Artificial Intelligence

CSE-0408 Summer 2021

Md Mahthiul Hasan Thanvi
UG02-47-18-025
Department of Computer Science and Engineering
State University of Bangladesh (SUB)
Dhaka, Bangladesh
thanvimdmhatheulhasan@gmail.com

***Abstract*—- The eight puzzle problem is the largest completely solvable problem of n×n sliding puzzle problems. Using Breadth First search (BFS) graph traversal we can reach the solution for a definite goal. The parallel algorithm is capable of providing us with much more faster solution.**

***Index Terms*—- Eight puzzle problem, Complete Solution, Breadthfirst searchs**

I. Introduction

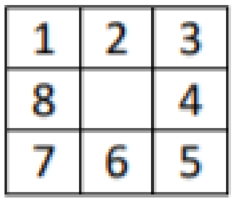
The eight puzzle problem serves as the workbench model to measure the performances of state space search algorithms. The puzzle requires rearranging a 3×3 board of square tiles into a specific order. We examined all the possible permutations and their solvability. We have found an optimized solution for all the 9!/2 permutations. Till now the eight puzzle problem is the largest possible N puzzle which can be completely solved. In a 3×3 board, there exist 9! possible permutations. Out of these 9! permutations every second one is able to reach the goal state [1]. It leaves us with a total of 9! /2=181440 problem instances which are solvable.

II. Literature Review

Piltaver, Rok, Mitja Luštrek, and Matjaž Gams. ”The pathology of heuristic search in the 8-puzzle.” Journal of Experimental Theoretical Artificial Intelligence 24.1 (2012): 65-94.

III. THE EIGHT PUZZLE PROBLEM

The objective of eight puzzle problem is to rearrange a given initial configuration of squared tiles in a 3x3 board into a specific goal configuration. This can be achieved only by successive sliding of tiles into orthogonally adjacent empty squares. The main interest lies in finding the optimal solutions with the minimum number of moves. Conventionally the following configuration is taken as the goal configuration:



1	2	3
8		4
7	6	5

On a 3x3 board there exist 9! Permutations and every second puzzle are solvable [Johnson and Storey, 1879]. This leaves total 9! /2 = 181440 solvable problem instances.

IV. Conclusion

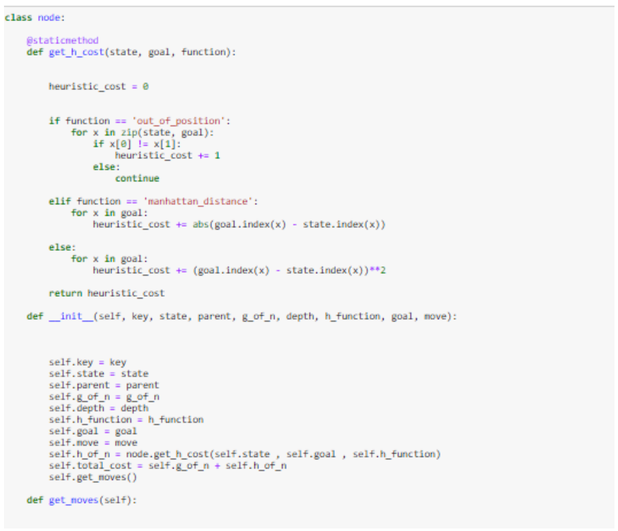
The eight puzzle problem is solved using BFS. Using BFS the goal state can be reached in minimum number of moves. To find a solution of this problem, the initial configuration is marked as the source vertex. All the reachable state from the source state/ vertex is marked as first level vertices. In a 3x3 board, there can be at most four reachable states from a single state. In sequential method the vertices in the same level are checked one after another. If the goal state is not found, the next levels of vertices are explored. This searching continues until the goal state is reached. In parallel method, all the vertices from same level are processed in parallel.

Acknowledgment

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

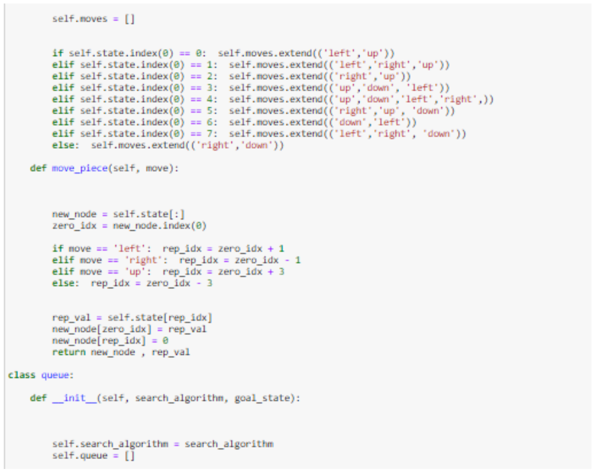
References

[1] W.W. Johnson and W.E. Storey. “Notes on the ‘15’-Puzzle.” Amer. J. Math. 2(1879), 397-404.
[2] P.D.A. Schofeld, N.L. Collins, D. Michie (eds.), “Complete solution of the Eight-Puzzle”Machine Intell. 1, Amer. Elsevier, NY (1967), 125-133.
[3] A. Reinefeld and T.A. Marsland. “Memory functions in iterativedeepening search.” Univ. Hamburg, FB Informatik, Tech. Rep. FBIHH-M-198/91.
[4] Duane Merrill, Michael Garland, Andrew Grimshaw. “Scalable GPU Graph Traversal” PPoPP’12, February 25–29, 2012, New Orleans, Louisiana, USA, 2012 ACM 978-1-4503-1160-1/12/02
[5] Korf, R. E., and Schultze, “Large-scale parallel breadth-first search.”,National Conference on Artificial Intelligence (AAAI), 1380–1385, 2005



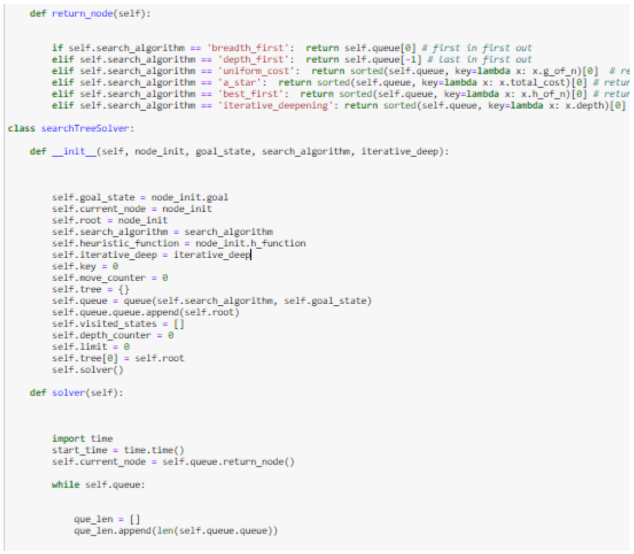
```
class node:

    @staticmethod
    def get_h_cost(state, goal, function):


        heuristic_cost = 0


        if function == 'out_of_position':
            for x in zip(state, goal):
                if x[0] != x[1]:
                    heuristic_cost += 1
                else:
                    continue

        elif function == 'manhattan_distance':
            for x in goal:
                heuristic_cost += abs(goal.index(x) - state.index(x))

        else:
            for x in goal:
                heuristic_cost += (goal.index(x) - state.index(x))**2

        return heuristic_cost

    def __init__(self, key, state, parent, g_of_n, depth, h_function, goal, move):



        self.key = key
        self.state = state
        self.parent = parent
        self.g_of_n = g_of_n
        self.depth = depth
        self.h_function = h_function
        self.goal = goal
        self.move = move
        self.h_of_n = node.get_h_cost(self.state , self.goal , self.h_function)
        self.total_cost = self.g_of_n + self.h_of_n
        self.get_moves()

    def get_moves(self):
```

Fig. 2. Proposed Methodology



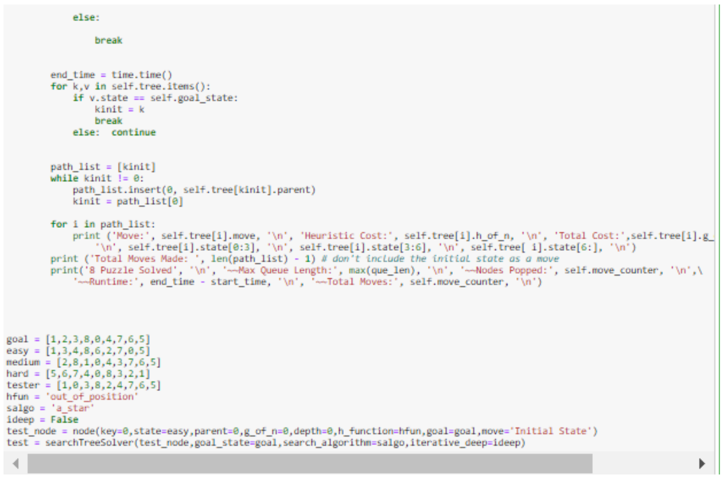
```
        self.moves = []


        if self.state.index(0) == 0:  self.moves.extend(('left','up'))
        elif self.state.index(0) == 1:  self.moves.extend(('left','right','up'))
        elif self.state.index(0) == 2:  self.moves.extend(('right','up'))
        elif self.state.index(0) == 3:  self.moves.extend(('up','down', 'left'))
        elif self.state.index(0) == 4:  self.moves.extend(('up','down','left','right',))
        elif self.state.index(0) == 5:  self.moves.extend(('right','up', 'down'))
        elif self.state.index(0) == 6:  self.moves.extend(('down','left'))
        elif self.state.index(0) == 7:  self.moves.extend(('left','right', 'down'))
        else:  self.moves.extend(('right','down'))

    def move_piece(self, move):



        new_node = self.state[:]
        zero_idx = new_node.index(0)

        if move == 'left':  rep_idx = zero_idx + 1
        elif move == 'right':  rep_idx = zero_idx - 1
        elif move == 'up':  rep_idx = zero_idx + 3
        else:  rep_idx = zero_idx - 3


        rep_val = self.state[rep_idx]
        new_node[zero_idx] = rep_val
        new_node[rep_idx] = 0
        return new_node , rep_val

class queue:

    def __init__(self, search_algorithm, goal_state):



        self.search_algorithm = search_algorithm
        self.queue = []
```

Fig. 3. Proposed Methodology



```
    def return_node(self):


        if self.search_algorithm == 'breadth_first':  return self.queue[0] # first in first out
        elif self.search_algorithm == 'depth_first':  return self.queue[-1] # last in first out
        elif self.search_algorithm == 'uniform_cost':  return sorted(self.queue, key=lambda x: x.g_of_n)[0]  # re
        elif self.search_algorithm == 'a_star':  return sorted(self.queue, key=lambda x: x.total_cost)[0] # retur
        elif self.search_algorithm == 'best_first':  return sorted(self.queue, key=lambda x: x.h_of_n)[0] # retur
        elif self.search_algorithm == 'iterative_deepening': return sorted(self.queue, key=lambda x: x.depth)[0]

class searchTreeSolver:

    def __init__(self, node_init, goal_state, search_algorithm, iterative_deep):



        self.goal_state = node_init.goal
        self.current_node = node_init
        self.root = node_init
        self.search_algorithm = search_algorithm
        self.heuristic_function = node_init.h_function
        self.iterative_deep = iterative_deep
        self.key = 0
        self.move_counter = 0
        self.tree = {}
        self.queue = queue(self.search_algorithm, self.goal_state)
        self.queue.queue.append(self.root)
        self.visited_states = []
        self.depth_counter = 0
        self.limit = 0
        self.tree[0] = self.root
        self.solver()

    def solver(self):



        import time
        start_time = time.time()
        self.current_node = self.queue.return_node()

        while self.queue:



            que_len = []
            que_len.append(len(self.queue.queue))
```

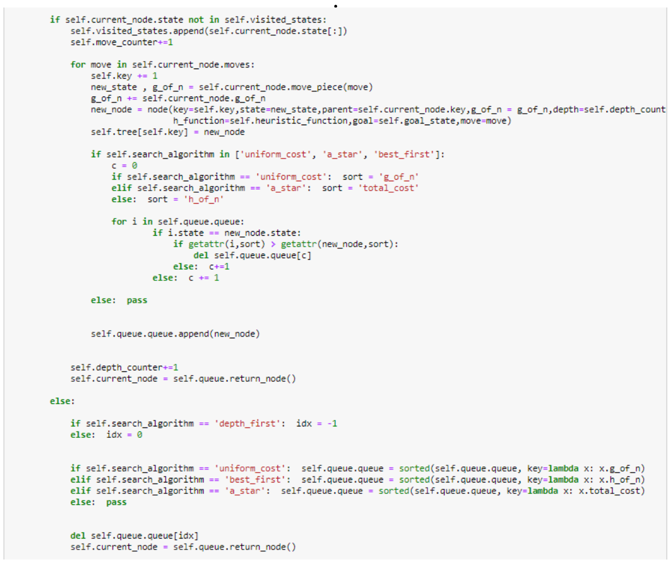
Fig. 4. Proposed Methodology



```
                else:

                    break


            end_time = time.time()
            for k,v in self.tree.items():
                if v.state == self.goal_state:
                    kinit = k
                    break
                else:  continue


            path_list = [kinit]
            while kinit != 0:
                path_list.insert(0, self.tree[kinit].parent)
                kinit = path_list[0]

            for i in path_list:
                print ('Move:', self.tree[i].move, '\n', 'Heuristic Cost:', self.tree[i].h_of_n, '\n', 'Total Cost:',self.tree[i].g_
                    '\n', self.tree[i].state[0:3], '\n', self.tree[i].state[3:6], '\n', self.tree[ i].state[6:], '\n')
            print ('Total Moves Made: ', len(path_list) - 1) # don't include the initial state as a move
            print('8 Puzzle Solved', '\n', '~~Max Queue Length:', max(que_len), '\n', '~~Nodes Popped:', self.move_counter, '\n',\
                '~~Runtime:', end_time - start_time, '\n', '~~Total Moves:', self.move_counter, '\n')



goal = [1,2,3,8,0,4,7,6,5]
easy = [1,3,4,8,6,2,7,0,5]
medium = [2,8,1,0,4,3,7,6,5]
hard = [5,6,7,4,0,8,3,2,1]
tester = [1,0,3,8,2,4,7,6,5]
hfun = 'out_of_position'
salgo = 'a_star'
ideep = False
test_node = node(key=0,state=easy,parent=0,g_of_n=0,depth=0,h_function=hfun,goal=goal,move='Initial State')
test = searchTreeSolver(test_node,goal_state=goal,search_algorithm=salgo,iterative_deep=ideep)
```

Fig. 5. Proposed Methodology

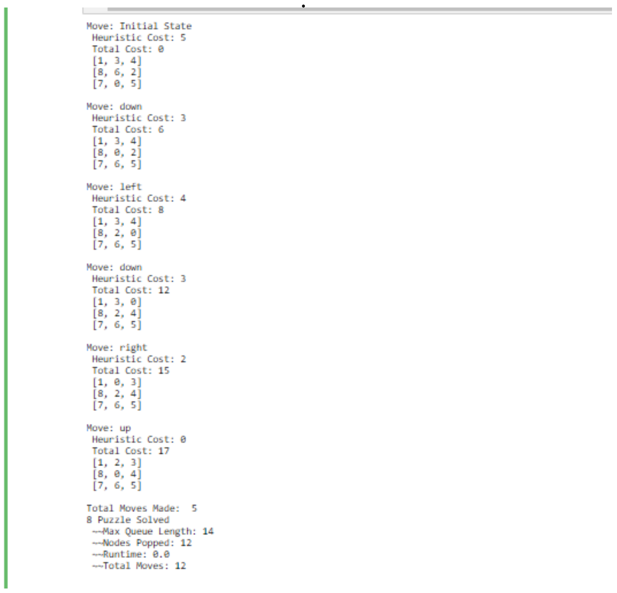
.



```
        if self.current_node.state not in self.visited_states:
            self.visited_states.append(self.current_node.state[:])
            self.move_counter+=1

            for move in self.current_node.moves:
                self.key += 1
                new_state , g_of_n = self.current_node.move_piece(move)
                g_of_n += self.current_node.g_of_n
                new_node = node(key=self.key,state=new_state,parent=self.current_node.key,g_of_n = g_of_n,depth=self.depth_count
                                h_function=self.heuristic_function,goal=self.goal_state,move=move)
                self.tree[self.key] = new_node

                if self.search_algorithm in ['uniform_cost', 'a_star', 'best_first']:
                    c = 0
                    if self.search_algorithm == 'uniform_cost':  sort = 'g_of_n'
                    elif self.search_algorithm == 'a_star':  sort = 'total_cost'
                    else:  sort = 'h_of_n'

                    for i in self.queue.queue:
                            if i.state == new_node.state:
                                if getattr(i,sort) > getattr(new_node,sort):
                                    del self.queue.queue[c]
                                else:  c+=1
                            else:  c += 1

                else:  pass


                self.queue.queue.append(new_node)


            self.depth_counter+=1
            self.current_node = self.queue.return_node()

        else:

            if self.search_algorithm == 'depth_first':  idx = -1
            else:  idx = 0


            if self.search_algorithm == 'uniform_cost':  self.queue.queue = sorted(self.queue.queue, key=lambda x: x.g_of_n)
            elif self.search_algorithm == 'best_first':  self.queue.queue = sorted(self.queue.queue, key=lambda x: x.h_of_n)
            elif self.search_algorithm == 'a_star':  self.queue.queue = sorted(self.queue.queue, key=lambda x: x.total_cost)
            else:  pass


            del self.queue.queue[idx]
            self.current_node = self.queue.return_node()
```

Fig. 6.Proposed Methodology

.



```
Move: Initial State
 Heuristic Cost: 5
 Total Cost: 0
 [1, 3, 4]
 [8, 6, 2]
 [7, 0, 5]

Move: down
 Heuristic Cost: 3
 Total Cost: 6
 [1, 3, 4]
 [8, 0, 2]
 [7, 6, 5]

Move: left
 Heuristic Cost: 4
 Total Cost: 8
 [1, 3, 4]
 [8, 2, 0]
 [7, 6, 5]

Move: down
 Heuristic Cost: 3
 Total Cost: 12
 [1, 3, 0]
 [8, 2, 4]
 [7, 6, 5]

Move: right
 Heuristic Cost: 2
 Total Cost: 15
 [1, 0, 3]
 [8, 2, 4]
 [7, 6, 5]

Move: up
 Heuristic Cost: 0
 Total Cost: 17
 [1, 2, 3]
 [8, 0, 4]
 [7, 6, 5]

Total Moves Made:  5
8 Puzzle Solved
 ~~Max Queue Length: 14
 ~~Nodes Popped: 12
 ~~Runtime: 0.0
 ~~Total Moves: 12
```

Fig. 6.Output

***Abstract*—- Breadth First 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.**

***Index Terms*—- Eight puzzle problem, Complete Solution, Breadthfirst searchs**

V. Introduction

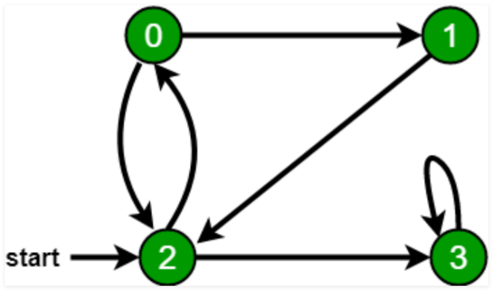
GRAPHS are a common representation in many problem domains, including engineering, finance, medicine, and scientific applications. Breadth-first search (BFS) is a crucial graph traversal algorithm used by many graph-processing applications. Different problems, such as VLSI chip layout, phylogeny reconstruction, data mining, and network analysis, map to very large graphs, often involving millions of vertices. Even though very efficient sequential implementations of BFS exist [1], [2], [3], they have work complexity of the order of number of vertices and edges. As a consequence, such sequential implementations become impractical when applied on very large graphs.

VI. Literature Review

Busato, Federico, and Nicola Bombieri. ”BFS-4K: an efficient implementation of BFS for kepler GPU architectures.” IEEE Transactions on Parallel and Distributed Systems 26.7 (2014): 1826-1838.

VII. Breadth First Search

BFS is one of the most import graph algorithms. It is used in several different contexts such as image processing, state space searching, network analysis, graph partitioning, and automatic theorem proving.



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 .

VIII. Conclusion

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.The sequential BFS algorithm labels vertices in increasing order of depth. Each depth level is fully explored before the next. The most popular parallel BFS algorithms are level synchronous. In level synchronous BFS, each level is processed in parallel as long as the sequential ordering of levels is maintained

Acknowledgment

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

References

[1] A.V Aho, J.E. Hopcroft and J.B. UlIman, 77x Design and Analysis of Computer Algorithms (Aii=n-Wesley, Readin MA, 1974).

[2] S. IB;ias;e, Computw AIgorit4rrs: ir;troduction to Design and Analysis (Addison-Wesley, Reading, MA, 1983).

[3] D. Coppersmith and S Winograd, Matrix multiplication via arithmetic progress;ons. Proc. 19th Ann. ACM S’ymp. on Theoqv of Cmwputing (May 1987) 1-6.

[4] V. Pan, How to Mdfiply Matrices Emter (Springer, BerI 1984).

[5] V. Strassen, Gaussian elimhtion is not optimal, Abnerische Mathematik 23 (1969) 354-3S6

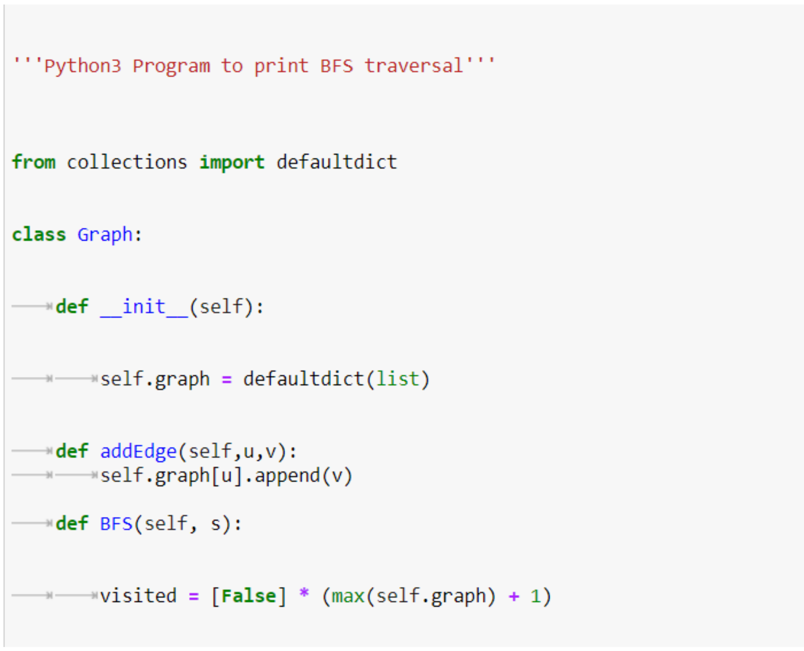
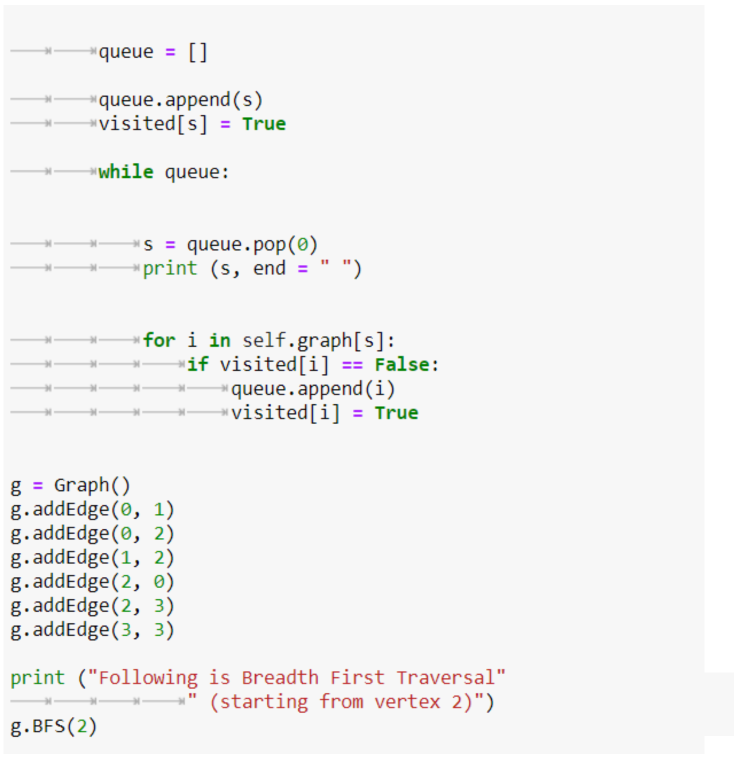
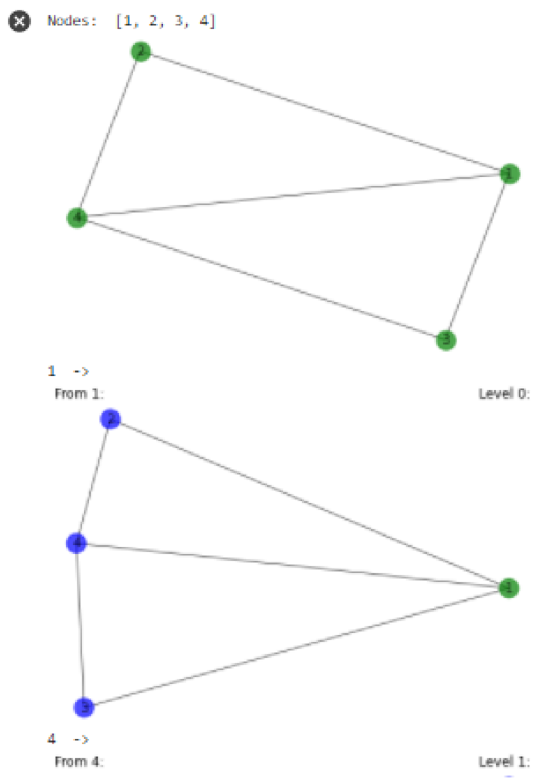

```
'''Python3 Program to print BFS traversal'''


from collections import defaultdict


class Graph:


    def __init__(self):


        self.graph = defaultdict(list)


    def addEdge(self,u,v):
        self.graph[u].append(v)

    def BFS(self, s):


        visited = [False] * (max(self.graph) + 1)
```

Fig. 7. Proposed Methodology



```
        queue = []

        queue.append(s)
        visited[s] = True

        while queue:


            s = queue.pop(0)
            print (s, end = " ")


            for i in self.graph[s]:
                if visited[i] == False:
                    queue.append(i)
                    visited[i] = True


g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print ("Following is Breadth First Traversal"
                " (starting from vertex 2)")
g.BFS(2)
```

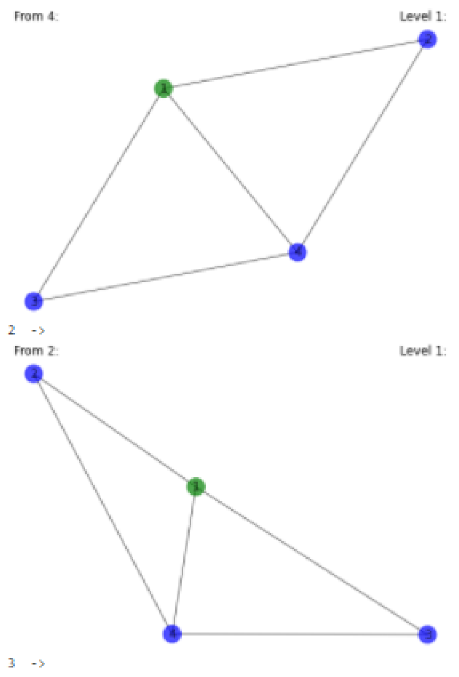
Fig. 8. Proposed Methodology



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

1  ->
From 1:                                    Level 0:

4  ->
From 4:                                    Level 1:
```

Fig. 9. Output

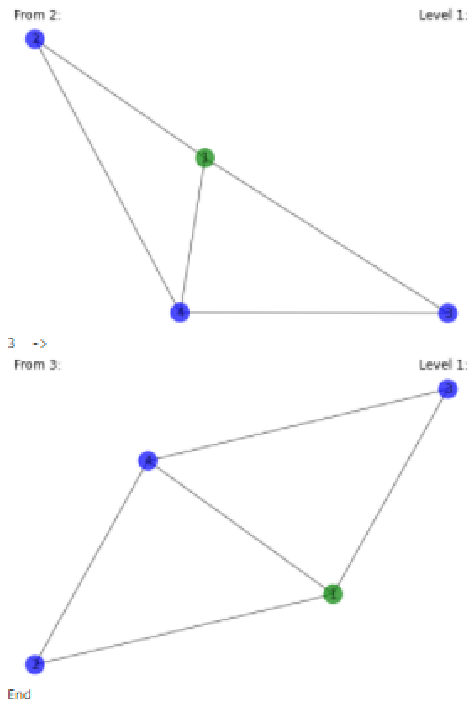
.



```
From 4:                                    Level 1:

2  ->
From 2:                                    Level 1:

3  ->
```

Fig. 10.Output

.



```
From 2:                                    Level 1:

3  ->
From 3:                                    Level 1:

End
```

Fig. 11.Output