ASHWIN SHENAI | HOMEWORK 3 | ES0207A
180156 | | 15/10/2019

Problem 1. GRAPH REVERSAL

1] The adjacency matrix of graph G is the matrix $M = [m_{ij}]_{v \times v}$ where

$$m_{ij} = \begin{cases} 1, & \text{if } (i,j) \in E \\ 0, & \text{otherwise.} \end{cases}$$

The adjacency matrix of the graph G^R is the matrix $M^R = [m'_{ij}]_{v \times v}$ where

$$m'_{ij} = \begin{cases} 1, & \text{if } (i,j) \in E^R \\ 0, & \text{otherwise.} \end{cases}$$

For $\forall (i,j) \in E^R$ we know that $(j,i) \in E$.

$\Rightarrow$ if $m_{ij} = 1$ then $m'_{ji} = 1$

if $m_{ij} = 0 \Rightarrow (i,j) \in E \Rightarrow (j,i) \notin E^R \Rightarrow m'_{ji} = 0$

$\therefore m_{ij} = m'_{ji} \quad \forall 1 \leq i,j \leq v. \Rightarrow M^R = M^T$

Hence adjacency matrix of G^R (M^R) can be found from adjacency matrix of G (M) by simply taking the transpose. Since transposing the matrix involves flipping each element across the diagonal, such an algorithm would take $\Theta(|V|^2)$ time. Consider the following algorithm.

~~R-MATRIX(G)~~
~~for i=0 to G.V~~
~~for j~~

We create a new matrix using the above property $m'_{ij} = m_{ji}$ and return it as the adjacency matrix of the reversed graph.

```
R_MATRIX(G)
1.  M[v][v] = 0 // zero matrix.
2.  for i = 0 to G.V
3.      for j = 0 to G.V
4.          M[j][i] = G.adj[i][j]
5.  return M.
```

Time complexity: ~~Assuming~~ line 1 takes $O(|V|^2)$ time to initialize each cell to zero. Since line 4 is a const. time operation lines 2-4 takes $O(|V|^2)$. Hence the algorithm runs in $O(|V|^2)$ time overall.

180156
ASHWIN SHENAI

HOMEWORK 3

ES0207A
15/10/2019

1) 2] The adjacency matrix M^R for the graph G^R is given by $[m_{ij}]_{v \times v}$ where

$$m_{ij} = \begin{cases} 1, & (i,j) \in E^R \Rightarrow (j,i) \in E \\ 0, & \text{otherwise.} \end{cases}$$

if $(j,i) \in E$ then $i \in G.Adj[j]$. Hence we can iterate over the adjacency list of each vertex v and for every vertex u in the list of v we set $m_{uv} = 1$ since $(v,u) \in E \Rightarrow (u,v) \in E^R$.

Consider the following algorithm

```
LIST-TO-MATRIX(G).
1  |M[v][v] = 0    //zero matrix
2  |for each v in G.V
3  |  |for each u in G.adj[v]
4  |  |  |M[u][v] = 1
5  |return M
```

For each vertex u in the adjacency list of v, we know $(v,u) \in E$. So $(u,v) \in E^R$, which means m_{uv} should be 1. We do this for all vertices in V.

Time complexity: (Assuming line 1 takes constant time)

Since line 4 runs in constant time, the loop at line 3 will take $\Theta(|Adj[v]|)$ time. Since loop at line 2 will runs for all vertices, total time taken by the loop is

$O\left(\sum_{v \in V} |Adj[v]|\right) = O(|E|)$ to run all iterations of line 4

The loop also takes constant time to access the adjacency list of v so running line 3 v times takes $O(|V|)$ time. Hence total running time for the loop would be $O(|V|+|E|)$.

So assuming line 1 takes constant time the algorithm runs in $O(|V| + |E|)$ time. If line 1 takes constant time to initialize each cell it would take $O(|V|^2)$ time, in which the running time of the algorithm would actually be $O(|V|^2+|E|)$.

1) 3] Let s be a source vertex of G. Then $\nexists\ v \in G$ s.t $(v, s) \in E$. since G & G^R share the same set of vertices, and if $(v,s) \notin E$ then $(s, v) \notin E^R$, we have that for the vertex s $\nexists\ v \in G^R$ s.t $(s, v) \in E^R$ since there are no (v,s) edges in E. This implies that s is a sink vertex for G^R.

Similiarly let u be a sink vertex of G. Then $\nexists\ v \in G$ s.t $(u,v) \in E$ Since G & G^R have same vertices and if $(u, v) \notin E$ then $(v, u) \notin E^R$ we have that for the vertex u.

$\nexists\ v \in G^R$ s.t $(v, u) \in E^R$ since there are no (u,v) edges in E. This implies that u is a source vertex for G^R.

Hence all source vertices of G become sink vertices for G^R and all sink vertices of G become source vertices for G^R.

Let C be a strong component of G. then by definition, $\forall\ u, v \in C$, u & v are both reachable from each other i.e. $u \rightsquigarrow v$ and $v \rightsquigarrow u$. This implies that there is a cycle involving u & v. in G. This cycle will inherently also be present in G^R, only with its direction reversed. So $u \rightsquigarrow v$ and $v \rightsquigarrow u$ i.e u & v are still both reachable from each other in G^R. Since we chose u & v arbitrarily from C, this should hold for all pairs of vertices of C in G^R, implying that C is also a strong component of G^R. Since C was an arbitrary strong component of G, we can conclude that G & G^R have the same set of strongly connected components

ASHWIN SHENAI
180156

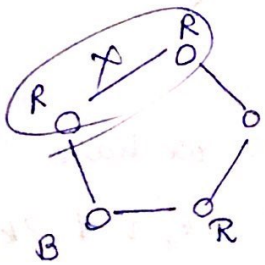
HOMEWORK 3

ESO207A
14/10/2019

Problem 2 - BIPARTITE GRAPHS

1] A graph G is bipartite iff it can be two-colored i.e. for every vertex u coloured red all the immediate neighbours are coloured blue and similiarly for every blue vertex v all the immediate neighbours are red. If a graph is bipartite, we can colour all vertices in V_1 red & all vertices in V_2 blue to generate a valid two-coloring. If a graph has a valid two coloring, we can designate the red vertices as V_1 & blue vertices as V_2. Since all edges have one red vertex & one blue vertex, the graph must be bipartite.

Furthermore, a graph has a valid two-coloring iff it has no odd cycles. We prove this by contradiction, assuming that a graph G with a valid two-coloring has an odd cycle of length 2k+1 with vertices $(v_1, v_2, v_3 \ldots v_{2k+1})$ where v_i has an edge to v_{i+1}, $i \leq 2k$ and v_{2k+1} has an edge to v_1. Without loss of generality, let us assume that v_1 is coloured red. Then in accordance to the rule of two colouring, v_2 must be blue, v_3 must be red, v_4 must be blue & so on. i.e v_{2i+1} must be red & v_{2i} must be blue $\forall$ $i = 1, 2 \ldots k$



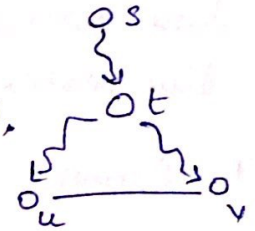
Ex. violation

Since v_{2k} must be blue, v_{2k+1} must be red. But here the rule of colouring is violated since v_{2k+1} is connected to v_1 & both are red vertices which is not allowed, contradicting the fact that the graph had a valid two-colouring.

Thus since a graph with a valid two-colouring is necessarily bipartite, a graph cannot be bipartite if it has odd cycles.

2) 1] Consider the BFS tree rooted at vertex 's' of the graph G. Let the vertex 't' be the lowest common ancestor of two vertices 'u' & 'v' in this tree s.t $(u,v) \in E$. Note that $|u.d - v.d|$ can only either be 0 or 1, since u & v can either at the same depth, or if they are at different depths the levels must be adjacent since they can be traversed by a single edge.

Consider the case when $|u.d - v.d| = 0$. This implies u & v are at the same level in the BFS tree. Then t, u, v form a cycle. The length of this cycle would be the path lengths of $t \to u$ & $t \to v$ plus 1. The path length $t \to u$ is simply $t.d - u.d$, similiarly length $t \to v$ is $t.d - v.d$.



BFS tree (sort of).

Since $v.d = u.d$, let $t.d - u.d = t.d - v.d = k$.
The length of the t-u-v cycle is thus $2k+1$, which is odd.
Hence if $|u.d - v.d| = 0$, the graph contains an odd cycle and so cannot be bipartite.

Consider the case when $|u.d - v.d| = 1$. Since $s.d = 0$, we have that $(u.d + v.d) \bmod 2 = 1$. i.e either $u.d = 2k$, $v.d = 2k+1$, or $u.d = 2k+1$, $v.d = 2k$. In this case we can form a valid two-coloring of the graph by coloring all vertices with even .d values red and coloring the odd ones blue. Since then either one of u is red & v is blue for all $(u,v) \in E$, the coloring is valid. Since the coloring is valid, we have that if $|u.d - v.d| = 1$, the graph is bipartite.

2) 1] Consider a graph that is bipartite. Without loss of generality let the vertex 's' be red. Since the graph has a valid two-coloring and $s.d = 0 = 2k$ for $k = 0$, all the blue vertices will have odd 'd' values & all the red vertices will have even 'd' values as the red & blue vertices will appear alternately in the BFS tree. Since per the rule of colouring either one of u is red & v is blue for all $(u,v) \in E$, we have that $|u.d - v.d| = 1 \quad \forall (u,v) \in E$ if G is bipartite

Hence G is bipartite iff $|u.d - v.d| = 1 \quad \forall (u,v) \in E$.

2] We can check whether G is bipartite by running BFS on G and checking whether a valid two-coloring is obtained, since as proved above if G is bipartite, it must have a valid two-colosing in the BFS tree. ~~The algorithm is as follows:~~

```
CHECK_BI(G,S)
    for each v in G.V
        V.color = WHITE
        v.d = -1
    S.color = RED
    S.d = 0
    Q = ϕ
    ENQUEUE(Q,S)
    while(!EMPTY(Q))
        u = DEQUEUE(Q)
        if u.color == WHITE
            u.color = -1 * s
```

For convenience we define the colours in the form of integers as follows:

WHITE = 0 : unvisited vertex

RED = 1

BLUE = -1

So that we can switch from RED to BLUE simply by multiplying by -1.

ASHWIN SHENAI
180156

HOMEWORK 3

ES0207A
14/10/2019

2) 2]

```
CHECK-BI(G,s)
1  for each v in G.V
2      v.color = WHITE
3      v.d = -1
4  s.color = RED  // could be BLUE
5  s.d = 0
6  Q = ϕ
7  ENQUEUE(Q,s)
8  while(!EMPTY(Q))
9      u = DEQUEUE(Q).
10     for each v in G.adj[u]
11         if v.color == WHITE
12             v.color = (-1)*u.color
13             v.d = u.d+1
14             ENQUEUE(Q,v)
15         else if v.color != (-1)*u.color
16             return false
17 return true.
```

Proof of correctness:

The algorithm runs BFS on G starting from 's'. We color s red and then run BFS. At every iteration of BFS, if the node is unvisited we flip & set the color and enqueue it. If it is visited, we check whether it has the opposite color. If it does not, the graph is not bipartite and we exit with F. If the BFS routine finishes without exiting graph is bipartite.

Time Complexity: The loop in lines 1-3 runs in $\Theta(|V|)$ time.
Lines 4-17 take constant time.

Assuming line 9 takes c_1 time & lines 10-16 take c_2 time, each iteration of the while loop takes $c_1 + c_2|adj[u]|$ time. Since the while loop at most runs for all vertices, time it takes is $\leq \sum_v c_1 + c_2|adj[u]|$
$= \Theta(|V| + |E|)$. Hence time complexity of the algorithm is $\Theta(|V|+|E|)$

Alternatively, instead of checking for two-coloring, one could also check for $|v.d - u.d| \neq 1$ at line 15. This would not change the time complexity & equivalence of both conditions has been proved earlier.

ASHWIN SHENAI
180156

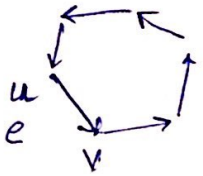
HOMEWORK 3

ES0207A
14/10/2019

Problem 3. EDGE CYCLE

The graph G has a cycle involving the edge $e = (u, v)$ iff there is a path from the vertex v to the vertex u, since if a cycle involves a e, it contains u & v and so must have both paths from u to v (e) and v to u as well.

We can check for a path from v to u by running a DFS on the graph starting from the vertex v, and seeing if any path ends up at u. If we reach u in the DFS ~~forest~~ tree rooted at v, we will have our cycle involving e.



cycle of 'e'

For convenience, define the following vertex colours:

WHITE : unvisited vertex

GRAY : visited but unprocessed vertex.

BLACK : visited & processed vertex,

where processing implies the adjacency list of that vertex is being traversed.

(For simplicity timestamps are omitted and parent is not stored)

→

3]

```
FIND_CYCLE(G,e).
1 | for each v in G.V
2 |   | v.color == WHITE
3 | (u,v) = e
4 | return CHECK(G, v, u)

CHECK(G, v, u).
1 | v.color = GRAY
2 | flag = false
3 | for each x in G.adj[v]
4 |   | if x == u
5 |   |   | flag = true
6 |   | else if x.color == WHITE
7 |   |   | flag = CHECK(G, x, u)
8 |   | if flag == true
9 |   |   | break
10| v.color = BLACK
11| return flag
```

The algorithm FIND_CYCLE (G,e) returns true if G has a cycle involving e. It recursively calls the CHECK subroutine which is a slightly modified version of DFS_VISIT. We maintain a flag throughout the recursive call stack to keep track of whether u was encountered as the neighbour of any vertex of the DFS tree. As soon as the flag sets to true, the loop exits and control falls through to the highest function call and the program terminates with a 'true' result. If u was not encountered at the current vertex's adjacency list we go deeper down the tree by calling CHECK on the current vertex's neighbours until all vertices have been visited

Time complexity analysis:

Since lines 2,3 takes O(1) time, the loop at line 1 iterates line 2 for |V| times so running time for lines 1-3 of FIND_CYCLE is O(|V|). Note that throughout the complete recursive call stack for the CHECK function, each node will be visited at most once. All operations such as checking if u is a neighbour, marking the flag and colour are done in O(1) Thus O(1) is spent per edge of the graph. as line 3 of the CHECK function represents all the edges connected to that vertex. Hence total time for line 4 of FIND_VISIT is O(|E|). Total running time for the algorithm is hence O(|V| + |E|).

Problem 4 - ALTERNATIVE TOPOLOGICAL SORT

In order to implement the given outline we need subroutines to find a source vertex and delete a vertex from the graph.

A source vertex v can be identified by checking that v does not appear in the adjacency list of any other vertex u, since otherwise $(u,v) \in E$ which contradicts the fact that v is a source.

Also, each time we delete a vertex, the edges through that vertex will also be removed and also the adjacency list will change. So instead of iterating through the new adjacency list everytime and finding a source vertex, it is more efficient to maintain an array A containing the number of occurences of each vertex in the adjacency lists of other vertices and a stack of all the current source vertices. This way one can easily delete the required vertices by simply updating the array A as effectively we only need that information to find the sources vertices. We can also check the values in array A after each iteration to find the new source vertices instead of updating the adjacency lists and iterating over them again.

The algorithm would roughly look like

```
ALT-TOP-SORT
    Compute array A.
    Push source vertices onto stack
    while stack not empty
        delete a source vertex & print it
        update array A
        find new source vertices and push onto stack.
```

Since the graph has a valid topological sort, it must be acyclic. Then it also must have a source vertex v.

ASHWIN SHENAI
180156

HOMEWORK 3

ES0207A
15/10/2019

4) We can find that vertex v by starting at an arbitrary node and traversing the graph in reverse. Since the graph is acyclic no node would be entered twice and so the process would evidently terminate at a source vertex as the number of vertices is finite & no node is encountered more than once. Removing this source vertex will again result in an acyclic graph with one less node. Since the existence of a source vertex is always guaranteed irrespective the number of vertices and we have finite vertices, at some stage the graph will become null and removal of source vertices will thus terminate. Hence the algorithm will run until the graph is empty or in other words our stack of source vertices will be empty only when the graph is empty.

Following are the required subroutines:

(1.) Compute array A.

```
1 | A[v] = 0  // zero array.
2 | for each u in G.V
3 |   | for each v in G.adj[u]
4 |   |   | A[v]++;
```

Time complexity: Since line 4 takes const. time, line 3 runs in $O(|adj[v]|)$ time loop at line 2 runs loop at line 3 for all times so total time is $O(\Sigma|adj[v]|) = O(|E|)$. Line 1 takes $O(|V|)$ time so net time taken is $O(|V|+|E|)$.

(2.) Find all source vertices & push onto stack

```
1 | stack S;
2 | for each v in G.V
3 |   | if A[v] == 0
4 |   |   | push(S, v)
```

Time complexity: lines 1, 3, 4 are const. time. line 2 runs V times, at most, so running time for this routine is $O(|V|)$.

(3.) Delete source vertex u & update A.

```
1 | for each v in G.adj[u]
2 |   | A[v]--; G.E = G.E - {(u,v)}
3 | G.V = G.V - {u}
```

Time complexity: This routine takes time $O(|adj[u]|)$.

4) Using the above subroutines, we can now write the complete algorithm to implement the given outline

```
ALT_TOP_SORT(G)
1   stack S;
2   int A[|G.V|];
3   for each v in G.V                          O(|V|)
4       A[v] = 0.
5   for each u in G.V                          O(|V|+|E|)
6       for each v in G.adj[u]
7           A[v] = A[v]+1
8   for each v in G.V                          O(|V|)
9       if A[v] == 0
10          push(S,v)
11  while(!EMPTY(S))                           O(|V|+|E|)
12      u = POP(S)
13      for each v in G.adj[u]
14          A[v] = A[v]-1
15          if A[v] == 0
16              push(S,v)
17          G.E = G.E - {(u,v)}
18      print(u)
19      G.V = G.V - {u}
```

Note that lines 17 & 19 which involve removal of an element from a list can be implemented in constant time using resizable arrays. We swap the element to be removed with the last element & then decrement the size of the array in O(1) time.

Time Complexity analysis:

Lines 1, 2, 4, 7, 9, 10, 12, 14-19 all take O(1) time. The loop at line 3 runs |V| times, it thus takes O(|V|) time. Lines 5-7 are subroutine ①, which takes O(|V|+|E|) time. Lines 8-10 are subroutine ②, which takes O(|V|) time since lines 14-17 are O(1) time & loop at line 13 runs |adj[u]| times, the loop takes O(|adj[u]|) time. Lines 12, 18, 19 take O(1) time so each iteration of the while loop takes $O(1) + O(|adj[u]|)$ time. As proved earlier, the stack S is empty only when $G.V = \phi$, which implies that since each element will be pushed at most once, the while loop runs |V| times, once per vertex. So lines 13-19 take running time $\sum_{u \in V} O(1) + O(|adj[u]|) = O(|V|+|E|)$.

Hence net running time of the algorithm

$= O(|V|) + O(|V|+|E|) + O(|V|) + O(|V|+|E|) = \underline{O(|V|+|E|)}$

180156
ASHWIN SHENAI

HOMEWORK 3

ES0207A
15/10/2019

Problem 5. ALL-PATH VERTEX

Lemma. If there exists such a vertex u which has paths to all vertices in G, then the finishing time for u in a DFS of G will be the highest of all vertices. i.e. if an 'all-path' vertex exists then it will be the last-finished vertex in a DFS traversal of the graph.

Proof: Let us assume that the required vertex is v, and that it is not the last finished vertex. Then $\exists\, u \in V$ s.t u finishes after v, implying that there is an edge $(u, v) \in E$ since predecessors finish only after their descendants. This implies that u is also connected to all vertices in V, as it is connected to the required vertex v. Hence we can replace v with u as the required vertex, maintaining the fact that the last finished vertex is an 'allpath' vertex.

So if the required vertex exists it will be the last finished vertex. However the required vertex may not exist at all if the graph is not properly connected. So, we need to check whether the last-finished vertex is actually connected to all the other vertices after we have found it.

We can find the last-finished vertex by simplying executing a complete DFS traversal of the graph. We can then check if that vertex is an allpath vertex by executing another DFS traversal starting from that vertex and counting the number of vertices visited.

5) Consider the following algorithm:

```
FIND_ALLPATH(G)
1  DFS(G)
2  u = NIL
3  max = -∞
4  for each v in G.V
5      if v.f > max
6          max = v.f
7          u = v
8  for each v in G.V
9      v.color = WHITE
10 if(CHECK(G,u) == |V|)
11     return u
12 else
13     return NIL
```

We first run the DFS on the graph G. This gives us finishing times for all the vertices. Then we select the vertex with the maximum finishing time. Using the CHECK routine, a slightly modified version of the VISIT routine, we check whether the max. finish time vertex is an all path vertex by recursively counting the number of calls / nodes visited. If the count returned equals the number of vertices we return the selected vertex else we return NIL since as proved above, if an all path vertex exists it must be the vertex with the highest finishing time.

```
CHECK(G,u)
1  u.color = GRAY; count = 1;
2  for each v in G.adj[u]
3      if v.color = WHITE
4          count = count + CHECK(G,v)
5  u.color = BLACK
6  return count
```

```
DFS(G)
1  for each v in G.V
2      u.color = WHITE
3  time = 0
4  for each v in G.V
5      if v.color = WHITE
6          VISIT(G,v)
```

```
VISIT(G,u)
1  time = time + 1
2  u.d = time
3  u.color = GRAY
4  for each v in G.adj[u]
5      if v.color = WHITE
6          VISIT(G,v)
7  u.color = BLACK
8  time = time + 1
9  u.f = time
```

5) Time Complexity Analysis:

Lines 11-13, 2, 3, 5-7 run in $O(1)$ time.

The DFS call at line 1 takes $O(|V|+|E|)$ time.

Since the loop at line 4 iterates lines 5-7 $|V|$ times, lines 4-7 have running time $O(|V|)$. Similiarly lines 8-9 also have running time $O(|V|)$.

Since the CHECK routine visits all nodes at most once over all of its recursive calls, the total running time for line 10 could be $|E| \times$ time spent at each node.

Since setting the color, incrementing count etc. operations run in constant time we have that running time for CHECK(G, u) is $O(|E|)$.

Hence total running time of the algorithm is

$= O(|V|+|E|) + c|V| + O(|V|) + O(|E|) + c\,O(1)$.

$= O(|V|+|E|)$.