N-Puzzle Game Project Report

From

Gokul Kaisaravalli Bhojraj
Id: 80789
Program: Business Intelligence
e-mail: h18gokka@du.se

Under the Guidance of

Pascal Rebreyend
e-mail: prb@du.se

ABSTRACT

In this project I have been able to solve n tiles puzzle, where n can be 3 or 8 or 15. It can solve the puzzle manually or automatically by using different algorithms Techniques.

OVERVIEW

The entire project will be able to perform the following tasks

• Execute in the command prompt

• Have a menu-system with at least following options:

  o Quit the program.

  o Create a new n-puzzle of a selectable size of n.

  o Mix the tiles by automatic move around them.

  o Manually solve the puzzle by move tile after tile.

  o Read in a state for an n-puzzle and assign this state as the start-state or goal state for automatically solving the puzzle from a file.

  o It is possible to select the type of search, breath, depth or best first search. If the user selects depth first search, it will be possible to limit the depth of the search. The best first search uses heuristic implementation.

• It Consists of classes that represent; the menu system, the solver, breadth first search, depth first search, best first search, puzzle and the program.

CONTENTS

Chapter 1

Introduction

The project is created in Java using NetBeans. We can create a puzzle with size n, where n can be 2*2, 3*3 or 4*4. Here it is possible to solve the puzzle manually by moving the tiles around. If you want to solve the puzzle automatically its possible by choosing the appropriate search technique to find the solution. If you want to import the puzzle from a file its possible as well. After importing its even possible to set it as the start state of the puzzle or you can set its as goal state.

To solve the puzzle automatically we make use of graph, state and tree to have an intuitive way of solving. The puzzle can be solved in 4 ways,

- Manually
- Breadth First Search
- Depth First Search
- Heuristics (A*)

By using any of this technique it's possible to obtain the solution to the given puzzle state.

Chapter 2

Class Operations

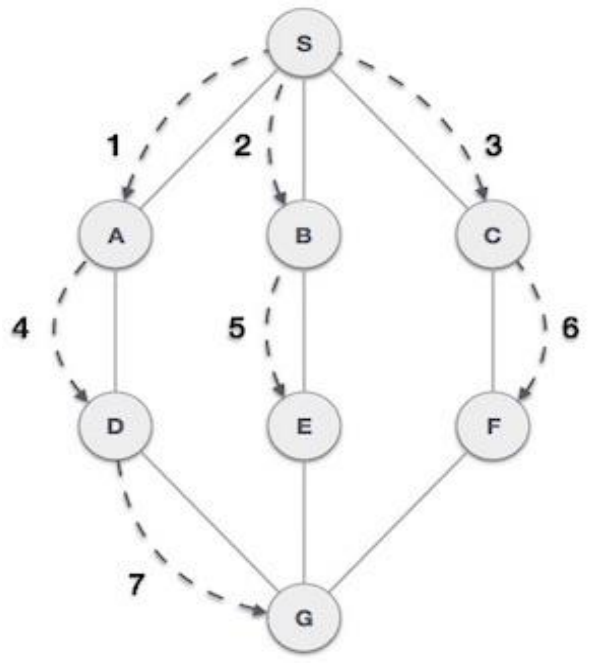
Each of the following classes perform the following operations:

- **Program:** This class has the main function in it, where it creates the object for the Menu class and from which the objects for rest of the classes are created.
- **Menu:** This class is useful to create the menu that gives different operations that can be performed.
- **Puzzle:** This is an important class which creates the structure to our puzzle, it is also responsible for various modifications to our puzzle as well.
- **Solver:** This is a class useful to invoke our heuristics operation class and some other operations required for the representations of the puzzle.
- **BFS:** It is a class useful for solving the puzzle using Breadth First Search technique, along with giving the best steps that we can use to solve the puzzle and it solves the puzzle automatically.
- **DFS:** It is a class useful for solving the puzzle using Depth First Search technique, along with giving the steps that it has used to solve the puzzle, but it will not be the best path to solve the puzzle and also it involves a lot of steps to solve , but it solves the puzzle automatically for us if the puzzle is solvable within the given depth.
- **Heuri:** This is the class that is useful to solve the puzzle in a best way by making use of Manhattan Distance strategy along with the A* algorithm implementations. It gives the bets path to solve the puzzle always and it also displays the path and the time to solve as well.

Chapter 3

Search Implementations

BFS: Breadth First Search (BFS) algorithm traverses a graph in a breadth ward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

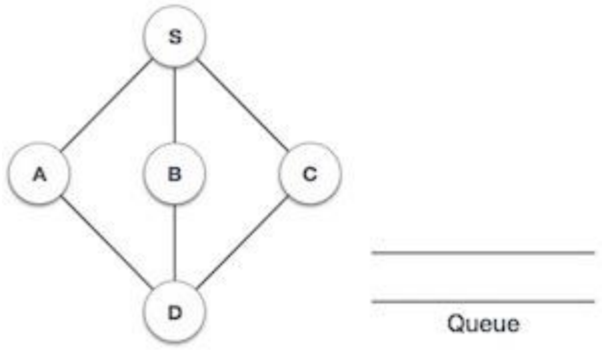


As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- **Rule 2** – If no adjacent vertex is found, remove the first vertex from the queue.
- **Rule 3** – Repeat Rule 1 and Rule 2 until the queue is empty

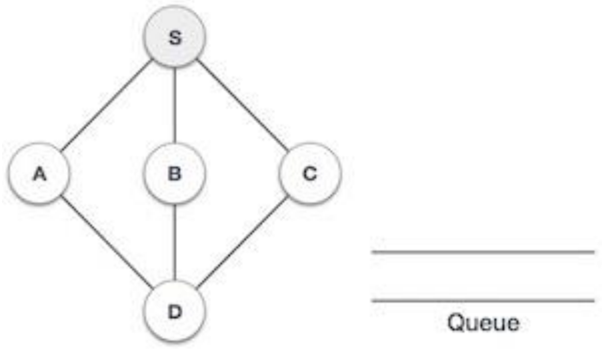
Steps:

1)



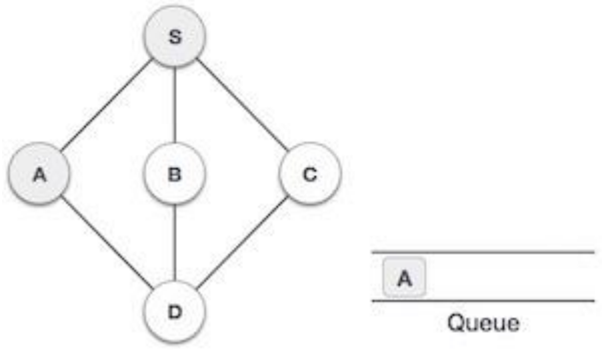
Initialize the queue.

2)



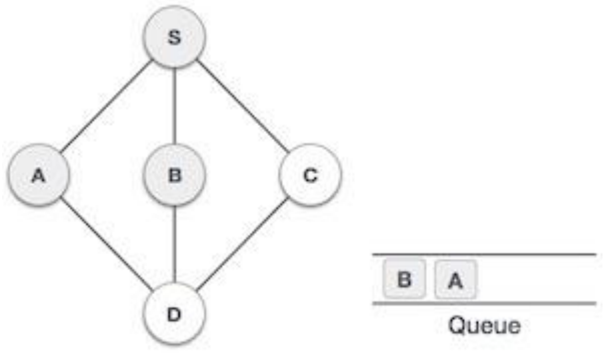
We start from visiting **S** (starting node), and mark it as visited

3)



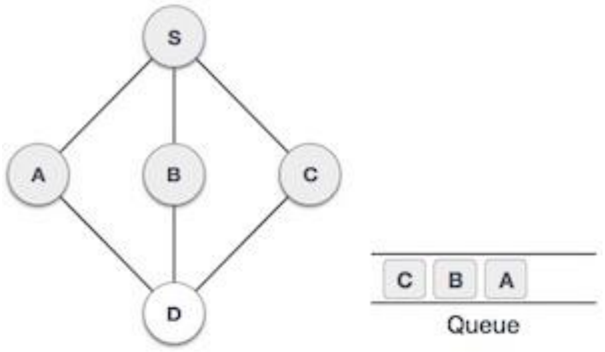
We then see an unvisited adjacent node from **S.** In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it.

4)



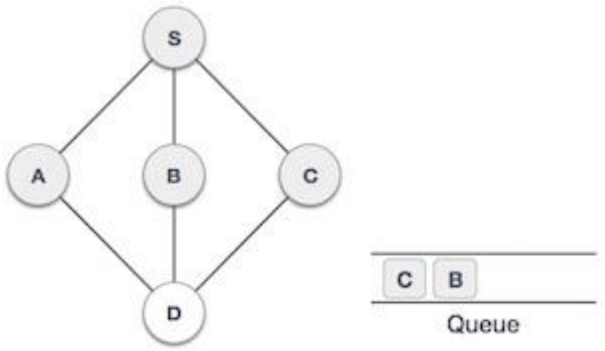
Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it.

5)



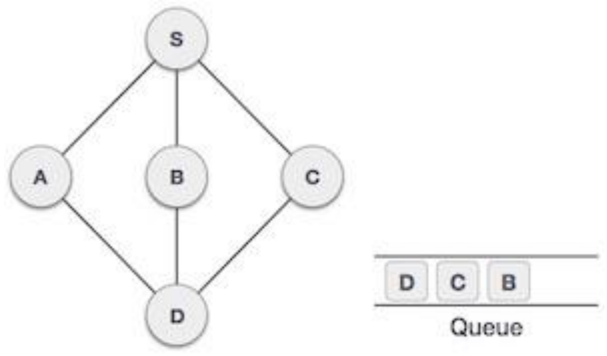
Next, the unvisited adjacent node from **S** is **C**. We mark it as visited and enqueue it.

6)



Now, **S** is left with no unvisited adjacent nodes. So, we dequeue and find **A.**

7)



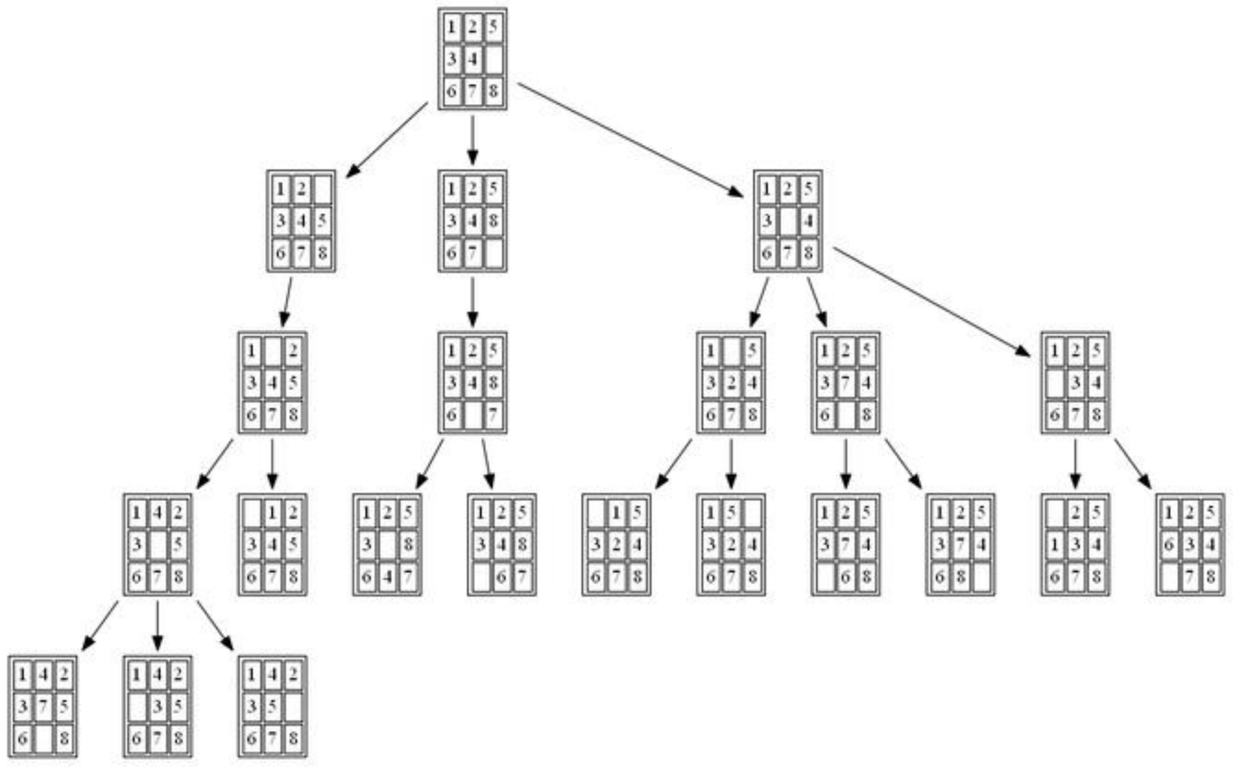
From **A** we have **D** as unvisited adjacent node. We mark it as visited and enqueue it.

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

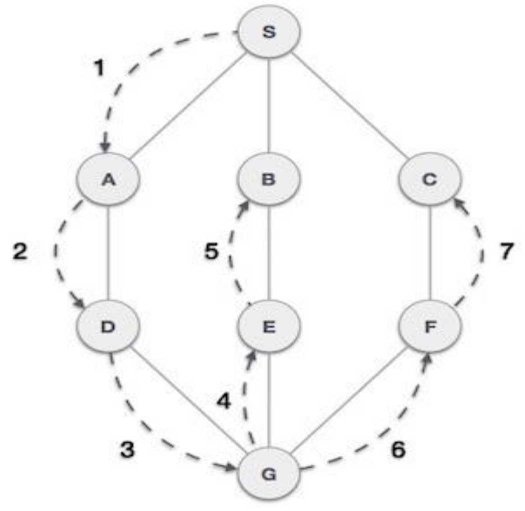
Algorithm Used:

```
begin
open:= [Start];                                                    % initialize
closed:= [ ];
while open ≠ [ ] do                                                % states remain
begin
remove leftmost state from open, call it X;
if X is a goal then return SUCCESS                                 % goal found
else begin
generate children of X;
put X on closed;
discard children of X if already on open or closed;               % loop check
put remaining children on right end of open                        % queue
end
end
return FAIL % no states left
end.
```

Each of the nodes represents a state of our puzzle and the search continues until we reach the goal state.



DFS: Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

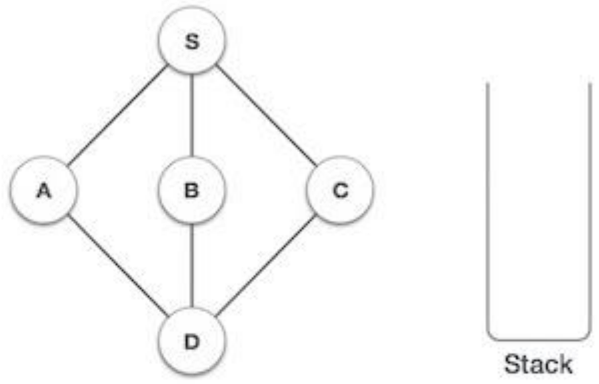


As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** – Repeat Rule 1 and Rule 2 until the stack is empty

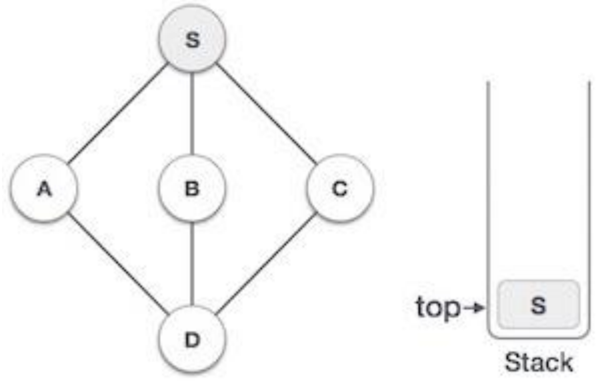
Steps:

1)



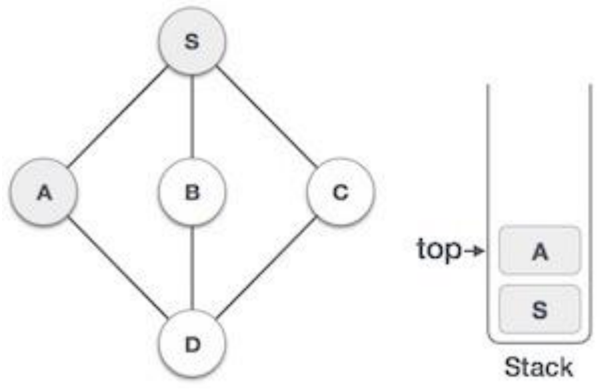
Initialize the stack.

2)



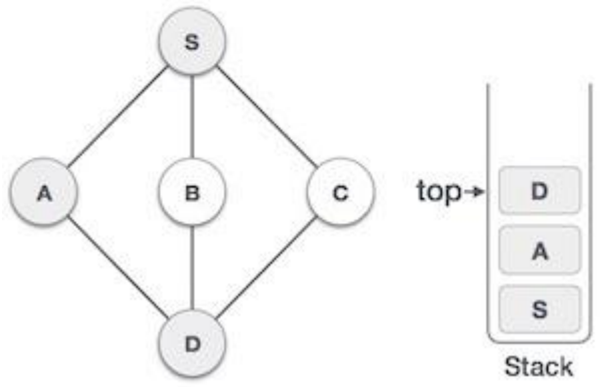
Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.

3)



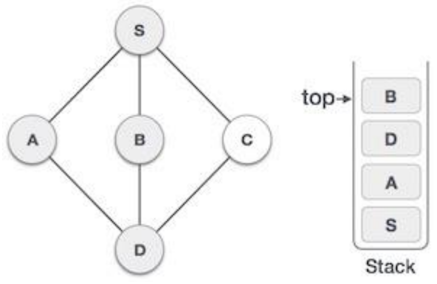
Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S**and **D** are adjacent to **A** but we are concerned for unvisited nodes only.

4)



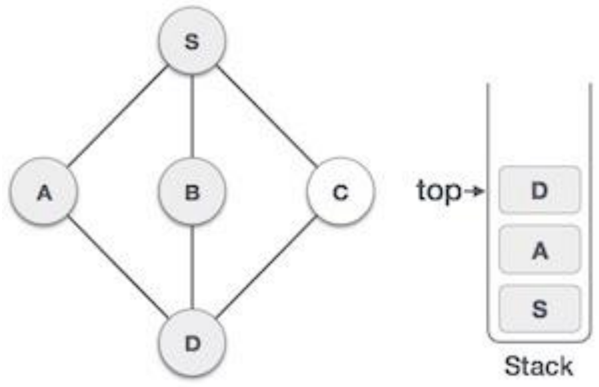
Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.

5)



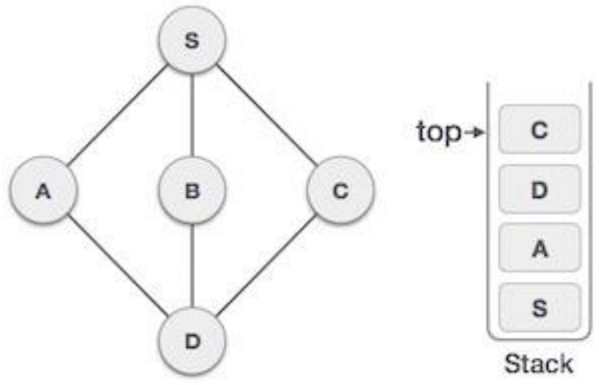
We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack.

6)



We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack

7)



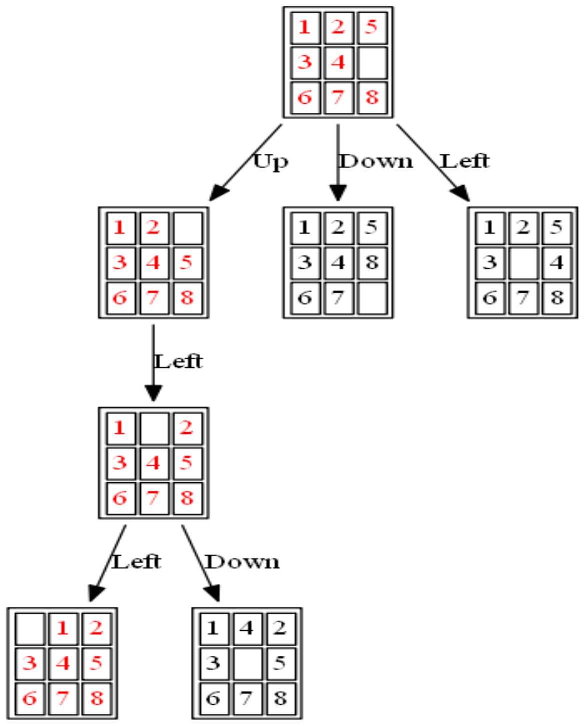
Only unvisited adjacent node is from **D** is **C** now. So, we visit **C**, mark it as visited and put it onto the stack.

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

Algorithm Used:

```
begin
open := [Start];                                          % initialize
closed := [ ];
while open ≠ [ ] do                                       % states remain
begin
remove leftmost state from open, call it X;
if X is a goal then return SUCCESS                        % goal found
else begin
generate children of X;
put X on closed;
discard children of X if already on open or closed;       % loop check
put remaining children on left end of open                % stack
end
end;
return FAIL                                               % no states left
end.
```

Each of the nodes represents a state of our puzzle and the search continues until we reach the goal state.



Continues until the solution is found (Goal).

Chapter 4

Heuristics

Heuristic search is a technique that uses heuristic value for optimizing the search. Here we make use of A* algorithm.

A* Algorithm

A* is a computer algorithm that is widely used in pathfinding and graph traversal, the process of plotting an efficiently traversable path between multiple points, called nodes. Noted for its performance and accuracy, it enjoys widespread use.

The key feature of the A* algorithm is that it keeps a track of each visited node which helps in ignoring the nodes that are already visited, saving a huge amount of time. It also has a list that holds all the nodes that are left to be explored and it chooses the most optimal node from this list, thus saving time not exploring unnecessary or less optimal nodes.

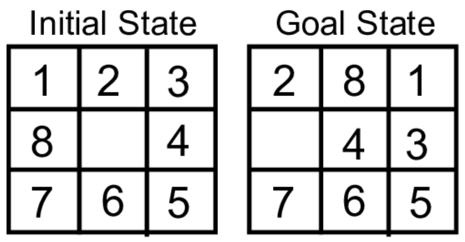
So we use two lists namely ‘open list‘ and ‘closed list‘ the open list contains all the nodes that are being generated and are not existing in the closed list and each node explored after it’s neighboring nodes are discovered is put in the closed list and the neighbors are put in the open list this is how the nodes expand. Each node has a pointer to its parent so that at any given point it can retrace the path to the parent. Initially, the open list holds the start(Initial) node. The next node chosen from the open list is based on its f score, the node with the least f score is picked up and explored.

f-score = h-score + g-score

A* uses a combination of heuristic value (h-score: how far the goal node is) as well as the g-score (i.e. the number of nodes traversed from the start node to current node).

In our n-Puzzle problem, we can define the h-score as the number of misplaced tiles by comparing the current state and the goal state or summation of the Manhattan distance between misplaced nodes.

g-score will remain as the number of nodes traversed from start node to get to the current node.



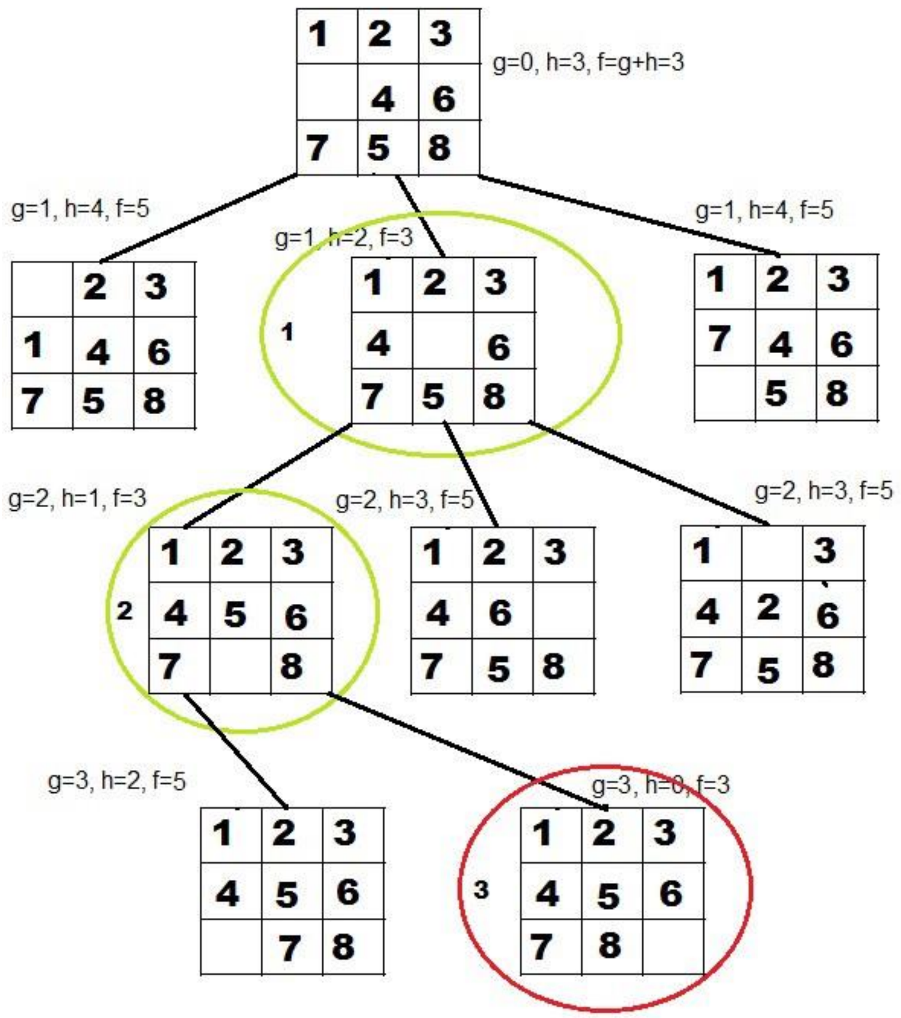
From Fig 1, we can calculate the h-score by comparing the initial(current) state and goal state and counting the number of misplaced tiles.

Thus, h-score = 5 and g-score = 0 as the number of nodes traversed from the start node to the current node is 0.

We first move the empty space in all the possible directions in the start state and calculate f-score for each state. This is called expanding the current state.

After expanding the current state, it is pushed into the closed list and the newly generated states are pushed into the open list. A state with the least f-score is selected and expanded again. This process continues until the goal state occurs as the current state. Basically, here we are providing the algorithm a measure to choose its actions. The algorithm chooses the best possible action and proceeds in that path.

This solves the issue of generating redundant child states, as the algorithm will expand the node with the least f-score.



Chapter 5

Usage

Each of the option that you choose performs the following tasks

Note: If the puzzle is read from the file and is set to be the start state, it should be in a solvable condition if not the automatic solving techniques doesn't work. But if the puzzle is chosen with in the program, then it will be solvable always.

1. Create a new puzzle:

   Creates a new puzzle.

2. Read Puzzle from File:

   a) Reads the puzzle form a given file.

   b) Sets the read puzzle to either start state or the goal state.

   Note: The puzzle that is been read from the file should match the size of the puzzle that's been chosen before.

3. Manually move the tiles:

   Used for solving the puzzle manually.

4. Mix the puzzle:

   Used for mixing of the puzzle (shuffling).

5. Solve by breadth first search.

   Used to solve the puzzle automatically by using BFS technique.

6. Solve by depth first search:

   Used to solve the puzzle automatically by using DFS technique.

Note: Provides an option to choose the depth or you can give unlimited depth. The solving will be successful if the given depth is enough and the memory is enough to solve the puzzle. If either of it is not satisfied, then the program terminates.

7. Solve by Best first search/Heuristics:

Used for solving the puzzle automatically using A* algorithm/ Best First Search.

8. START STATE:

Used to display the current start state of the Puzzle.

9. Goal STATE:

Used to display the current Goal state of the Puzzle.

10. Quit the program.

Used to terminates the program.

Chapter 6

Results

Start state and Goal state that are used below, are as per the given test puzzle.

1) Heuristics:

If

Start state:					**Goal state:**				
	1	**2**	**4**	**7**		**1**	**2**	**3**	**4**
	5	**6**	**3**	**0**		**5**	**6**	**7**	**8**
	10	**13**	**11**	**8**		**9**	**10**	**11**	**12**
	9	**14**	**15**	**12**		**13**	**14**	**15**	**0**

Solution:

```
p = 3 = g+h = 0+3
1 2 4 7
5 6 3 0
10 13 11 8
9 14 15 12
success

p = 4 = g+h = 1+3
1 2 4 0
5 6 3 7
10 13 11 8
9 14 15 12
success

p = 5 = g+h = 2+3
1 2 0 4
5 6 3 7
10 13 11 8
9 14 15 12
success

p = 6 = g+h = 3+3
1 2 3 4
5 6 0 7
10 13 11 8
9 14 15 12
success
p = 7 = g+h = 4+3
1 2 3 4
5 6 7 0
10 13 11 8
9 14 15 12
success
```

```
p = 8 = g+h = 5+3
1 2 3 4
5 6 7 8
10 13 11 0
9 14 15 12
success

p = 9 = g+h = 6+3
1 2 3 4
5 6 7 8
10 13 11 12
9 14 15 0
success

p = 10 = g+h = 7+3
1 2 3 4
5 6 7 8
10 13 11 12
9 14 0 15
success

p = 11 = g+h = 8+3
1 2 3 4
5 6 7 8
10 13 11 12
9 0 14 15
success

p = 12 = g+h = 9+3
1 2 3 4
5 6 7 8
10 0 11 12
9 13 14 15
success

p = 13 = g+h = 10+3
1 2 3 4
5 6 7 8
0 10 11 12
9 13 14 15
success

p = 14 = g+h = 11+3
1 2 3 4
5 6 7 8
9 10 11 12
0 13 14 15
success


p = 15 = g+h = 12+3
1 2 3 4
5 6 7 8
9 10 11 12
13 0 14 15
success
```

```
p = 16 = g+h = 13+3
1 2 3 4
5 6 7 8
9 10 11 12
13 14 0 15
success

p = 17 = g+h = 14+3
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 0
Success

elapsed (ms) = 129

 ******SOLVED PUZZLE******

  1  2  3  4
  5  6  7  8
  9 10 11 12
 13 14 15 _
```

2) BFS:

If

Start state:					**Goal state:**				
	1	**2**	**4**	**7**		**1**	**2**	**3**	**4**
	5	**6**	**3**	**0**		**5**	**6**	**7**	**8**
	10	**13**	**11**	**8**		**9**	**10**	**11**	**12**
	9	**14**	**15**	**12**		**13**	**14**	**15**	**0**

Solution:

```
Solution Exists at Level 14 of the tree

 At 14

 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15  0

 At 13

 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14  0 15
```

```
 At 12

 1  2  3  4
 5  6  7  8
 9 10 11 12
13  0 14 15

 At 11

 1  2  3  4
 5  6  7  8
 9 10 11 12
 0 13 14 15

 At 10

 1  2  3  4
 5  6  7  8
 0 10 11 12
 9 13 14 15

 At 9

 1  2  3  4
 5  6  7  8
10  0 11 12
 9 13 14 15

 At 8

 1  2  3  4
 5  6  7  8
10 13 11 12
 9  0 14 15

 At 7

 1  2  3  4
 5  6  7  8
10 13 11 12
 9 14  0 15

 At 6

 1  2  3  4
 5  6  7  8
10 13 11 12
 9 14 15  0


 At 5

 1  2  3  4
 5  6  7  8
10 13 11  0
 9 14 15 12
 At 4

 1  2  3  4
 5  6  7  0
10 13 11  8
 9 14 15 12

 At 3

 1  2  3  4
 5  6  0  7
10 13 11  8
 9 14 15 12

 At 2

 1  2  0  4
 5  6  3  7
10 13 11  8
 9 14 15 12

 At 1

 1  2  4  0
 5  6  3  7
10 13 11  8
 9 14 15 12

 At 0

 1  2  4  7
 5  6  3  0
10 13 11  8
 9 14 15 12

 ******SOLVED PUZZLE******

  1  2  3  4
  5  6  7  8
  9 10 11 12
 13 14 15  _
```

3) DFS:

If

Start state:

1	2	4	7
5	6	3	0
10	13	11	8
9	14	15	12

Goal state:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	0

Solution:

```
 1  2  4  7
 5  6  3  0
10 13 11  8
 9 14 15 12


.

.

.

.

.

.



 1  2  3  4
 5  0  7  8
 9  6 11 12
13 10 14 15

 1  2  3  4
 5  6  7  8
 0  9 11 12
13 10 14 15

 1  2  3  4
 5  6  7  8
 9 11  0 12
13 10 14 15

 1  2  3  4
 5  6  7  8
 9 10  0 12
13 14 11 15

 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15  0
```

Depth=614398

SUCCESS

Permitted Depth :Unlimited

```
******SOLVED PUZZLE******

 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15  _
```

(Not all the steps are included in this result of DFS).