Assignment No: 01

CSE-0408 Summer 2021

Joy Sarkar
Department of Computer Science and Engineering
State University of Bangladesh (SUB)
Dhaka, Bangladesh
joysarker39@gmail.com

***Abstract*—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.**

***Index Terms*—heuristic, puzzle**

I. Introduction

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

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.

III. Proposed Methodology

IV. Conclusion

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.

Acknowledgment

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

References

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

```cpp
#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;
    return A.diff < B.diff;
}

```

Fig. 1. Proposed Methodology

```cpp

/// 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;
}

```

Fig. 2. Proposed Methodology

```
/// 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;
    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 positio
    return 0;
}
```

Fig. 3. Proposed Methodology

```
Current State:
8 1 2
3 6 4
0 7 5

Goal State:
1 3 2
8 0 4
7 6 5

.............Search Started...............

Step no : 0
8 1 2
3 6 4
0 7 5
 heuristic Value is :  6
 Current level is : 0

Step no : 1
8 1 2
3 6 4
7 0 5
 heuristic Value is :  5
 Current level is : 1

Step no : 2
8 1 2
3 0 4
7 6 5
 heuristic Value is :  3
 Current level is : 2

Step no : 3
8 1 2
0 3 4
7 6 5
 heuristic Value is :  4
 Current level is : 3

Step no : 4
0 1 2
8 3 4
7 6 5
 heuristic Value is :  3
 Current level is : 4

Step no : 5
1 0 2
8 3 4
7 6 5
 heuristic Value is :  2
 Current level is : 5

Step no : 6
1 3 2
8 0 4
7 6 5
 heuristic Value is :  0
 Current level is : 6

Goal state has been discovered
level : 6
 Step no : 6

Process returned 0 (0x0)   execution time : 0.003 s
Press ENTER to continue.
```

Assignment No: 02
BFS Algorithms

CSE-0408 Summer 2021

Joy Sarkar
Department of Computer Science and Engineering
State University of Bangladesh (SUB)
Dhaka, Bangladesh
joysarker39@gmail.com

***Abstract*—Breadth-first search (BFS) is an algorithm for searching a tree data structure for a node that satisfies a given property. It starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level. Extra memory, usually a queue, is needed to keep track of the child nodes that were encountered but not yet explored. This assignment is basically implementation of a BFS algorithms.**

***Index Terms*—BFS, graph, Networkx, Matplotlib**

I. Introduction

Graph traversal means visiting every vertex and edge exactly once in a well-defined order. While using certain graph algorithms, you must ensure that each vertex of the graph is visited exactly once. The order in which the vertices are visited are important and may depend upon the algorithm or question that you are solving.

During a traversal, it is important that you track which vertices have been visited. The most common way of tracking vertices is to mark them.

Breadth First Search (BFS)

There are many ways to traverse graphs. BFS is the most commonly used approach.

BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.

II. Literature Review

Konrad Zuse devised BFS and its use in finding related components of graphs in his (rejected) Ph.D. thesis on the Plankalkül programming language in 1945, but it wasn't published until 1972. Edward F. Moore recreated it in 1959 to determine the shortest path out of a maze, and C. Y. Lee later extended it into a wire routing method (published 1961).

III. Proposed Methodology

Input: A graph G and a starting vertex root of G
Output: Goal state. The parent links trace the shortest path back to root

```
procedure BFS(G, root) is
 let Q be a queue
    label root as explored
    Q.enqueue(root)
    while Q is not empty do
       v := Q.dequeue()
       if v is the goal then
          return v
       for all edges from v to w in G.adjacentEdges(v) do
            if w is not labeled as explored then
               label w as explored
               Q.enqueue(w)
```

We are using **NetworkX** for creating Graph. NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks(Graph).
Also using **matplotlib.pyplot**, we can graphically represent our graph.

IV. Conclusion

This assignment is based on graphically representation on python of BFS algorithms. Time Complexity: O(V+E) where V is number of vertices in the graph and E is number of edges in the graph.

Acknowledgment

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

References

[1] Cormen Thomas H.; et al. (2009). "22.3". Introduction to Algorithms. MIT Press.

BFS

August 8, 2021

[1]:
```
import networkx as nx
import matplotlib.pyplot as plt
from collections import deque
import random
```

[2]:
```
def CreateGraph(node, edge):
    G = nx.Graph()
    for i in range(1, node+1):
        G.add_node(i)
    for i in range(edge):
        u, v = random.randint(1, node), random.randint(1, node)
        G.add_edge(u, v)
    return G
```

[3]:
```
def DrawGraph(G, color):
    pos = nx.spring_layout(G)
    nx.draw(G, pos, with_labels = True, node_color = color, edge_color = 'black'␣
 ↪,width = 1, alpha = 0.7)  #with_labels=true is to show the node number in the␣
 ↪output graph
```

[4]:
```
def DrawIteratedGraph(G,col_val):
    pos = nx.spring_layout(G)
    color = ["green", "blue", "yellow", "pink", "red", "black", "gray", "brown",␣
 ↪"orange", "plum"]
    values = []
    for node in G.nodes():
        values.append(color[col_val[node]])
    nx.draw(G, pos, with_labels = True, node_color = values, edge_color =␣
 ↪'black' ,width = 1, alpha = 0.7)  #with_labels=true is to show the node number␣
 ↪in the output graph
```

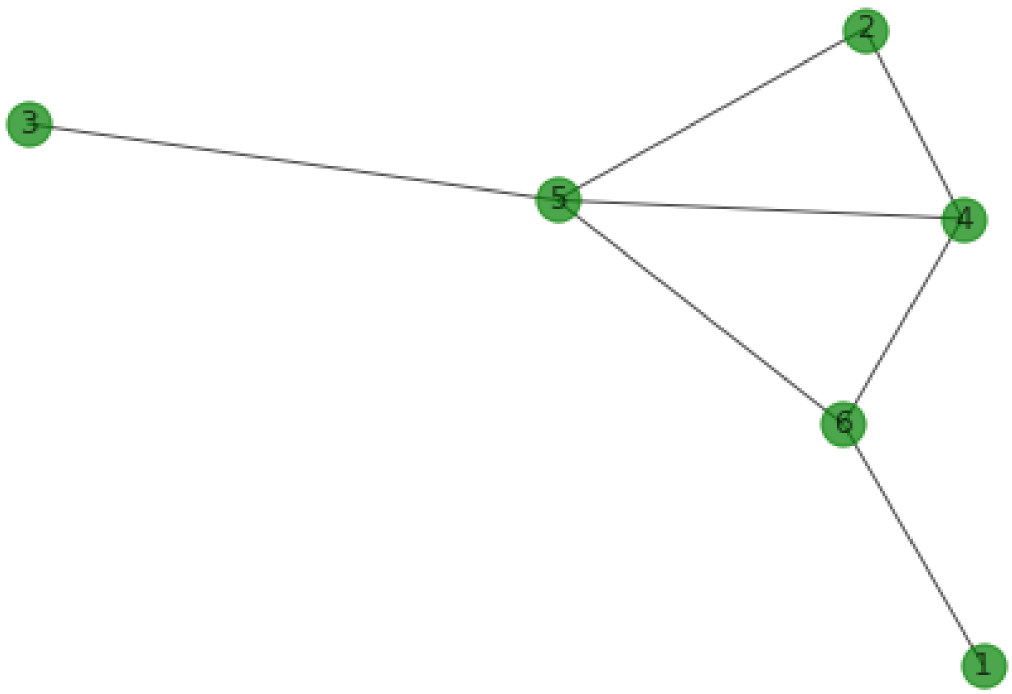
[5]:
```
def DrawSolutionGraph(G,col_val):
    pos = nx.spring_layout(G)
    values = []
    for node in G.nodes():
        values.append(col_val.get(node, col_val.get(node)))
```

```
    nx.draw(G, pos, with_labels = True, node_color = values, edge_color =␣
↪'black' ,width = 1, alpha = 0.7)  #with_labels=true is to show the node number␣
↪in the output graph
```

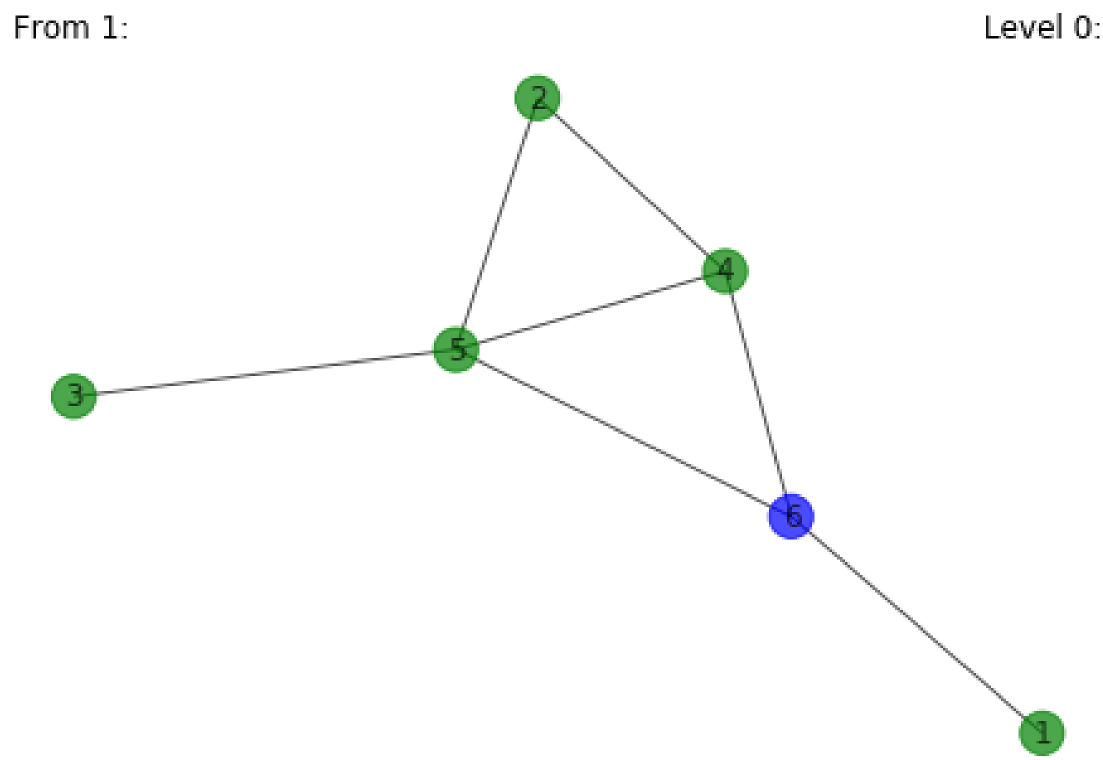
[6]:
```
def BFS(start):
    queue = deque()
    queue.append(start)
    visited[start] = True
    level[start] = 0

    while queue:
        u = queue.popleft()
        print(u, " -> ", end = "")
        for v in G.adj[u]:
            if not visited[v]:
                queue.append(v)
                visited[v] = True
                level[v] = level[u] + 1

        DrawIteratedGraph(G, level)
        plt.title('From {}:'.format(u), loc='left')
        plt.title('Level {}:'.format(level[u]), loc='right')
        plt.show()

    print("End")
```

[7]:
```
if __name__ == "__main__":

    node, edge = 6, 10

    G = CreateGraph(node, edge)
    print("Nodes: ", G.nodes)
    DrawGraph(G, "green")
    plt.show()
    visited = [False for i in range(node+1)]
    level = [0 for i in range(node+1)]
    parent = [0 for i in range(node+1)]
    root = 1
    BFS(root)
```

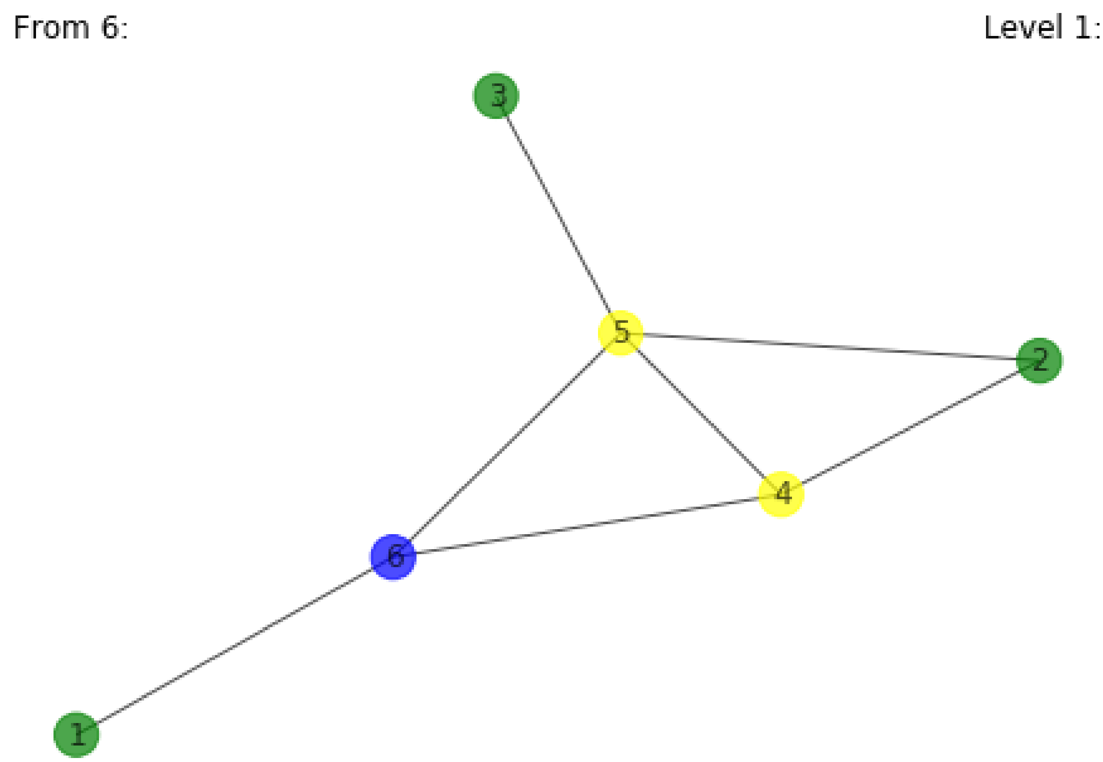
```
Nodes:  [1, 2, 3, 4, 5, 6]
```



1 ->



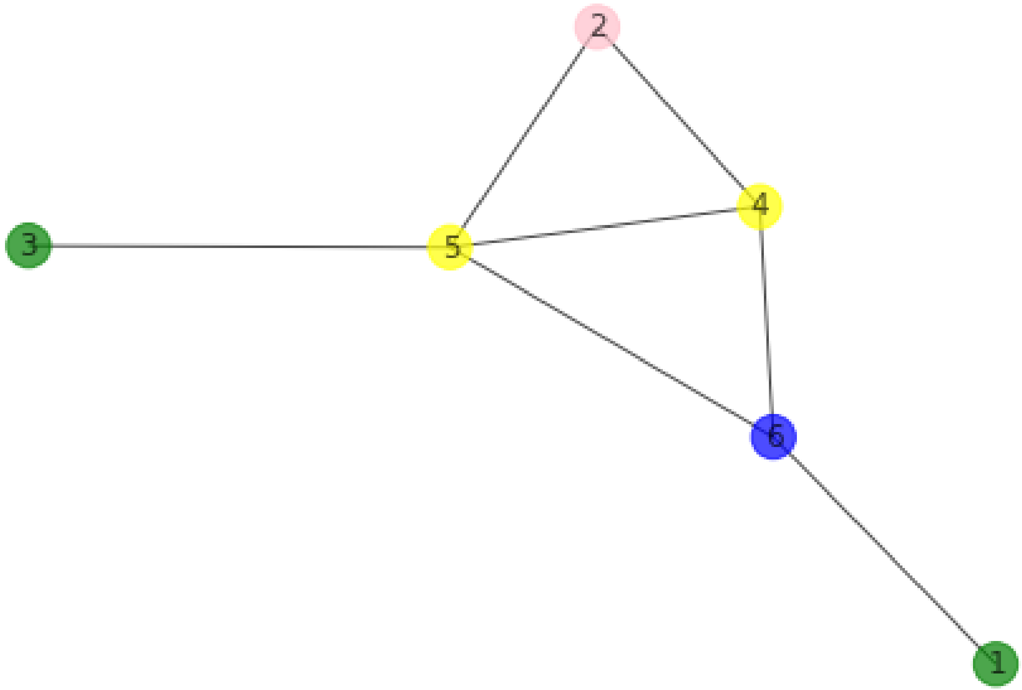
6 ->



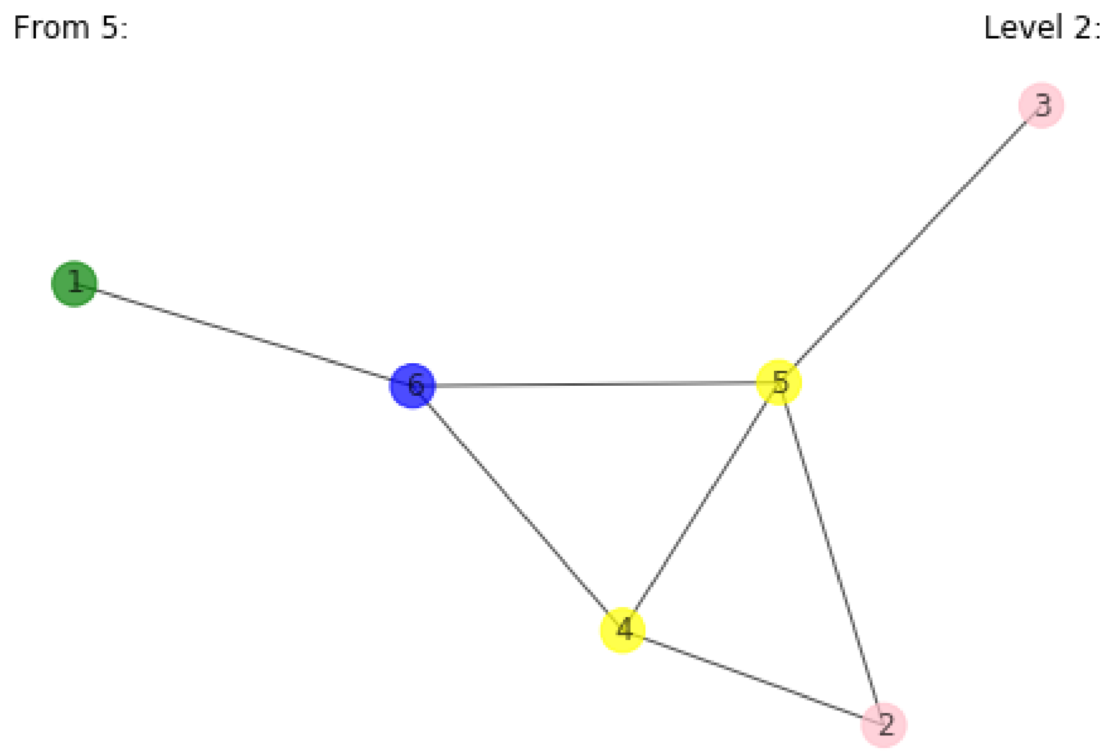
4 ->

From 4:

Level 2:

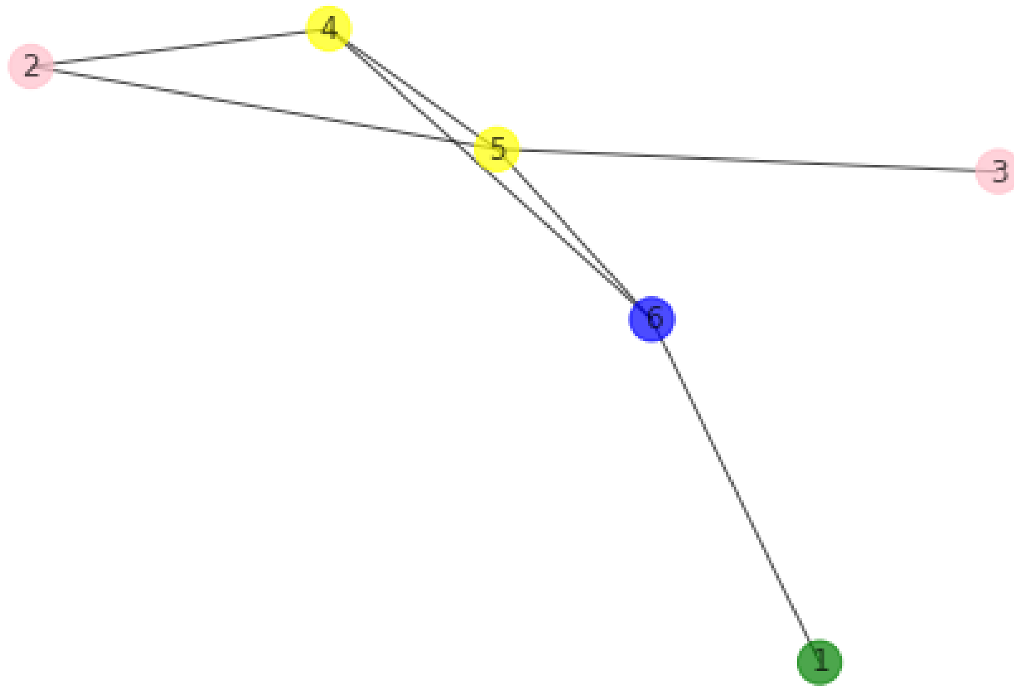


5 ->

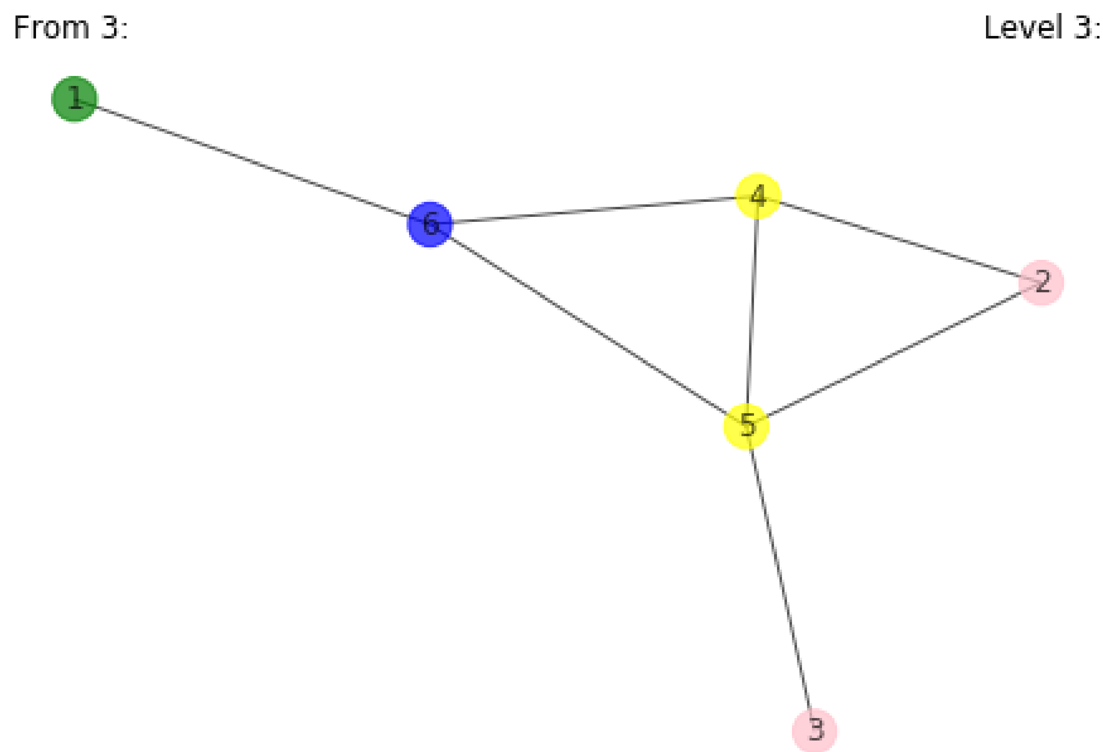


2 ->

From 2: Level 3:



3 ->



End

```
[ ]:
```