# ARTIFICIAL INTELLIGENCE.

## LAB4

Sahil Satyam
RA1911003010675

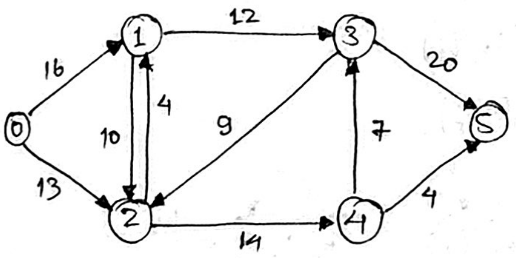
AIM: To study, understand and implement applications of BFS (Breadth First search) & DFS (Depth First Search).

- Breadth First Search:

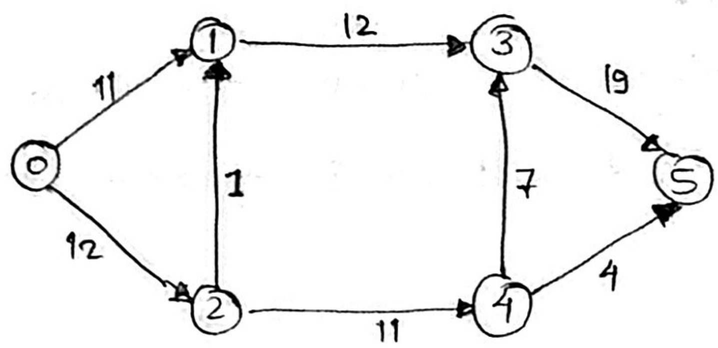
PROBLEM STATEMENT: Given a graph which represents a flow network where every edge has a capacity. Also given two vertices source "s" & sink 't' in the graph, find the maximum possible flow from s to t with following constraints:

- Flow on an edge doesn't exceed the given capacity of the edge.
- Incoming flow is equal to outgoing flow for every vertex except s and t.

PROBLEM SOLVING:




The maximum possible flow in the above graph is 23.




## ALGORITHM :

The following is a simple idea of Ford-Fulkerson Algorithm:

1) ~~start~~ with initial flow as 0.

2) while there is a augmenting path from source to sink.
   add this path flow to flow.

3) return flow.

Residual Graph of a flow network is a graph which indicates additional possible flow. Every edge of residual graph has a value called residual capacity which is equal to the original capacity of the edge minus current flow.

- Residual capacity is 0 if there is no edge between two vertices.
- we can initiate the residual graph as original graph as there is no initial flow and initially residual capacity is equal to original capacity.
- BFS builds the parent arr[] using which we traverse through the found path and find possible flow through this path by finding minimum residual capacity along the path.
- we need to update residual capacities, we subtract path flow from all edges along the path and we add path flow along the reverse edges.

```
# Maximum flow in a network:
from collections import defaultdict

class Graph:

    def __init__(self, graph):
        self.graph = graph
        self. ROW = len(graph)


    def BFS(self, s, t, parent):


        visited = [False]*(self.ROW)


        queue = []


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

        while queue:


            u = queue.pop(0)
            for ind, val in enumerate(self.graph[u]):
                if visited[ind] == False and val > 0:

                    queue.append(ind)
                    visited[ind] = True
                    parent[ind] = u
                    if ind == t:
                        return True

        return False



    def FordFulkerson(self, source, sink):


        parent = [-1]*(self.ROW)

        max_flow = 0
```

```
        while self.BFS(source, sink, parent) :


            path_flow = float("Inf")
            s = sink
            while(s !=  source):
                path_flow = min (path_flow, self.graph[parent[s]][s])
                s = parent[s]

            max_flow +=  path_flow


            v = sink
            while(v !=  source):
                u = parent[v]
                self.graph[u][v] -= path_flow
                self.graph[v][u] += path_flow
                v = parent[v]

        return max_flow




graph = [[0, 16, 13, 0, 0, 0],
        [0, 0, 10, 12, 0, 0],
        [0, 4, 0, 0, 14, 0],
        [0, 0, 9, 0, 0, 20],
        [0, 0, 0, 7, 0, 4],
        [0, 0, 0, 0, 0, 0]]

g = Graph(graph)

source = 0; sink = 5

print ("The maximum possible flow is %d " % g.FordFulkerson(source,
sink))
```

dfs.py × bfs.py ×

```python
graph = [[0, 16, 13, 0, 0, 0],
        [0, 0, 10, 12, 0, 0],
        [0, 4, 0, 0, 14, 0],
        [0, 0, 9, 0, 0, 20],
        [0, 0, 0, 7, 0, 4],
        [0, 0, 0, 0, 0, 0]]

g = Graph(graph)
```

7:14 Python Spaces: 4

RA19110030106 × RA19110030106 ×

Run Command: RA1911003010675/bfs.py Runner: Python 3 CWD ENV

```
The maximum possible flow is 23


Process exited with code: 0
```

Depth first search:

PROBLEM FORMULATION:

Lexicographic sorting of a given set of keys.
Given a set of strings, return them in alphabetical order.

PROBLEM SOLVING:

Lexicographic sorting of a set of keys can be accomplished with a simple trie-based algorithm by:

- Insert all keys into a trie.
- print all keys in the trie by performing pre-order traversal on trie to get output in alphabetically increasing order.

ALGORITHM:

To print the strings in algorithmic alphabetical order we have to first insert them in the trie and then perform traversal to print in alphabetical order.

The node of trie contain an index[] array which stores the index position of all the strings of arr[] ending at that node.

Except for the trie's left node all the other nodes have the size 0 for the index[] array.

```
#lexicographic sorting
class Trie:
    def __init__(self):
        self.key = None


        self.character = [None] * 26



def insert(head, s):

    # start from the root node
    curr = head

    for c in s:
        key = ord(c) - ord('a')

        # create a new node if the path doesn't exist
        if curr.character[key] is None:
            curr.character[key] = Trie()

        # go to the next node
        curr = curr.character[key]

    # store key in the leaf node
    curr.key = s


# Function to perform preorder traversal on a given Trie
def preorder(curr):

    # return if Trie is empty
    if curr is None:
        return

    for i in range(26):
        if curr.character[i]:
            # if the current node is a leaf, print the key
            if curr.character[i].key:
                print(curr.character[i].key)

            preorder(curr.character[i])
```

```python
if __name__ == '__main__':

    # given set of keys
    words = [
        'lexicographic', 'sorting', 'of', 'a', 'set', 'of', 'keys',
'can', 'be',
        'accomplished', 'with', 'a', 'simple', 'trie', 'based',
'algorithm',
        'we', 'insert', 'all', 'keys', 'in', 'a', 'trie', 'output',
'all',
        'keys', 'in', 'the', 'trie', 'by', 'means', 'of', 'preorder',
        'traversal', 'which', 'results', 'in', 'output', 'that',
'is', 'in',
        'lexicographically', 'increasing', 'order', 'preorder',
'traversal',
        'is', 'a', 'kind', 'of', 'depth', 'first', 'traversal'
    ]

    head = Trie()

    for word in words:
        insert(head, word)

    preorder(head)
```