Assignment No : 01 and 02
Assignment name : 8 puzzle problem and Best-First search in Graph representation problem solving

CSE-0408 Summer 2021

Farhana Rahman
Department of Computer Science and Engineering
State University of Bangladesh (SUB)
Dhaka, Bangladesh
swapnarahman40@gmail.com

***Abstract*—Code 1 : In practice, an incomplete heuristic search nearly always finds better solutions if it is allowed to search deeper, i.e. expand and heuristically evaluate more nodes in the search tree and determine the best path to take next. The heuristic function is a way to inform the search about the direction to a goal. It provides an informed way to guess which neighbor of a node will lead to a goal. There is nothing magical about a heuristic function. It must use only information that can be readily obtained about a node.**

Code 2 : As discussed earlier, Breadth-First Search (BFS) is an algorithm used for traversing graphs or trees. Traversing means visiting each node of the graph. Breadth-First Search is a recursive algorithm to search all the vertices of a graph or a tree. BFS in python can be implemented by using data structures like a dictionary and lists. Breadth-First Search in tree and graph is almost the same. The only difference is that the graph may contain cycles, so we may traverse to the same node again.

***Index Terms*—heuristic, puzzle traverse, cycle, graph**

I. Introduction

Code 1 : Breadth First Traversal (or Search) for a graph is similar to Breadth First Traversal of a tree (See method 2 of this post). The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex. For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Breadth First Traversal of the following graph is 2, 0, 3, 1.

Code 2 : Many problems, such as game-playing and path-finding, can be solved by search algorithms. To do so, the problems are represented by a search graph or tree in which the nodes correspond to the states of the problem. In this assignment we are going to implement a algorithms to solve 8 puzzle problem.

II. Literature Review

Code 1 : Sadikov and Bratko (2006) studied the suitability of pessimistic and optimistic heuristic functions for a real-time search in the 8-puzzle. They discovered that pessimistic functions are more suitable. They also observed the pathology, which was stronger with the pessimistic heuristic function. However, they did not study the influence of other factors on the pathology or provide any analysis of the gain of a deeper search.

Code 2 :The two variants of Best First Search are Greedy Best First Search and A* Best First Search. Greedy BFS: Algorithm selects the path which appears to be the best, it can be known as the combination of depth-first search and breadthfirst search. Greedy BFS makes use of Heuristic function and search and allows us to take advantages of both algorithms. A* BFS: Is an informed search algorithm, or a best-first search, meaning that it is formulated in terms of weighted graphs: starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost (least distance travelled, shortest time, etc.).

III. Proposed Methodology

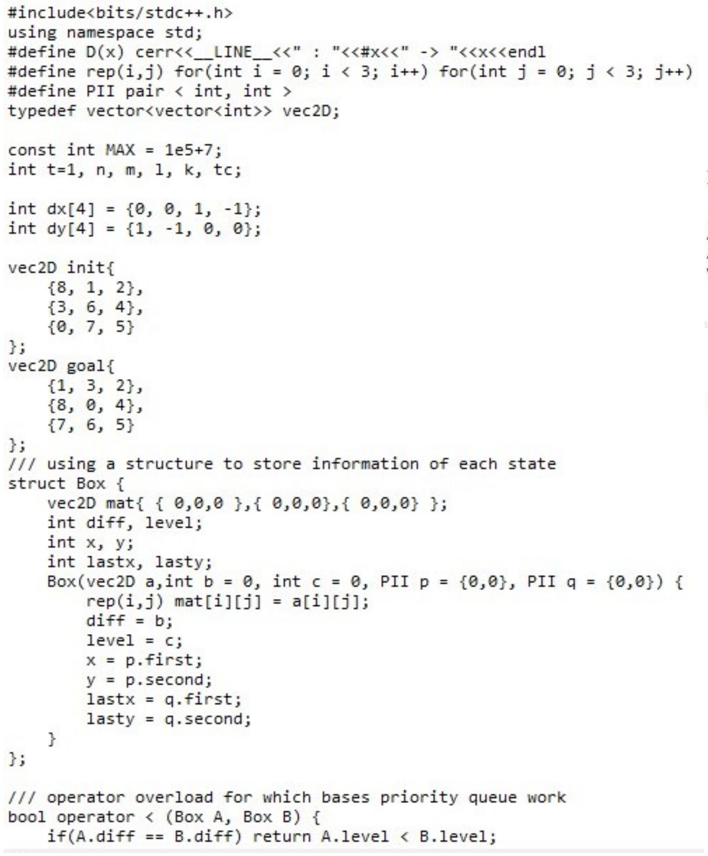
Code 1 : The 8-puzzle problem is a puzzle invented and popularized by Noyes Palmer Chapman in the 1870s. It is played on a 3-by-3 grid with 8 square blocks labeled 1 through 8 and a blank square. Your goal is to rearrange the blocks so that they are in order.

Code 2: Step 1: Choose the starting node and insert it into queue Step 2: Find the vertices that have direct edges with the vertex(node) Step 3: Insert all the vertices found in step 3 into queue Step 4: Remove the first vertex(node) in queue Step 5: Continue this process until all the vertices are visited

IV. Conclusion

Code 1 : We tested our code to see how many states it would take to get from the current state to the goal state, and we came up with seven.

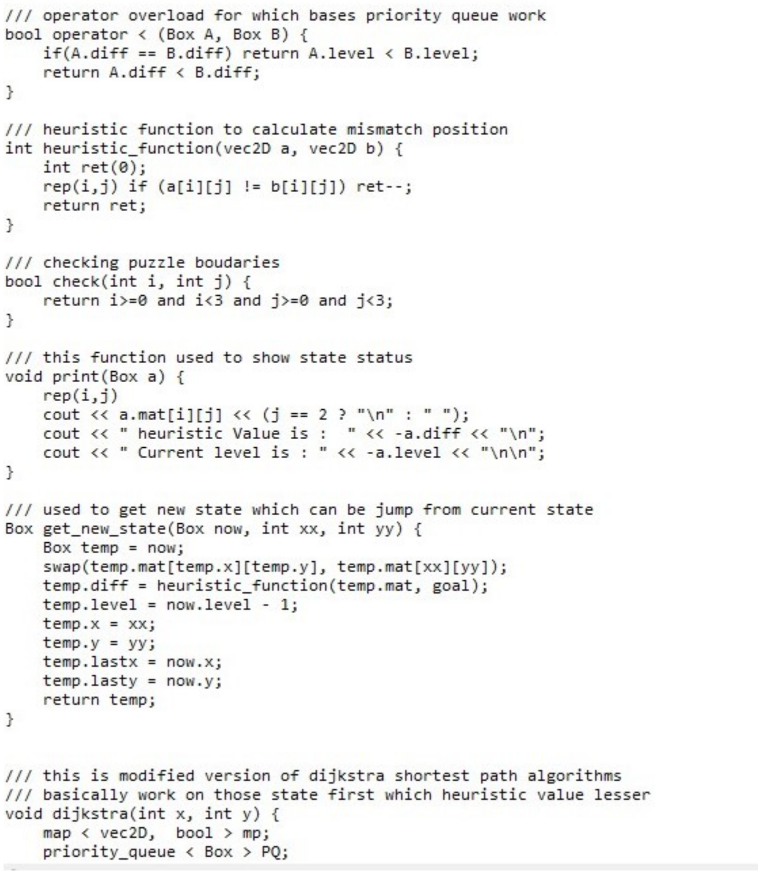
Code 2 : The BFS algorithm is useful for analyzing the nodes in a graph and constructing the shortest path of traversing through these.



```
#include<bits/stdc++.h>
using namespace std;
#define D(x) cerr<<__LINE__<<" : "<<#x<<" -> "<<x<<endl
#define rep(i,j) for(int i = 0; i < 3; i++) for(int j = 0; j < 3; j++)
#define PII pair < int, int >
typedef vector<vector<int>> vec2D;

const int MAX = 1e5+7;
int t=1, n, m, l, k, tc;

int dx[4] = {0, 0, 1, -1};
int dy[4] = {1, -1, 0, 0};

vec2D init{
    {8, 1, 2},
    {3, 6, 4},
    {0, 7, 5}
};
vec2D goal{
    {1, 3, 2},
    {8, 0, 4},
    {7, 6, 5}
};
/// using a structure to store information of each state
struct Box {
    vec2D mat{ { 0,0,0 },{ 0,0,0},{ 0,0,0} };
    int diff, level;
    int x, y;
    int lastx, lasty;
    Box(vec2D a,int b = 0, int c = 0, PII p = {0,0}, PII q = {0,0}) {
        rep(i,j) mat[i][j] = a[i][j];
        diff = b;
        level = c;
        x = p.first;
        y = p.second;
        lastx = q.first;
        lasty = q.second;
    }
};

/// operator overload for which bases priority queue work
bool operator < (Box A, Box B) {
    if(A.diff == B.diff) return A.level < B.level;
```

a) fig :1

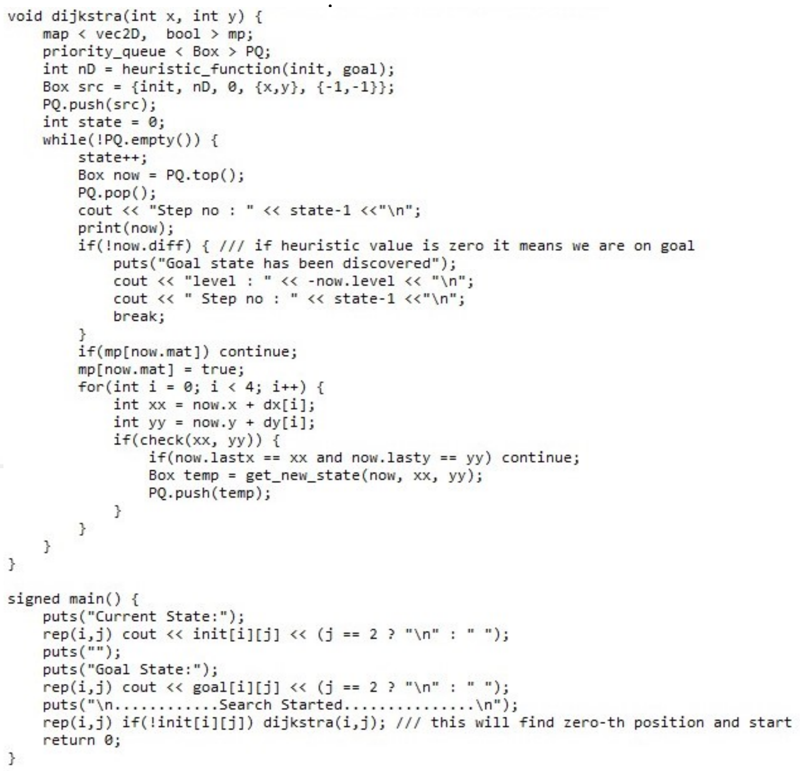
:



```
/// operator overload for which bases priority queue work
bool operator < (Box A, Box B) {
    if(A.diff == B.diff) return A.level < B.level;
    return A.diff < B.diff;
}

/// heuristic function to calculate mismatch position
int heuristic_function(vec2D a, vec2D b) {
    int ret(0);
    rep(i,j) if (a[i][j] != b[i][j]) ret--;
    return ret;
}

/// checking puzzle boudaries
bool check(int i, int j) {
    return i>=0 and i<3 and j>=0 and j<3;
}

/// this function used to show state status
void print(Box a) {
    rep(i,j)
    cout << a.mat[i][j] << (j == 2 ? "\n" : " ");
    cout << " heuristic Value is :  " << -a.diff << "\n";
    cout << " Current level is : " << -a.level << "\n\n";
}

/// used to get new state which can be jump from current state
Box get_new_state(Box now, int xx, int yy) {
    Box temp = now;
    swap(temp.mat[temp.x][temp.y], temp.mat[xx][yy]);
    temp.diff = heuristic_function(temp.mat, goal);
    temp.level = now.level - 1;
    temp.x = xx;
    temp.y = yy;
    temp.lastx = now.x;
    temp.lasty = now.y;
    return temp;
}


/// this is modified version of dijkstra shortest path algorithms
/// basically work on those state first which heuristic value lesser
void dijkstra(int x, int y) {
    map < vec2D,  bool > mp;
    priority_queue < Box > PQ;
```

b) fig :1.2

:



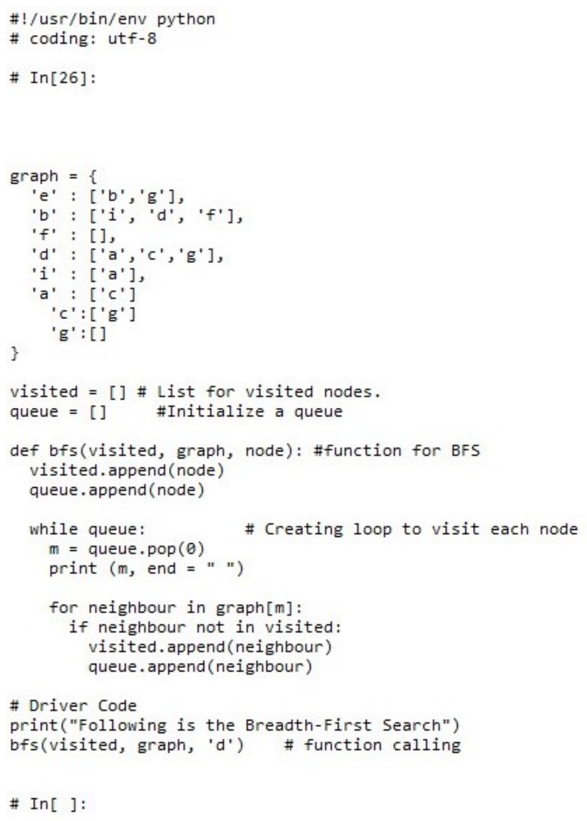
```
void dijkstra(int x, int y) {
    map < vec2D,  bool > mp;
    priority_queue < Box > PQ;
    int nD = heuristic_function(init, goal);
    Box src = {init, nD, 0, {x,y}, {-1,-1}};
    PQ.push(src);
    int state = 0;
    while(!PQ.empty()) {
        state++;
        Box now = PQ.top();
        PQ.pop();
        cout << "Step no : " << state-1 <<"\n";
        print(now);
        if(!now.diff) { /// if heuristic value is zero it means we are on goal
            puts("Goal state has been discovered");
            cout << "level : " << -now.level << "\n";
            cout << " Step no : " << state-1 <<"\n";
            break;
        }
        if(mp[now.mat]) continue;
        mp[now.mat] = true;
        for(int i = 0; i < 4; i++) {
            int xx = now.x + dx[i];
            int yy = now.y + dy[i];
            if(check(xx, yy)) {
                if(now.lastx == xx and now.lasty == yy) continue;
                Box temp = get_new_state(now, xx, yy);
                PQ.push(temp);
            }
        }
    }
}

signed main() {
    puts("Current State:");
    rep(i,j) cout << init[i][j] << (j == 2 ? "\n" : " ");
    puts("");
    puts("Goal State:");
    rep(i,j) cout << goal[i][j] << (j == 2 ? "\n" : " ");
    puts("\n............Search Started...............\n");
    rep(i,j) if(!init[i][j]) dijkstra(i,j); /// this will find zero-th position and start
    return 0;
}
```

c) fig :1.3

:

ACKNOWLEDGMENT

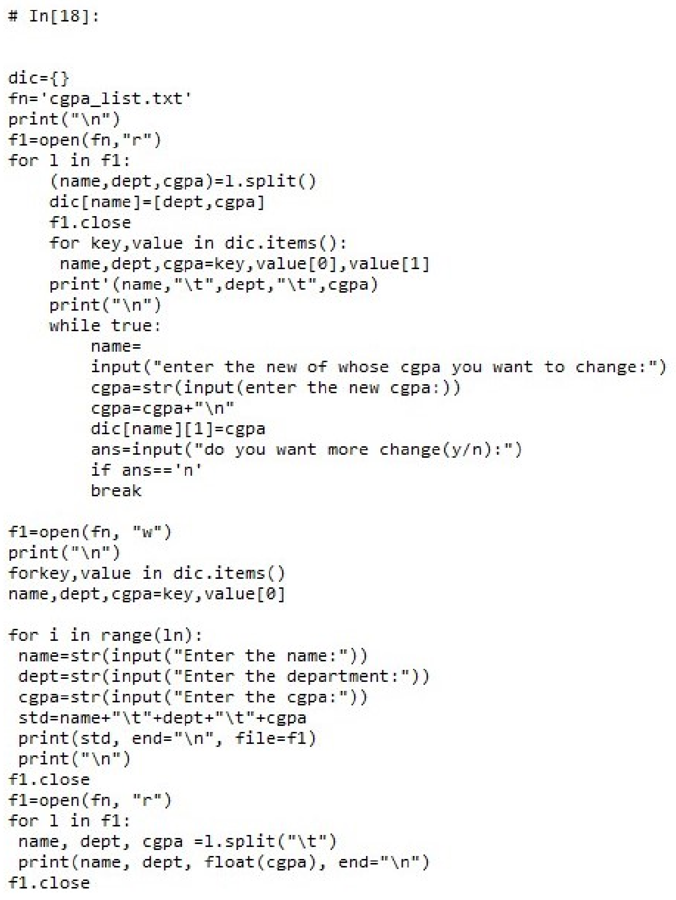
I would like to thank my honourable **Khan Md. Hasib Sir** for his time, generosity and critical insights into this project.



```
#!/usr/bin/env python
# coding: utf-8

# In[26]:



graph = {
  'e' : ['b','g'],
  'b' : ['i', 'd', 'f'],
  'f' : [],
  'd' : ['a','c','g'],
  'i' : ['a'],
  'a' : ['c']
    'c':['g']
    'g':[]
}

visited = [] # List for visited nodes.
queue = []     #Initialize a queue

def bfs(visited, graph, node): #function for BFS
  visited.append(node)
  queue.append(node)

  while queue:          # Creating loop to visit each node
    m = queue.pop(0)
    print (m, end = " ")

    for neighbour in graph[m]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)

# Driver Code
print("Following is the Breadth-First Search")
bfs(visited, graph, 'd')    # function calling


# In[ ]:
```

d) fig :2.0

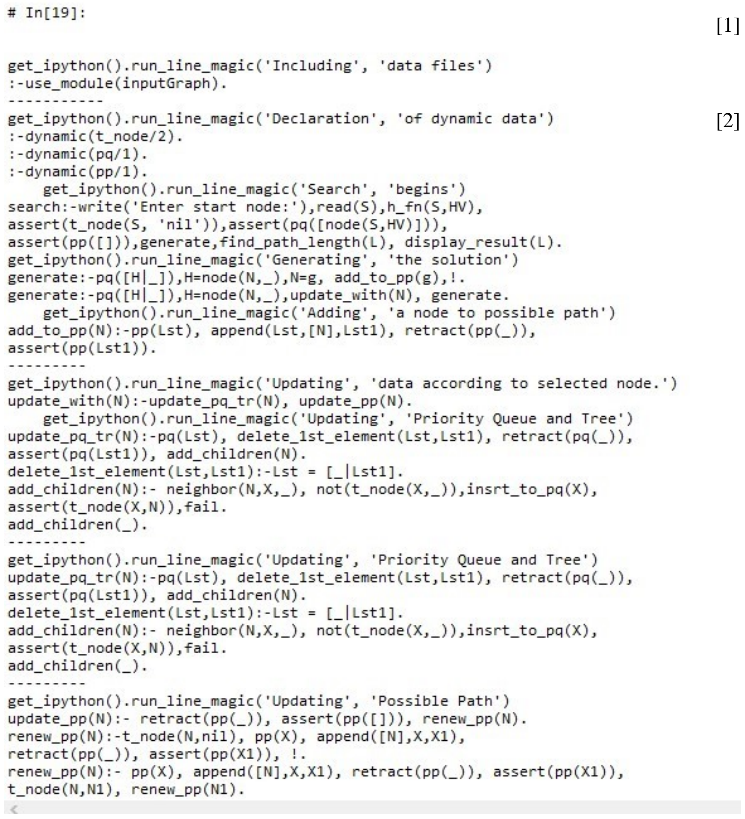
:



```
# In[18]:


dic={}
fn='cgpa_list.txt'
print("\n")
f1=open(fn,"r")
for l in f1:
    (name,dept,cgpa)=l.split()
    dic[name]=[dept,cgpa]
    f1.close
    for key,value in dic.items():
     name,dept,cgpa=key,value[0],value[1]
    print'(name,"\t",dept,"\t",cgpa)
    print("\n")
    while true:
        name=
        input("enter the new of whose cgpa you want to change:")
        cgpa=str(input(enter the new cgpa:))
        cgpa=cgpa+"\n"
        dic[name][1]=cgpa
        ans=input("do you want more change(y/n):")
        if ans=='n'
        break

f1=open(fn, "w")
print("\n")
forkey,value in dic.items()
name,dept,cgpa=key,value[0]

for i in range(ln):
 name=str(input("Enter the name:"))
 dept=str(input("Enter the department:"))
 cgpa=str(input("Enter the cgpa:"))
 std=name+"\t"+dept+"\t"+cgpa
 print(std, end="\n", file=f1)
 print("\n")
f1.close
f1=open(fn, "r")
for l in f1:
 name, dept, cgpa =l.split("\t")
 print(name, dept, float(cgpa), end="\n")
f1.close
```

e) fig :2.1

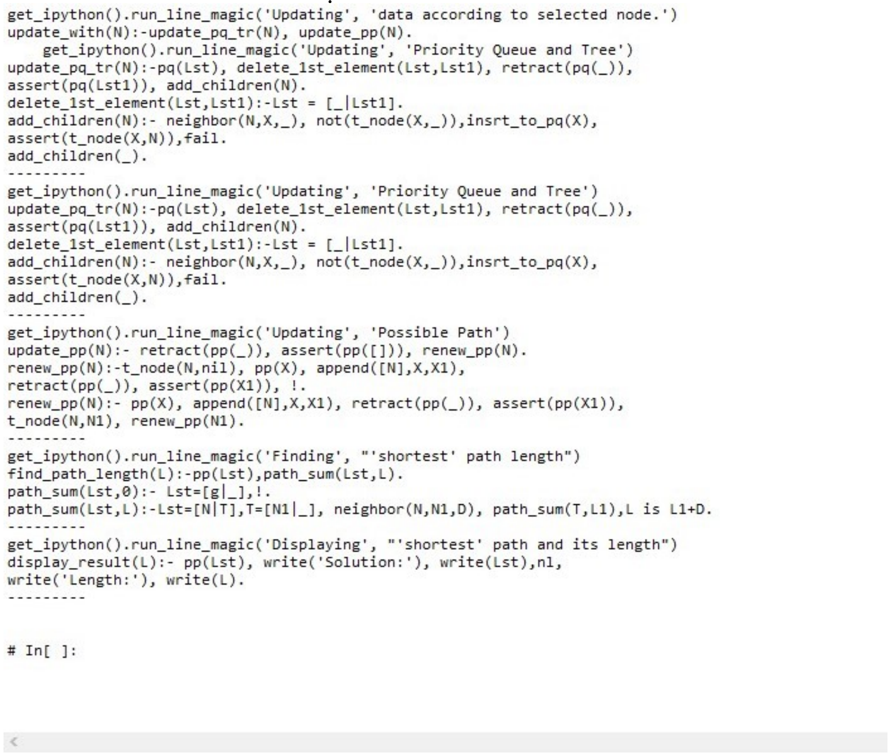
:



```
# In[19]:


get_ipython().run_line_magic('Including', 'data files')
:-use_module(inputGraph).
-----------
get_ipython().run_line_magic('Declaration', 'of dynamic data')
:-dynamic(t_node/2).
:-dynamic(pq/1).
:-dynamic(pp/1).
    get_ipython().run_line_magic('Search', 'begins')
search:-write('Enter start node:'),read(S),h_fn(S,HV),
assert(t_node(S, 'nil')),assert(pq([node(S,HV)])),
assert(pp([])),generate,find_path_length(L), display_result(L).
get_ipython().run_line_magic('Generating', 'the solution')
generate:-pq([H|_]),H=node(N,_),N=g, add_to_pp(g),!.
generate:-pq([H|_]),H=node(N,_),update_with(N), generate.
    get_ipython().run_line_magic('Adding', 'a node to possible path')
add_to_pp(N):-pp(Lst), append(Lst,[N],Lst1), retract(pp(_)),
assert(pp(Lst1)).
---------
get_ipython().run_line_magic('Updating', 'data according to selected node.')
update_with(N):-update_pq_tr(N), update_pp(N).
    get_ipython().run_line_magic('Updating', 'Priority Queue and Tree')
update_pq_tr(N):-pq(Lst), delete_1st_element(Lst,Lst1), retract(pq(_)),
assert(pq(Lst1)), add_children(N).
delete_1st_element(Lst,Lst1):-Lst = [_|Lst1].
add_children(N):- neighbor(N,X,_), not(t_node(X,_)),insrt_to_pq(X),
assert(t_node(X,N)),fail.
add_children(_).
---------
get_ipython().run_line_magic('Updating', 'Priority Queue and Tree')
update_pq_tr(N):-pq(Lst), delete_1st_element(Lst,Lst1), retract(pq(_)),
assert(pq(Lst1)), add_children(N).
delete_1st_element(Lst,Lst1):-Lst = [_|Lst1].
add_children(N):- neighbor(N,X,_), not(t_node(X,_)),insrt_to_pq(X),
assert(t_node(X,N)),fail.
add_children(_).
---------
get_ipython().run_line_magic('Updating', 'Possible Path')
update_pp(N):- retract(pp(_)), assert(pp([])), renew_pp(N).
renew_pp(N):-t_node(N,nil), pp(X), append([N],X,X1),
retract(pp(_)), assert(pp(X1)), !.
renew_pp(N):- pp(X), append([N],X,X1), retract(pp(_)), assert(pp(X1)),
t_node(N,N1), renew_pp(N1).
```

f) fig :2.2

:



```
get_ipython().run_line_magic('Updating', 'data according to selected node.')
update_with(N):-update_pq_tr(N), update_pp(N).
    get_ipython().run_line_magic('Updating', 'Priority Queue and Tree')
update_pq_tr(N):-pq(Lst), delete_1st_element(Lst,Lst1), retract(pq(_)),
assert(pq(Lst1)), add_children(N).
delete_1st_element(Lst,Lst1):-Lst = [_|Lst1].
add_children(N):- neighbor(N,X,_), not(t_node(X,_)),insrt_to_pq(X),
assert(t_node(X,N)),fail.
add_children(_).
---------
get_ipython().run_line_magic('Updating', 'Priority Queue and Tree')
update_pq_tr(N):-pq(Lst), delete_1st_element(Lst,Lst1), retract(pq(_)),
assert(pq(Lst1)), add_children(N).
delete_1st_element(Lst,Lst1):-Lst = [_|Lst1].
add_children(N):- neighbor(N,X,_), not(t_node(X,_)),insrt_to_pq(X),
assert(t_node(X,N)),fail.
add_children(_).
---------
get_ipython().run_line_magic('Updating', 'Possible Path')
update_pp(N):- retract(pp(_)), assert(pp([])), renew_pp(N).
renew_pp(N):-t_node(N,nil), pp(X), append([N],X,X1),
retract(pp(_)), assert(pp(X1)), !.
renew_pp(N):- pp(X), append([N],X,X1), retract(pp(_)), assert(pp(X1)),
t_node(N,N1), renew_pp(N1).
---------
get_ipython().run_line_magic('Finding', "'shortest' path length")
find_path_length(L):-pp(Lst),path_sum(Lst,L).
path_sum(Lst,0):- Lst=[g|_],!.
path_sum(Lst,L):-Lst=[N|T],T=[N1|_], neighbor(N,N1,D), path_sum(T,L1),L is L1+D.
---------
get_ipython().run_line_magic('Displaying', "'shortest' path and its length")
display_result(L):- pp(Lst), write('Solution:'), write(Lst),nl,
write('Length:'), write(L).
---------


# In[ ]:
```

g) fig :2.3

:

REFERENCES

[1] Code 1 :
] Piltaver, R., Lustrek, M., and Gams, M. (2012). The pathology of heuristic ˘ search in the 8-puzzle. Journal of Experimental and Theoretical Artificial Intelligence, 24(1), 65-94

[2] Code 2 : Daniel Carlos Guimarães Pedronette received a B.Sc. in computer science (2005) from the State University of São Paulo (Brazil) and the M.Sc. degree in computer science (2008) from the University of Campinas (Brazil). He got his doctorate in computer science at the same university in 2012.