Problem Solving HW 16

Alex

June 14, 2017

22.1-3

Use adjacency-list representation

GRAPH-TRANSPOSE(G)

```
1  Let G^T be a new graph with all vertices in G.
2  for each v ∈ G
3        for each u ∈ G.adj[v]
4              add v to G^T.adj[u]
5  return G^T
```

Running Time $O(V + E)$

Use adjacency-matrix representation

GRAPH-TRANSPOSE(G)

```
1  for i = 1 to G.degree − 1
2        for j = 0 to i − 1
3              exchange G[i][j] with G[j][i]
4  return G
```

Running Time $O(V^2)$

22.1-8

Suppose the hash table is implemented by linked list. If all the edges is hashed uniformly then the expected time to decide whether an edge is in the graph is $O(1 + \alpha)$, where α is the time to compute the hash hash function. One big disadvantage is that it takes a lot of time to find all the vertices that are connected with a given vertex. Because you have to go through all the hash table or you need to check all other possible vertices. And like adjacency-list, it takes a lot of time to remove a vertex. We can change Adj$[u]$ to a dynamic array. All the verices in it is sorted. So we can use binary search to find an edge. And we can all decide all the vertices that are adjacent to a given vertex. But it also takes a lot of time to remove or add a vertex.

22.2-3

Because only line 2 and 13 checks the color of a vertex, and both of the statements check whether a vertex is white or not, any color other than white will give the same result. More precisely, a vertex will only enter the

queue once, so after a vertex had already entered the queue, no matter it has come out or not it won't enter the queue again. So there is no need for we to color a vertex black after it dequed.

22.2-4

```
BFS(G,s)
 1  for each vertex u ∈ G.V − {s}
 2        u.color = WHITE
 3        u.d = ∞
 4        u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11        u = DEQUEUE(Q)
12        for v = 0 to G.degree-1
13              if G[u][v] == 1 AND v.color == WHITE
14                    v.color = GRAY
15                    v.d = u.d + 1
16                    v.π = u
17                    ENQUEUE(Q, v)
```

Running Time $O(V^2)$

22.2-5

Since $u.d = \mathrm{d}(s, v)$, the distance between two is independent from the way the graph is presented, so $u.d$ is in dependent from the order in the adjacency-list.
In Figure 22.3, if the adjacency-list of w is $\{x, t\}$ rather than $\{t, x\}$, then x will enter the queue first thus makes xu be an edge in the breadth-first tree.

22.3-6

The theorem 22.10 has already showed that there are only tree edges and back edges in a DFS of graph G. So we only need to show that for an edge (u, v) if (u, v) is encountered first then (u, v) is a tree edge otherwise it is a back edge. (we must assume that u is discovered first, otherwise there is no difference between (u, v) and (v, u)) So if (u, v) is discovered first, v must be white at that time. Because v is pushed into the stack after u, so when we are discovering u, v can't be in the stack, so it is either black or white. But if it is black then (v, u) must be already discovered. So v is white then (u, v) is a tree edge. As mentioned above while discovering (u, v), v can be either white or black, so if it is black, then (v, u) is first discovered, and when (v, u) is discovered, u is gray, so (v, u) is a back edge.

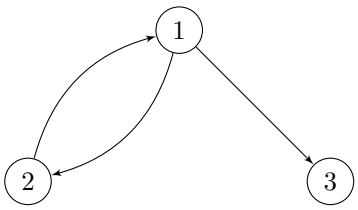
22.3-7

To do this, an extra pointer is needed for all the vertices which points to the next vertices to discover, this pointer is set to the first element of the adjacency-list of each vertex at first.
We only need to change DFS-VISIT.

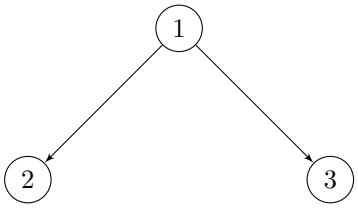
DFS-VISIT(G)

```
 1  time = time + 1
 2  u.d = time
 3  u.color = GRAY
 4  PUSH(S, u)
 5  while S ≠ ∅
 6        t = S.top
 7        if t.p == NIL
 8              time = time + 1
 9              t.f = time
10              t.color = BLACK
11              POP(S)
12        else
13              if t.p.color == WHITE
14                    time = time + 1
15                    t.p.d = time
16                    t.p.color = GRAY
17                    PUSH(S, t.p)
18              Set t.p to the next vertex in the adjacency-list of t.
```

22.3-8



In this case, if we begin the DFS at 1, then $2.d < 3.d$. But the forest it produces is:



Obviously *3* is not *2*'s descendant.

22.3-9

Use the same graph above. $2.f = 3$, $3.d = 5$, $2.f < 3.d$.

22.4-2

We can use dynamic programming to solve this problem. Let D(u, t) to be the number of simple paths between from u to t. Then we have the transfer formula:

$$D(u,t) = \begin{cases} 1 & u = t \\ \sum_{v \in Adj[u]} D(v,t) & u \neq t \end{cases}$$

So it is pretty easy to design a top-town approach to solve this problem. Since there are V subproblems and the number of sub-subproblems equals to the outdgree of the corresponding vertex. So the running time is $O(E)$.

22.4-3

Just simply modify the code for BFS. If we discovered any vertices that is not white, then we can concluded there is a cycle. As for the running time, if a graph contains no cycle, then there are at most $|V|$-*1* edges, so the running time is $O(V)$. And if a graph contains a cycle, then it will explore at most $|V|$-*1* edges so the running time is still $O(V)$.

22.5-5

First compute the strongly connected component of G. This would take $O(V + E)$ of time. Then for each tree we produced, we assign a number for each vertex as the number of vertex in the G^{SCC}, this would take $O(V + E)$ of time. Then we go through all the edges. If the two vertices are in different tree, add them to the $E(G^{SCC})$, this only takes $O(E)$ of time. There is a trick here to avoid repeated edges, for all the edges, we apply radix sort to the vertices pairs. So that all the repeated edges would aggregate, then it is a piece of cake.

22.5-7

First compute the component graph of G. Then we apply topological sort to G^{SCC}. Then we get a linked list of strongly connected component. Then for every two adjacent vertices in the list, check whether all of them are directly connected. If so this graph is semiconnected, otherwise it isn't.

Proof

Obviously a graph G is semiconnected if and only if G^{SCC} is semiconnected. (I want to skip the proof of this, it's too obvious.) Since G^{SCC} is a dag, we can apply topological sort to it. If every two adjacent vertices are directly connected, then it is semiconnected. But if there exist two adjacent vertices that are not directly connected, say u and v, and suppose u is before v. From the property of the topological sort, we know that a vertex can only be directly connected with the vertices after it. So for v it can only be directly connected with the vertices after it, and none of them are connected directly with u, so there isn't a (v, u) path. And for u, it only be connected with vertices after v and also none of them are directly connected to v. So there isn't a (u, v) path. So this graph is not semiconnected, thus we finished the proof.
The time complexity is $O(V + E)$.