Anuj Kumar - 214297998

Assignment 04

Paradox

(a) Describe how to construct a graph to solve this problem?

Corresponding to each statement, Si, construct two vertices, one to denote that it is true (SiT) and another to denote that it is false (SiF). So for N statements, construct 2N vertices.

If the statement Si is that the statement Sj is True, then add two edges corresponding to the following two cases;

Case1: Assume, statement Si is True, then Sj is True. Add a directed edge from SiT to SjT.
Case2: Assume, statement Si is False, then Sj is False. Add a directed edge from SiF to SjF.

If the statement Si is that the statement Sj is False, then add two edges corresponding to the following two cases;

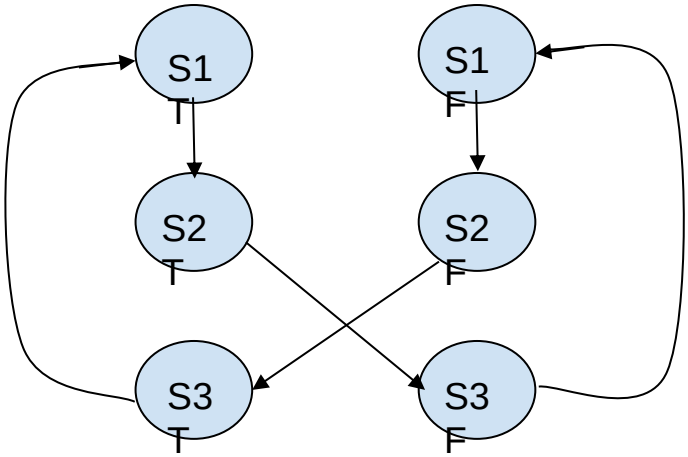
Case1: Assume, statement Si is True, then Sj is False. Add a directed edge from SiT to SjF.
Case2: Assume, statement Si is False, then Sj is True. Add a directed edge from SiF to SjT.

(b) The necessary and sufficient condition for the N statements to form a paradox is that there exists a path from SiT to SiF or a path from SiF to SiT. That means that the statement Si is both True and False at the same time.

(c) Consider the following example having 3 statements;

S1: S2 is True
S2: S3 is False
S3: S1 is True

The Graph constructed corresponding to the above set of statements is,



If we start from S1T, we can reach S1F. That means both S1 is both True and False at the same time which implies a contradiction.

Hence the algorithm is to detect reachability from SiT to SiF or SiF to SiT. We can devise an algorithm similar to the depth first search (DFS) for this as follows. Start from each vertex Si. Since each vertex has only one edge out, simply explore the path until an already explored vertex is visited again or an opposite vertex (SiF is the opposite of SiT and vice versa) is visited. If an opposite vertex is visited at any time, then declare it as a **paradox**.

(d) Since each vertex has exactly one edge out of it, there are 2N edges. It is enough to explore each path only once to detect any conflicts for a vertex. But this has to be done for each vertex in the worst case. So the worst time complexity is $O(N^2)$ where N is the number of statements.

(e) Pseudocode

```
// we define a function to get the opposite of a vertex
// e.g. for S1T the opposite is S1F and vice versa
String opposite(vertex_name) {
   If vertex_name ends in T then
        replace it with F
   Else
        Replace it with T

   Return vertex_name
}

bool paradox(String[] A) {
   // construct a Graph G as described in (a)
   N = size of A // number of statements
   // Graph G has 2N vertices
```

```
G = empty graph
For i = 1 to N {
    Add two vertices SiT and SiF to graph G
}
// Add the edges
For i = 1 to N {
     If A[i] is "statement Sj is True" then {
        Add an edge from SiT to SjT
        Add an edge from SiF to SjF
     }
     Else if A[i] is "statement Sj is False" then {
        Add an edge from SiT to SjF
        Add an edge from SiF to SjT
     }
// Perform DFS like search for checking reachability
// start from a vertex and follow path until an already visited vertex is reached
// or a paradox is found
For each vertex u in G {
     // initialize all vertices to be unvisited initially
     For each vertex v in G {
        Set v.visited = False
     }
     while(u.visited is False) {
        u.visited = True
        For the edge u->v in G, if v.visited is False {
            If v == opposite(u) {
               return True
            }
            u = v
        }
     }
}
```

Anuj Kumar - 214297998

Bucket Flips

(a) The graph is constructed as follows; A vertex will be of the form (a,b) where a denotes the number of litres in bucket A and b denotes the quantity in bucket B. If the capacity of bucket A is m and that of bucket B is n, then the total number of vertices is given by (m+1)*(n+1). E.g. vertex (0,0) indicates that both the buckets A&B are empty. Vertex (m, n) indicates both the buckets A&B are full.

The edges from a vertex (a,b) are found out as follows.

1. Bucket A can be emptied. Then add an edge from (a,b) to (0,b).
2. Bucket B can be emptied. Then add an edge from (a,b) to (a,0).
3. Bucket A can be filled. Then add an edge from (a,b) to (m,b).
4. Bucket B can be filled. Then add an edge from (a,b) to (a,n).
5. Flip water in bucket A to B. Add edge from (a,b) to (max(0,a+b-n), min(n,a+b)).
6. Flip water in bucket B to A. Add edge from (a,b) to (max(a+b-m,0), min(m,a+b)).

If we need to measure k litres, then the target states are either (k, b) or (a, k) where a & b can be any integers. I.e. Either of the buckets contains k litres of water.

(b) The algorithm works as follows; Initialize all the vertices of the graph to be unvisited. The initial state is (0,0), i.e. both the buckets are empty. Consider all possible moves from this state. These are either to fill A (which leads to the vertex (m,0)) or to fill B (which leads to the vertex (0,n)). Then from each of these vertices consider all possible moves. But never visit a vertex which is already visited. Hence the breadth first search (**BFS**) algorithm is used which visits the vertices in a breadth first manner. Along with each vertex, keep its history; i.e. which moves are performed to reach that particular vertex. So as soon as a target vertex is found, its history is printed. This is guaranteed to be the shortest move because the BFS explores all moves at a depth of d before exploring moves of depth r+1.

(c) The worst case run time is that of the BFS which is simply $O(V + E)$. Since the number of vertices is mn and each vertex can have at most 6 edges, the worst case run time is $O(mn)$.

(d) pseudocode

```
bucket_flip(m, n, k) {
   // Create the graph and add the vertices
   G = empty graph
   For i = 0 to m {
       For j = 0 to n {
          // Add a new vertex
          v = vertex(i,j)
          G.add(v)
          // set the vertex as unvisited
          v.visited = False
       }
   }
   // perform BFS
   // use a Queue to keep track of vertices to explore
   Q = empty Queue
   // Add the start vertex (0,0) to the Q
   v = vertex(0,0)
   v.visited = True
   v.history = “”
   ENQUEUE(Q, v)
   While Q is not empty {
       u = DEQUEUE(Q)
       // check if we have reached the target
       If u[0] == k or u[1] == k {
          print(u.history)
          // stop execution
          return
       }
       For each vertex v in G.Adj(u) {
          If v.visited == False {
              v.visited = True
              v.history = u.history + move to reach v from u
              ENQUEUE(Q, v)
          }
       }
   }
   // Q becomes empty then we can’t reach the target
   print(“NULL”)
   return
}
```