Problem 1:

a) To the get desired result we need to slightly modify bfs and add a loop
Initialize visited[all nodes in tree] = false
Let r be the root of the tree.
Bfs(r)

```
Bfs(r):
   Create an empty queue q
    q.add(r)
    visited[r] = true
    while(q is not empty) {
         qSize = q.size()
          for(i=0;i<qSize;i++){
            node = q.pop()
            visited[node] = true
            output/print(node);
            for all children c of node
                    if visited[c] == false
                            q.add(c)
          }
     }
```

b)

```
Initialize visited[all nodes in tree] = false
Let r be the root of the tree.
dfs(r)

dfs(r):
  visited[r] = true
  for all children c of r
        if visited[c]==false
           dfs(c)
  output/print(node)
```

c)

```
for all v in vertices V initialize
      d[v] = c[v]
       visited[v] = false
dfs_maxnumsubtree(start_node)

dfs_maxnumsubtree(v):
    visited[v] = true
    for all children c of v:
          if visited[c] == false
                d[v] = max(d[v], dfs_maxnumsubtree(c))
    return d[v]
```

Running time: We call dfs for V vertices and for each vertex v in V we traverse just through its children and not all edges E. So running time is similar to that of DFS i.e O(|V| + |E|).

Correctness: Suppose leafs are at level L. For all leafs l the maximum number in subtree rooted at l is the number in the leaf node itself. When we move to level L-1. For subtree rooted at any node in L-1 level we call dfs with max function which returns maximum number among leaf nodes and node under consideration. Hence by method of induction if this works for L and L-1 levels it can be shown it will work for L-2, L-3 up till the root node. Hence proving the correctness of the algorithm.

problem 2:

We will present with two cases why Dijkstras algorithm doesn't work with negative edges and for graphs where negative edges are removed by adding a positive weight.

Following are the assumption of Dijkstra's algorithm:

- That all edges are positive and adding an edge will increase the path length.
- Selecting the minimum edge length (local optimum) at any given moment will eventually give us the shortest path to end vertex (global optimum).

These very assumptions fail in case of negative edges

- There can be negative edge which leads to series of positive expensive edges
- There can be an expensive edge (which we do not select because we're calculating local optima) which has path to end vertex via series of negative edges.

Following are the examples with explanation:

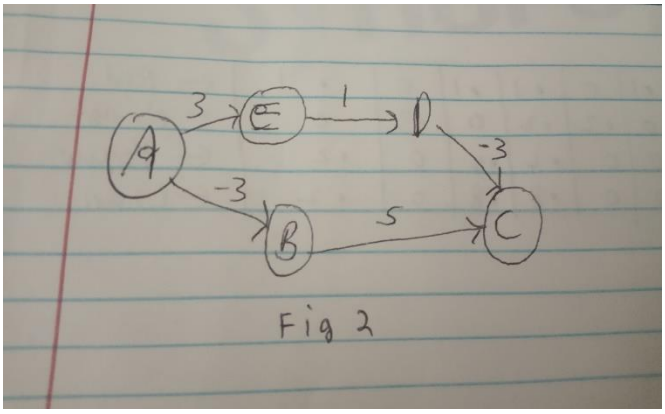


Fig 2

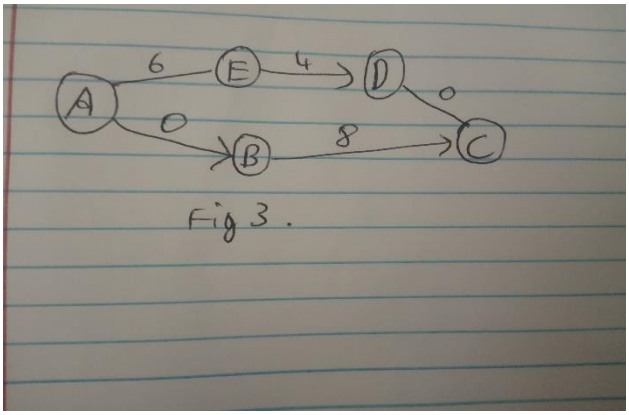
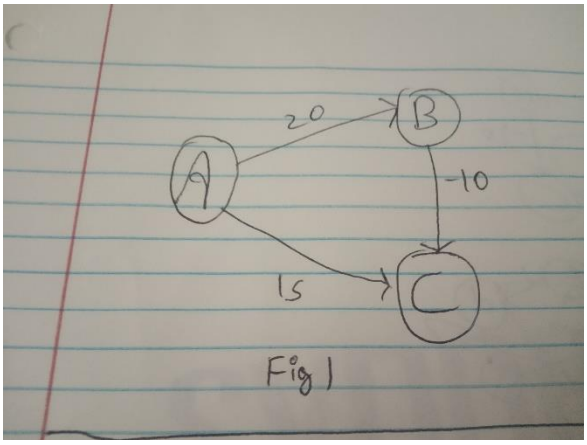


Fig 3.

In figure 2: Let's add -3 to all edges which results in figure 3 and compute shortest path from A to C

- According to figure 2 shortest path from A to C is through E and D with edge length 3 + 1 -3 =1
- When we add 3 to all edges in the graph, edge lengths change as per shown in the figure 3. Now when we run Dijkstra's algorithm on this graph we get a shortest path to C via B with path length 0 + 8 = 8.
- Though Dijsktras' algorithm gave us the shortest path from A to C but it technically changed the graph properties by adding positive weight to all edges. Irrespective of adding any positive weight the shortest path returned should have been the same!
- Dijkstras fails in this case because adding weight to every edge adds more weights to the longer paths than shorter paths.
- In our case we have increased path length of A->E->D->C by 6 but increased A->b->C path length only by 3.



In figure 1: let's compute distance from A->C

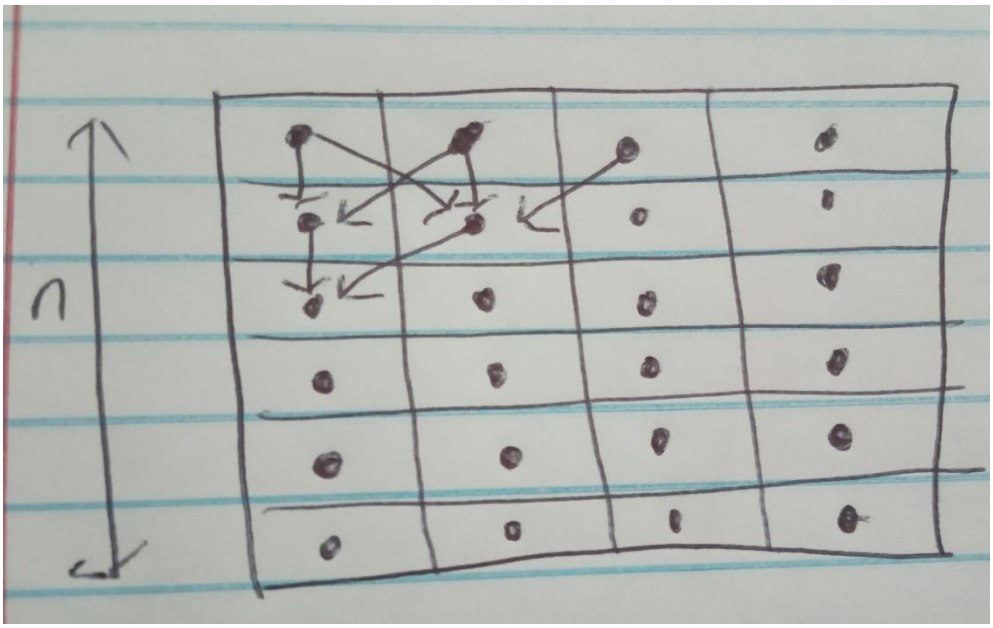
Assumption: we take an edge into consideration when its other vertex is in the Q

- We pop A from Q as it is the minimum and mark distance from A to B as 20 and A to C as 15.
- Next we pop out C. It has no edges.
- Next we pop out B but the Q is empty. B->C edge is not used for shortest path calculation. As per dijkstra's adding an edge from B to C will only result in increased path length as it assumes edge lengths will always be positive.
- A to C path is returned as 15 when it should have returned 10.

Hence even when we add positive weight to edges having negative edge lengths Dijkstras doesn't give shortest path to the end node as it increases path lengths in graph disproportionally.

Problem 3:

a)

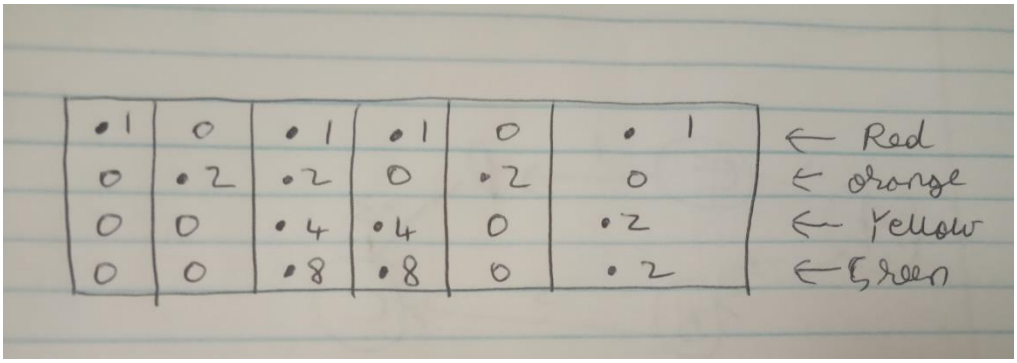


- consider n*m board where each slot has a peg and board is full. For each peg maximum number of paths to it from row above is 3 i.e from pegs directly and diagonally above it.
- The corner pegs have only 2 paths i.e one from above peg and the diagonal one.
- For computation of Omega we consider lower order terms so let's just consider number of paths to corner pegs.
- For each corner peg we have 2 paths coming in. Ignoring the number of pegs in each row assuming the number of rows n>>m is too large. We have 2 + 2 + 2 + 2+ ….+2(n times) = 2^n .
- Hence total number of possible path is 2^n and running time is lower bounded by Omega(2^n).
- For this peg configuration minimum number of paths is 2^n .
- It is easy to count paths when n is small, as n>10 it becomes tedious to manually keep track of all paths.
- Hence for large n we conclude counting paths manually is not possible.

b) For each yellow peg from left to right red->orange->yellow paths ending at that peg are 4,4 and 2 respectively.

c) For the leftmost green peg number of red ->orange ->yellow ->green paths ending at that peg are 8.

d)



Following is the strategy that was used to fill up the paths up to a peg in the above image:

- Consider n*m board.
- Initially set all empty positions to 0. Initialize the pegs in first row to 1 and rest with -1.
- **For i in range(1,n)** //starting from row 1 as row 0 is already initialized
  For j in range(0,m)
  If board[i][j] == -1 // there is a peg at (i,j)
  board[i][j] = board[i-1][j] +
  board[i-1,j-1] + board[i-1,j+1] // consider corner cases in code

number_paths = 0 //number rainbow paths will be stored in last row where green pegs are present

for j in range(0,m)

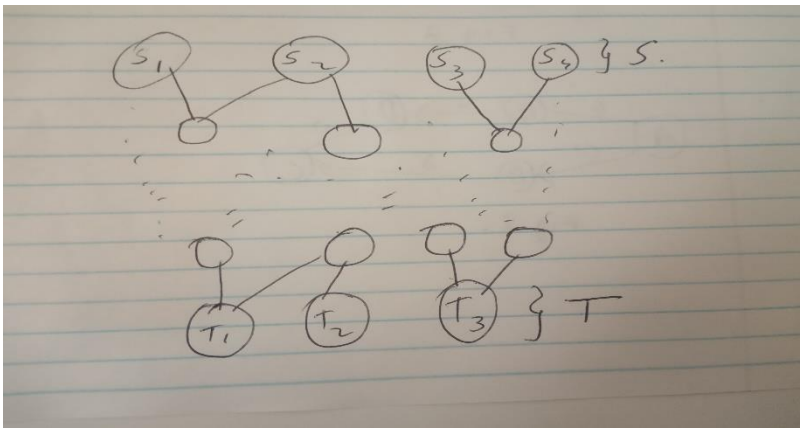
number_paths + = board[n-1][j]

return number_paths

Correctness:

- We initialize all the pegs in first row to value 1. We assume this because we start counting paths from pegs in row 1. So 1 value 1 is correct.
- For the pegs in row 2. Total number of paths depends on sum of number of paths of 3 vertices in the above row i.e **board[i][j] = board[i-1][j] + board[i-1,j-1] + board[i-1,j+1]**
- By proof of induction since it holds true for row 1 and row 2 which are base cases it holds for true for rest of the rows. Hence proved correctness.

e)



Input : A directed acyclic graph G = (V, E).

A set of start vertices S and end vertices T

Output: Total number of paths from S and end in T

Algorithm:

```
for all vertices v in {V-S}
    number_paths[v] = 0
for all vertices s in S
    number_paths[s] = 1
Initialize visited[of all vertices] = false
Q = create queue
Add each vertex s in S to Q and mark visited[s] = true
while(Q is not empty):
    u = Q.pop()
     for all neighbours of n of u
            if visited[n] == false
                Q.add(n)
                 visited[n] = true
            number_paths[n] + = number_paths[u]
total_paths = 0
for all vertices  t in T:
    total_paths + =number_paths[t]
return total_paths
```

Running time: We call bfs on each vertex i.e |V| times and for each vertex v in V we traverse just through its neighbours and not all vertices. Inner loop runs for |E| times. Hence runtime is same as BFS which is O(|V|+|E|)

Correctness:

- We initialize number_paths to all vertices in S to 1 as we're starting to count number paths from vertices in S. Rest number_paths to vertices are initialized to 0
- For each neighbour n of a vertex s1 in S we mark n as visited, add it to the queue and update number_paths[n] + = number_paths[s1].
- For each neighbour n of a vertex s2 in S if n is not visited we mark n as visited, add it to the queue. If n is visited we just update number_paths[n]+= number_paths[s2]

- After iterating over all vertices in level 0 ie in S we would have calculated total number of paths to each vertex n in Level 1 from S.
- As this hold true for base cases level 0 and level 1 when we finish the algorithm each vertex t in T would have paths to it from all nodes in S stored in number_paths[t].
- If we sum number_paths for each vertex t in T we will get total number of paths from nodes in S to nodes in T
- Hence proved correctness by induction.

Problem 4:

a) **Largest degree** of any vertex is **12** and **average** is **2.75826729497008**.

b) Let's consider an unweighted and undirected graph G with V vertices and E edges.

Input: G = (V, E), depth d and start vertex s € V

Output: Count of vertices that are at a distance d or less from v.

```
For all vertices v in V
        Initialize visited[v] = false
Number_vertices = 0
Bfs_modified(s,d)

 Bfs_modified(vertex v, depth d):
        Create queue Q
        Q.push(v)
        Visited[v] = true
        While(Q is not empty):
                If (d < 0):
                        break
                Qsize = Q.size()
                Number_vertices += Qsize
                For i from 0 to Qsize:
                        Vertex u = Q.pop()
                        For all neighbours n of vertex u:
                                If visited[n] == false:
                                        Q.push(n)
                d = d – 1
print(s,d,number_vertices)
```

Runtime only depends on number of vertices at a depth of d from start vertex. We exit after d<0.

NOTE: I have considered depth = 0(start vertex) as well for calculating total number of vertices from starting vertex to depth d.

c)

number of vertices at depth 5 from 926815 is 32
number of vertices at depth 5 from 231582 is 59
number of vertices at depth 5 from 99560 is 56
number of vertices at depth 5 from 579008 is 26
number of vertices at depth 5 from 298081 is 32
number of vertices at depth 5 from 740981 is 31
number of vertices at depth 5 from 1129377 is 41
number of vertices at depth 5 from 599839 is 53
number of vertices at depth 5 from 617718 is 44
number of vertices at depth 5 from 68807 is 48

number of vertices at depth 10 from 1158289 is 521
number of vertices at depth 10 from 53610 is 134
number of vertices at depth 10 from 695218 is 190
number of vertices at depth 10 from 1100570 is 301
number of vertices at depth 10 from 974225 is 108
number of vertices at depth 10 from 650086 is 113
number of vertices at depth 10 from 289480 is 151
number of vertices at depth 10 from 902449 is 242
number of vertices at depth 10 from 674056 is 303
number of vertices at depth 10 from 755282 is 409

number of vertices at depth 50 from 74839 is 14788
number of vertices at depth 50 from 1281810 is 23300
number of vertices at depth 50 from 544356 is 11097
number of vertices at depth 50 from 787593 is 8672
number of vertices at depth 50 from 905191 is 6181
number of vertices at depth 50 from 674089 is 21067
number of vertices at depth 50 from 614003 is 3761
number of vertices at depth 50 from 333936 is 12311
number of vertices at depth 50 from 132753 is 18516
number of vertices at depth 50 from 412653 is 7189

number of vertices at depth 500 from 887414 is 884848
number of vertices at depth 500 from 711407 is 1253588
number of vertices at depth 500 from 155656 is 790078
number of vertices at depth 500 from 609164 is 593624
number of vertices at depth 500 from 1066994 is 846511
number of vertices at depth 500 from 249657 is 1106307
number of vertices at depth 500 from 475986 is 1002336
number of vertices at depth 500 from 189614 is 309721
number of vertices at depth 500 from 1207414 is 596115
number of vertices at depth 500 from 438503 is 612660

yes, my code can handle depth of 500.