpace();              //Locates the position of the blank space

               //System.out.println(numCorrect);

               checkNum();

               getLarge();

               //System.out.println(x);

               if(x==leftNum)

                     changeTableLeft();

                       else if(x==rightNum)

                             changeTableRight();

                               else if(x==upNum)

                                     changeTableUp();

                                       else

                                             changeTableDown();

               printinitvalues();

             }

     }

//Counts tiles in correct placement

     static void countTiles()

     {

       int i;

       numCorrect =0;

       for (i=0; i<9; ++i)

             {

               if (newTable[i]==goalTable[i])

               {

                     numCorrect = numCorrect + 1;

               }

             }

     }

//Check correct placement after each possible move

     static void checkNum()

     {

       makeMoveLeft();

       //printNewValues();              //Prints the puzzle

       locateSpace();              //Locates the position of the blank space

       countTiles();

       leftNum = numCorrect;

       //System.out.println(leftNum);

       resetTable();

       makeMoveUp();

       //printNewValues();              //Prints the puzzle

       locateSpace();              //Locates the position of the blank space

       countTiles();

       upNum = numCorrect;

       //System.out.println(upNum);

       resetTable();

       makeMoveRight();

       //printNewValues();              //Prints the puzzle

       locateSpace();              //Locates the position of the blank space

       countTiles();

       rightNum = numCorrect;

       //System.out.println(rightNum);

       resetTable();

       makeMoveDown();

       //printNewValues();              //Prints the puzzle

       locateSpace();              //Locates the position of the blank space

       countTiles();

       downNum = numCorrect;

       //System.out.println(downNum);

       resetTable();

     }

//Checks which move made greatest impact

     static void getLarge()

     {

       x=leftNum;

       if (x<rightNum)

             {

               x=rightNum;

             }

       if (x<upNum)

             {

               x=upNum;

             }

       if (x<downNum)

             {

               x=downNum;

             }

     }

     static void changeTableLeft()

     {

       makeMoveLeft();

       int a;

       for (a=0; a<9; ++a)

       {

             table[a] = newTable[a];

       }

     }

     static void changeTableRight()

     {

       makeMoveRight();

       int a;

       for (a=0; a<9; ++a)

             {

               table[a] = newTable[a];

             }

     }

     static void changeTableDown()

     {

       makeMoveDown();

       int a;

       for (a=0; a<9; ++a)

             {

               table[a] = newTable[a];

             }

     }

     static void changeTableUp()

     {

       makeMoveUp();

       int a;

       for (a=0; a<9; ++a)

             {

               table[a] = newTable[a];

             }

     }

//Makes moves of blank tiles to the left--does error checking to ensure move is allowed

     static void makeMoveLeft()

     {

       if(blankSpot!=0)

       {

             if (blankSpot !=3)

             {

               if (blankSpot !=6)

               {

                     int temp;

                     temp = table[blankSpot-1];

                       if (temp != goalTable[blankSpot-1])

                             {

                               newTable[blankSpot-1]=table[blankSpot];

                               newTable[blankSpot] = temp;

                               countTiles();

                             }

               }

             }

       }

       //else makeMoveUp();

     }

//Makes moves of blank tiles to right

     static void makeMoveRight()

     {

       if(blankSpot!=2)

             {

               if (blankSpot !=5)

                     {

                       if (blankSpot !=8)

                             {

                               int temp;

                               temp = table[blankSpot+1];

                               if (temp != goalTable[blankSpot+1])

                                     {

                                       newTable[blankSpot+1]=table[blankSpot];

                                       newTable[blankSpot] = temp;

                                       return;

                                     }

                             }

                     }

             }

     }

//Makes moves of blank tiles up

     static void makeMoveUp()

     {

       if(blankSpot!=0)

             {

               if (blankSpot !=1)

                     {

                       if (blankSpot !=2)

                             {

                               int temp;

                               temp = table[blankSpot-3];

                               if (temp != goalTable[blankSpot-3])

                                     {

                                       newTable[blankSpot-3]=table[blankSpot];

                                       newTable[blankSpot] = temp;

                                     }

                             }

                     }

             }

     }

//Makes moves of blank tiles down

     static void makeMoveDown()

     {

       if(blankSpot!=6)

             {

               if (blankSpot !=7)

                     {

                       if (blankSpot !=8)

                             {

                               int temp;

                               temp = table[blankSpot+3];

                               if (temp != goalTable[blankSpot+3])

                                     {

                                       newTable[blankSpot+3]=table[blankSpot];

                                       newTable[blankSpot] = temp;

                                     }

                             }

                     }

             }

     }

/\*I used this method to print the values that are stored for the puzzle\*/

     static void printinitvalues()

       {

             int t, i=1;

             for (t=0; t<9; ++t)

               {

                     System.out.print(table [t] + "  ");

                       if (i == 3)                              //I use this loop to create a new row

                             {

                               System.out.println();

                               i = i - 3;

                             }

                     i = i + 1;

               }

                     System.out.println();

                     System.out.println();

       }

//Printing the Goal State

     static void printGoalvalues()

     {

             int t, i=1;

             for (t=0; t<9; ++t)

               {

                     System.out.print(goalTable [t] + "  ");

                       if (i == 3)                           //I use this loop to create a new row

                             {

                               System.out.println();

                               i = i - 3;

                             }

                     i = i + 1;

               }

                     System.out.println();

                     System.out.println();

     }

//Prints the modified table

     static void printNewValues()

       {

             int t, i=1;

             for (t=0; t<9; ++t)

               {

                     System.out.print(newTable [t] + "  ");

                       if (i == 3)                          //I use this loop to create a new row

                             {

                               System.out.println();

                               i = i - 3;

                             }

                     i = i + 1;

               }

                     System.out.println();

                     System.out.println();

       }

//I used this method to locate the blank space in the puzzle

     static void locateSpace()

     {

       int t;

       for (t=0; t<9; ++t)

             {

               if (table[t]==0)

                     {

                       blankSpot = t;

                       return;

                     }

             }

     }

//End of method

//Resets the testing table

     static void resetTable()

     {

       int i;

       for (i=0; i<9; ++i)

       {

             newTable[i]=table[i];

       }

     }