Problem 5

> Refs:
>
> None

(1)

Algorithm

$$s(r, T) = \deg(r) \text{ in } E_T$$

Time Complexity

Finding $\deg(v)$ with adjacency list for any vertex v uses $O(1)$ time.

Correctness

For any vertex v such that edge $(r, v) \in E_T$, the sub-tree with root at v is a connected component after the removal of r. And since T is a tree, each of these sub-trees isn't connected to each other without r. Therefore the number of connected components after removing r is the degree of r in E_T.

(2)

Algorithm

$$s(v, G) = 2$$

Time Complexity

Since it's always 2, it takes $O(1)$ time.

Correctness

Because there are no edges from descendants of v to ancestor of v, and G is undirected, all edges between descendants of v are tree edges, and they form a connected component after removing v. All the other vertices are connected to r, and they form the second connected component after removing v.

(3)

Algorithm

$$s(v, G) = |\{\mathrm{up}_T(w_t) \mid \mathrm{up}_T(w_t) = \mathrm{depth}(v) \text{ and } 1 \leq t \leq k\}| + 1$$

Correctness

Because G is undirected, every edge is either a tree edge or a back edge. This means the vertex that $\mathrm{up}_T(w_k)$ implies is either v or an ancestor of v.

If this vertex is v, $\mathrm{up}_T(w_k) = \mathrm{depth(v)}$, and after removing v, w_k and its descendants forms a connected component themselves.

If this vertex is an ancestor of v, $\mathrm{up}_T(w_k) < \mathrm{depth(v)}$, and after removing v, w_k and its descendants joins the connected component of ancestors of v.

Therefore $s(v, G)$ is the number of w_k such that $\mathrm{up}_T(w_k) = \mathrm{depth(v)}$ plus the connected component of ancestors of v.

(4)

Algorithm

1. Choose any vertex as root r. Run DFS from r.
    1. During DFS, maintain the depth of each vertices in the DFS tree with: $v.\mathrm{depth} = v.\pi.\mathrm{depth} + 1$
    2. After visiting every neighboring vertices of v, calculate $\mathrm{up}_T(v)$ with: $\mathrm{up}_T(v) = \min_{w \in W,\, u \in U}(\mathrm{up}_T(w), u.\mathrm{depth})$, where W is the set of all children of v and U is the set of all neighbors of v.
    3. Also calculate $s(v, G)$ with the method in (3) if $v \neq r$.
2. For r, $s(r, G) = \deg(r)$ in DFS tree.

Time Complexity

- Initialization for DFS takes $O(|V|)$ time.
- For each vertex v:

    - Visiting every neighbor vertices v takes $O(|E_v|)$ time. E_v is the set of edges with v at one end.
    - Calculating $s(v, G)$ and $\mathrm{up}_T(v)$ takes $O(|E_v|)$ time.
- And calculating $s(r, G)$ takes $O(|E_r|)$ time (checking all neighboring vertices' predecessor).
- Total time complexity is $O(|V|) + \Sigma_{v \in V}\ O(|E_v|) = O(|V|) + O(2 \cdot |E|) = O(|V| + |E|)$

Correctness

$\mathrm{up}_T(v)$ can be calculated by:

$$\mathrm{up}_T(v) = \min_{w \in W,\ u \in U}(\mathrm{up}_T(w), u.\mathrm{depth})$$

where W is the set of all children of v and U is the set of all neighbors of v. Because $\min_{w \in W}\ \mathrm{up}_T(w)$ includes all neighbors of "descendants of v excluding v", only neighbors of v haven't been considered.

$s(w, G)$ and $\mathrm{up}_T(w)$ are calculated after visiting w. $u.\mathrm{depth}$ is also calculated after visiting all children because u must be either v's ancestor or descendant (no cross edge in undirected graph). Therefore, all needed values for $s(v, G)$ and $\mathrm{up}_T(v, G)$ are available during calculation.

For root r, because there are no cross edges, $s(r, G)$ is number of sub-trees, which is also $\deg(r)$.

Problem 6

> Refs:
>
> None

(1)

Algorithm

Run Prim's algorithm but with some modification:

- Choose the edge with highest weight that connects to a vertex not in the tree yet.
- Initial value of $v.key$ is $-\infty$.
- Update $v.key$ when there is an edge to v with higher weight.

Time complexity

With priority queue implemented with binary max-heap:

- Building heap takes $O(V)$ time.
- Extracting max takes $O(\log V)$ time, and this is called V times.
- Modifying key takes $O(\log V)$ time, and this is called $O(E)$ times.

Total time complexity is $O(V) + V \cdot O(\log V) + O(E) \cdot O(\log V) = O((V + E) \log V)$.
And because G is connected, $V = O(E)$, $O((V + E) \log V) = O(E \log V)$.

Correctness

> **Cut property:**
>
> For any partition that separates vertices into two sets (a "cut" C), let e be the edge with highest weight that connects these two sets. The maximum spanning tree contains e.
>
> **Proof:**
>
> Suppose e isn't in the maximum spanning tree. First add e into the tree, and there should be cycle.
>
> There exists an edge e' that is on this cycle and also on the cut C. By removing e', we get a spanning tree with higher total weight.
>
> Therefore e is the maximum spanning tree

Consider vertices in the current tree and vertices not in to be two sets. The algorithm always adds the highest-weight edge, and by cut property this edge must be in the maximum spanning tree.

And because the algorithm gives a spanning tree, it must be the maximum spanning tree.

(2)

Consider the width of each road to be the weight of that edge. For edges with the same weight, we define their order by order of the smaller vertex first, then the larger vertex.

Suppose that there are multiple paths between s and t.

For any two paths (p_1, p_2), they should together form a cycle in G. WLOG, assume that $w(p_1) = w_1 < w(p_2) = w_2$, and the minimum edge in this cycle is e with $w(e) = w_1$.

Suppose e is in the maximum spanning tree T, removing e should separate the tree into two components. And there should be another edge e' in the cycle that isn't in the tree now, and can connect these two components. By adding e', we have a new spanning tree with larger weight, therefore e must not be in the maximum spanning tree.

By this fact, only the widest path between s and t is still in T. Therefore T qualifies to be G', and edges in T can be E'.

(3)

There is a shortest path from s to u or v that ends at road $e = (u, v) \Rightarrow e$ is downwards critical

WLOG, assume that there is a shortest path p from s to v that ends at e. If $d(e)$ is decreased, p now have a shorter total distance, and it must be a shortest path from s to v.

e is downwards critical $\Rightarrow$ There is a shortest path from s to u or v that ends at e

Assume that road e is downwards critical and there are **no** shortest path from s to either u or v that ends at e.

Suppose the shortest path from s to v is p, and the shortest path from s to v "that ends at (u, v)" is q. Let $\delta = d(q) - d(p)$.

$\delta > 0$ because p is shorter than q. If we decrease $d(e)$ by $\frac{\delta}{2}$, q' (path q with this change) has length:

$$d(q') = d(q) - \frac{\delta}{2} > d(q) - \delta = d(p)$$

Since $d(q') > d(p)$, the shortest path distance from s to v is still the same.

The same argument can be made about u, making e not downwards critical. Therefore the assumption is false. Road (u, v) is downwards critical $\Rightarrow$ there is a shortest path from s to u or v that ends at (u, v).

(4)

Claim

Road $e = (u, v)$ is upwards critical if and only if every shortest path from s to u or v contains e .

Every shortest path from s to u or v contains e $\Rightarrow$ Road e is upwards critical.

WLOG, assume that every shortest path from s to v contains e. Suppose the shortest path from s to v has length r_1, and "without e, the shortest length is r_2".

If $d(e)$ is increased by $\delta > 0$, the new shortest length from s to v is $\min(r_1 + \delta, r_2) > r_1$. Therefore e is upwards critical.

Road e is upwards critical $\Rightarrow$ Every shortest path from s to u or v contains e.

Assume that e is upwards critical and p is a shortest path from s to v without e.

Increasing $d(e)$ by any $\delta > 0$ doesn't change $d(p)$ because p doesn't contain e, and the shortest path distance from s to v is still the same.

The same argument can be made for u. Therefore e isn't upwards critical, and the assumption is false. Road e is upwards critical $\Rightarrow$ Every shortest path from s to u or v contains e.

(5)

Algorithm

1. Run Dijkstra's algorithm from s with some modifications:
    - For each vertex v, instead of storing only one vertex in $v.\pi$, store a list of vertices that all yields shortest path.
    - During `Relax()`:
        - If the path through u to v is as good as $v.d$, add u to $v.\pi$.
        - If the path through u to v is better, empty $v.\pi$ then add u to $v.\pi$.
2. For every vertex v, mark edge (u, v) as **downwards critical** for all $u \in v.\pi$. If $v.\pi$ contains only one vertex, also mark that edge as **upwards critical**.

Time complexity

- In step 1, the modification adds $O(1)$ time for each edge. Therefore time complexity is $O((V + E) \log V) + E \cdot O(1) = O(E \log V)$ with priority queue implemented with binary heap.
- In step 2, each vertex v takes $O(E_v)$ time where E_v is the is the set of edges with v at one end. Total time complexity here is $\sum_{v \in V} O(E_v) = O(E)$.
- Total time complexity of the algorithm is $O(E \log V) + O(E) = O(E \log V)$.

Correctness

$u \in v.\pi \iff$ There is a shortest path from s to v containing (u, v) (by Dijkstra's algorithm's correctness). Therefore, step 2 marks every and only the correct edges.

(6)

Algorithm

1. Define the new weight of edge (u, v) to be $d'(u, v) = d(u, v) - \frac{k(u)+k(v)}{2K}$.

2. Run Bellman-Ford algorithm from any vertices with the new weight d'.
3. If it detects a negative cycle, there is an exciting cycle. Otherwise, there is no exciting cycle.

Time complexity

- Re-weighting every edge takes $O(E)$ time.
- Bellman-Ford algorithm takes $O(VE)$ time.
- Total time complexity is $O(E) + O(VE) = O(VE)$.

Correctness

Observe the definition of an "exciting cycle":

$$
\begin{aligned}
\frac{\sum_{i=1}^{n} k(v_i)}{\sum_{i=1}^{n} d(v_i, v_{i+1})} &> K \\
K \sum_{i=1}^{n} d(v_i, v_{i+1}) &< \sum_{i=1}^{n} k(v_i) \\
\sum_{i=1}^{n} d(v_i, v_{i+1}) - \frac{1}{K} \sum_{i=1}^{n} k(v_i) &< 0 \\
\sum_{i=1}^{n} d(v_i, v_{i+1}) - \frac{1}{2K} \sum_{i=1}^{n} [k(v_i) + k(v_i)] &< 0 \\
\sum_{i=1}^{n} d(v_i, v_{i+1}) - \sum_{i=1}^{n} \frac{k(v_i) + k(v_{i+1})}{2K} &< 0 \\
\sum_{i=1}^{n} [d(v_i, v_{i+1}) - \frac{k(v_i) + k(v_{i+1})}{2K}] &< 0
\end{aligned}
$$

If we define the new weight of edge (u, v) to be $d'(u, v) = d(u, v) - \frac{k(u)+k(v)}{2K}$, the detection of exciting cycles becomes the detection of negative cycles with this new weight. And Bellman-Ford algorithm detects negative cycles.