***Practical 1***

**Aim:** *Implement Breadth first search algorithm for Romanian map problem.*

**Theory:**

**Romanian Map problem**

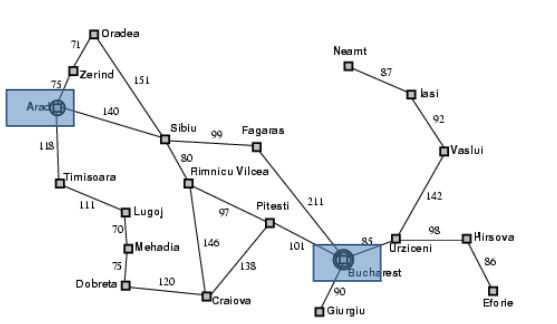
• In the state space view of the world, finding a solution is finding a path through the state space.

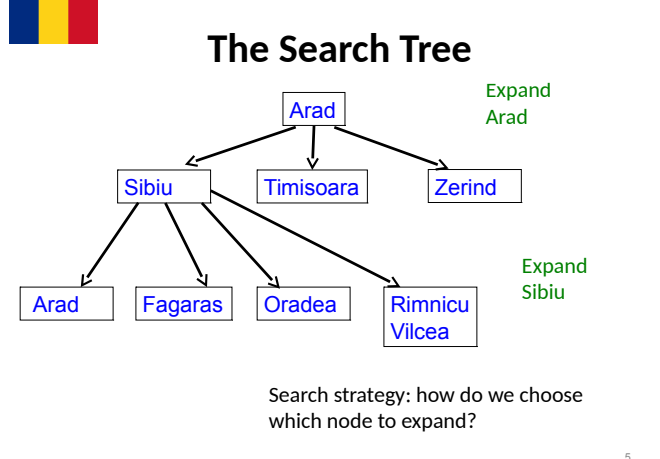
• When we (as humans) solve a problem like the 8-puzzle we have some idea of what constitutes the next best move.

• It is hard to program this kind of approach.

• Instead, we start by programming the kind of repetitive task that computers are good at.

• A brute force approach to problem solving involves exhaustively searching through the space of all possible action sequences to find one that achieves the goal.





***Search Tree Exploration***

• The tree is built by taking the initial state and identifying the states that can be obtained by a single application of the operators/actions available.

• These new states become the children of the initial state in the tree.

• These new states are then examined to see if they are the goal state.

• If not, the process is repeated on the new states.

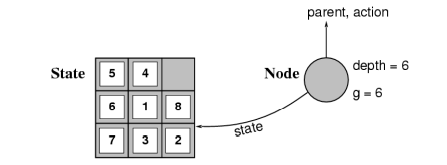
• We can formalise this description by giving an algorithm for it.

• We have different algorithms for different choices of nodes to expand.

***Implementation: States vs. Nodes***

• A state is a (representation of) a physical configuration.

• A node is a data structure constituting part of a search tree that includes state, parent node, action, path cost g(x), depth.



Expanding the tree creates new nodes, filling in the various fields and creating the corresponding states.

**General Algorithm for Search**

agenda = [initial state];

while agenda not empty do

pick node from agenda;

new nodes = apply operations to state;

if goal state in new nodes then

return solution;

else add new nodes to agenda;

*• Question: How to pick states for expansion?*

*• Two obvious strategies:*

*– depth first search;*

*–* ***breadth first search***

**Breadth First Search**

• Start by expanding initial state - gives tree of depth 1.

• Then expand all nodes that resulted from previous step gives tree of depth 2.

• Then expand all nodes that resulted from previous step, and so on.

• Expand nodes all at depth n before going to level n + 1.

***General Breadth First Search***

/\* Breadth first search \*/

agenda = [initial state];

while agenda not empty do

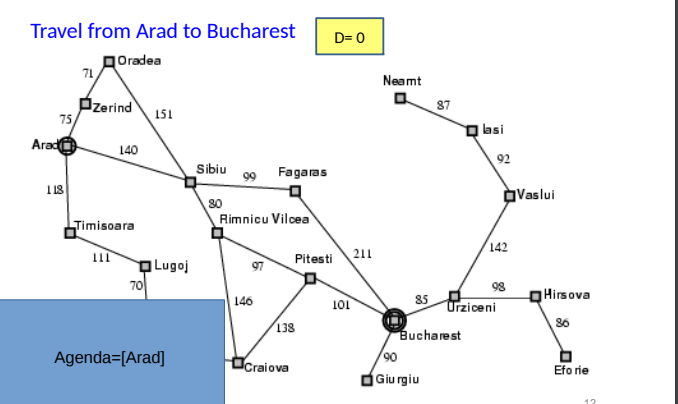
pick node from front of agenda;

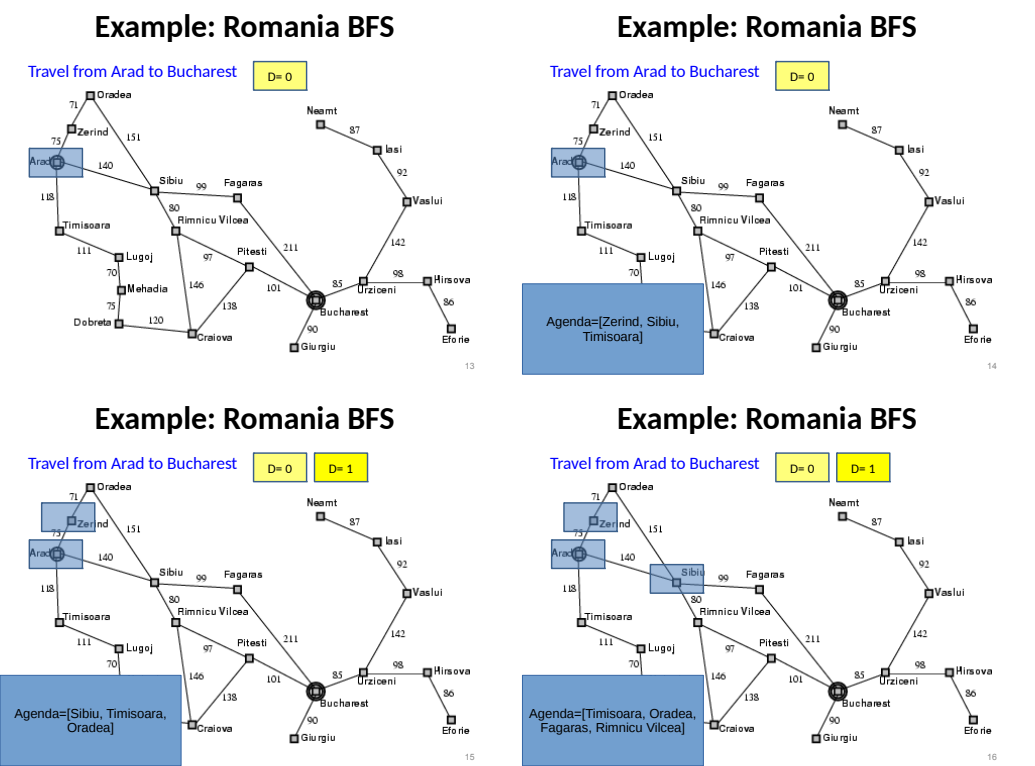
new nodes = apply operations to state;

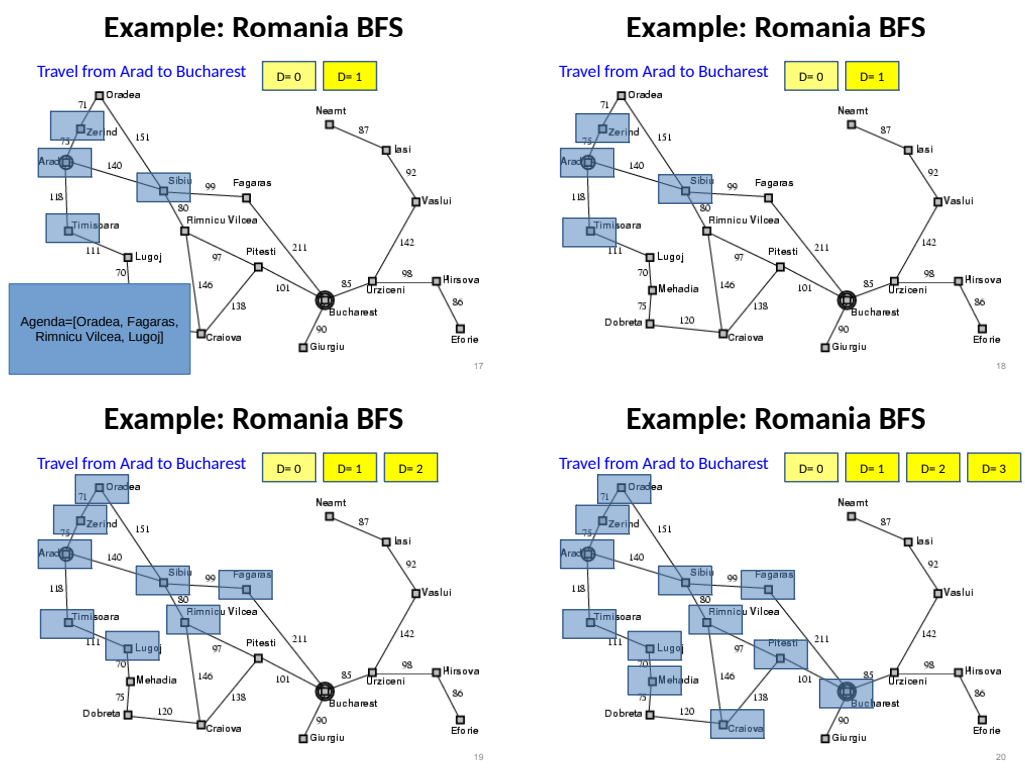
if goal state in new nodes then

return solution;

else APPEND new nodes to END of agenda

**

**

**

***Properties of Breadth First Search***

*• Advantage: guaranteed to reach a solution if one exists.*

*• Finds the shortest (cheapest) solution in terms of the number of operations applied to reach a solution.*

*• Disadvantage: time taken to reach solution.*

*– Let b be branching factor - average number of operations that may be performed from any level.*

*– If solution occurs at depth d, then we will look at*

*b + b2+ b3+ · · · + bd*

*nodes before reaching solution exponential.*

*– The memory requirement is bd*

***Code***

from queue import Queue

romaniaMap = {

    'Arad': ['Sibiu', 'Zerind', 'Timisoara'],

    'Zerind': ['Arad', 'Oradea'],

    'Oradea': ['Zerind', 'Sibiu'],

    'Sibiu': ['Arad', 'Oradea', 'Fagaras', 'Rimnicu'],

    'Timisoara': ['Arad', 'Lugoj'],

    'Lugoj': ['Timisoara', 'Mehadia'],

    'Mehadia': ['Lugoj', 'Drobeta'],

    'Drobeta': ['Mehadia', 'Craiova'],

    'Craiova': ['Drobeta', 'Rimnicu', 'Pitesti'],

    'Rimnicu': ['Sibiu', 'Craiova', 'Pitesti'],

    'Fagaras': ['Sibiu', 'Bucharest'],

    'Pitesti': ['Rimnicu', 'Craiova', 'Bucharest'],

    'Bucharest': ['Fagaras', 'Pitesti', 'Giurgiu', 'Urziceni'],

    'Giurgiu': ['Bucharest'],

    'Urziceni': ['Bucharest', 'Vaslui', 'Hirsova'],

    'Hirsova': ['Urziceni', 'Eforie'],

    'Eforie': ['Hirsova'],

    'Vaslui': ['Iasi', 'Urziceni'],

    'Iasi': ['Vaslui', 'Neamt'],

    'Neamt': ['Iasi']

}

def bfs(startingNode, destinationNode):

    # For keeping track of what we have visited

    visited = {}

    # keep track of distance

    distance = {}

    # parent node of specific graph

    parent = {}

    bfs\_traversal\_output = []

    # BFS is queue based so using 'Queue' from python built-in

    queue = Queue()

    # travelling the cities in map

    for city in romaniaMap.keys():

        # since intially no city is visited so there will be nothing in visited list

        visited[city] = False

        parent[city] = None

        distance[city] = -1

    # starting from 'Arad'

    startingCity = startingNode

    visited[startingCity] = True

    distance[startingCity] = 0

    queue.put(startingCity)

    while not queue.empty():

        u = queue.get()     # first element of the queue, here it will be 'arad'

        bfs\_traversal\_output.append(u)

        # explore the adjust cities adj to 'arad'

        for v in romaniaMap[u]:

            if not visited[v]:

                visited[v] = True

                parent[v] = u

                distance[v] = distance[u] + 1

                queue.put(v)

        # reaching our destination city i.e 'bucharest'

    g = destinationNode

    path = []

    while g is not None:

        path.append(g)

        g = parent[g]

    path.reverse()

    # printing the path to our destination city

    print(path)

# Starting City & Destination City

bfs('Arad', 'Bucharest')

***Output:***

*Arad to Bucharest*

******

*Arad to Neamt*

**

***Conclusion:***

*Implemented Breadth first search algorithm for Romanian map problem.*