1. Study Group

Jialiang Tang (myself), SID: 3039758308

Buji Xu, SID: 3039687809

2. We only need to count the sum of subtree of nodes at a distance of K away from a node. To do it, we can adopt DFS 2 times.

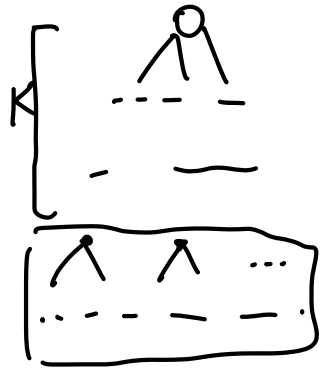
Algorithm Description: Firstly, run DFS on the tree and record the pre-post value of every node for computation of local subtree scale ((post - pre) // 2 + 1 is exactly the number of descendants of the node including itself, noted as "subtree scale")

Then, run DFS again. This time we use a stack to store the current path. When we visit a node, we can locate the ancestor node K away from it through indexing it in the stack K positions before current node (if not exist, do nothing), and add the subtree scale of current node to S[v] (Initially, S[i] are all 0. v is the ancestor node mentioned above).

After running DFS 2 times, we have correct S array for each node.

Proof of correctness: It's easy to see

$$S[v] = \sum_{dis(v,i)=K} |subtree(i)|$$

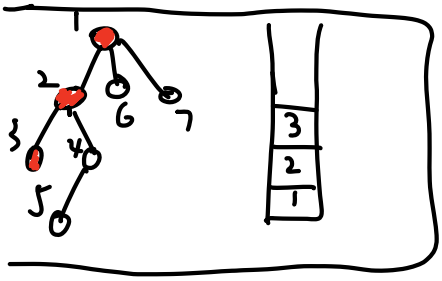


|subtree(i)| is card of subtree whose root node is i. By the DFS Alg, we have

$$|subtree(i)| = |post(i) - pre(i)| // 2 + 1$$

Also, when using a stack, we can store the DFS path, in which the nodes are recorded

in depth order in the tree, and they are all ancestor or descendant of each other. So We can query the ancestor at distance of K away from any node v by using Stack[v-K] (if v-K<0, then v has no ancestor K away from it).



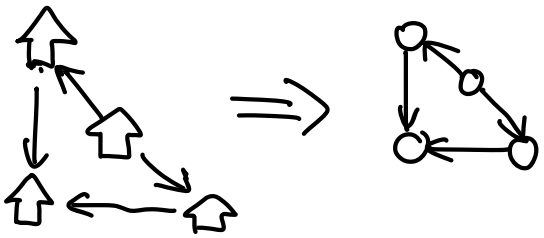
Based on these facts, correctness has been proved.

Runtime Analysis: We only need to run DFS 2 times. Each takes $O(|V|+|E|) = O(|V|)$. Computing subtree(i) takes $O(|V|)$ and make addition also takes $O(|V|)$.

To sum, it takes $O(|V|)$ overall.

Note: If using dynamic programming or recurrence, the normal algorithm will both take $O(|V| \cdot K)$. But runtime of this 2-DFS algorithm will be $O(|V|)$, even independent of K!

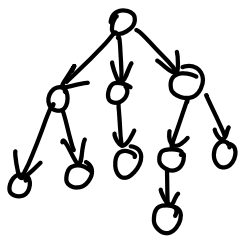
3. (a) See igloos as nodes and one-way roads as directed edge between nodes. Then we get a directed graph.



Description of Algorithm: Run SCC algorithm on the graph but count the number of nodes in each scc. (Using pre, post to compute)
Output the sum of nodes in scc which has more than 1 node.

Runtime Analysis: Assume number of igloos is n, number of path is m. So the runtime is $O(n+m)$.
~~Constructing~~ a graph needs linear time

(b) See species of pokemon as nodes and there is an edge between 2 species if one directly descends from the other. Then we get a tree.

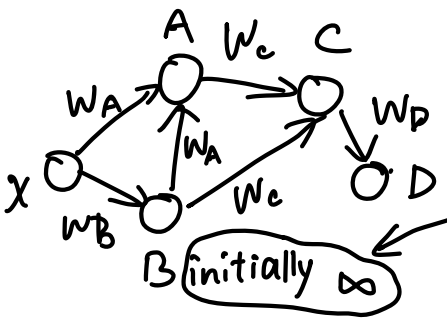


Description of Algorithm: Run DFS on the tree, record the pre, post value of every node. Store these $O(p)$ data. The query program is: get pre[a], post[a], pre[b], post[b]. If pre[a] < pre[b] < post[b] < post[a] then output option (2); If pre[b] < pre[a] < post[a] < post[b], then output option (1); Otherwise, output option (3).

Runtime Analysis: Query needs $O(1)$ time since we have taken $O(p)$ to preprocess the data. (Constructing and DFS both take $O(p)$ time)

(c) See each box as a node. There is a directed edge from node A to node B iff A has a size between 0.85 to 1 times that of B, and the weight of an edge is the w value of B.

Description of Algorithm: Run DFS Algorithm (with no repeat visit in same path) on the directed graph while source node is smallest box x.



Each time we visit a node, we add the w value of edge going into it to the total sum (Initially sum is W_X). Check whether it makes sum smaller (if yes, renew the smallest cost and record the node)

When backtracking, minus weight of the node from total sum instead. After DFS, tracking upwards until visiting x from the recorded node.

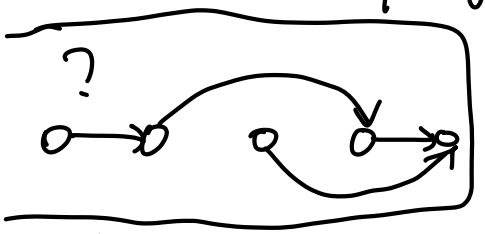
Then we get a lightest sequence of boxes (exactly the boxes in the path)

Runtime Analysis: Assume we have n boxes and m fit relations (edges), then running DFS takes $O(n+m)$, computing and comparing in the process of DFS takes $O(n+m)$.

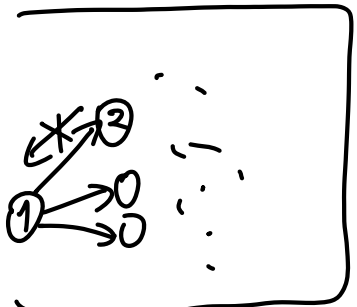
Overall, it takes $O(n+m)$ time.

4. Proof: "if": If there is a directed path visiting all vertices of G, then for any vertices A and B in G, they are all on the path. So G is semiconnected.

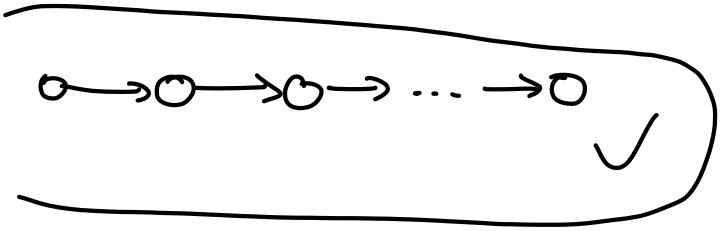
"only if": Topologically sort G, we get a sequence of vertices in topology order. Since G is semiconnected,



the vertice with 0 in-degree is unique in initial G, otherwise there's no path between 2 vertices with 0 in-degree. So the first vertex sorted is just the vertex (On the other hand, since G is acyclic, there must be a 0 in-degree vertex). After removing the first vertex, there are also only 1 vertex with 0 in-degree for the same reason and also there's no back edge from these vertices to the first vertex (otherwise there's a cycle).



After we sort a vertex, it is guaranteed to have a direct edge with the previous vertex. And so on, the vertices in topo order all have forward edge with its neighbor behind in the sequence. So the sequence exports a path connecting every vertex in G.

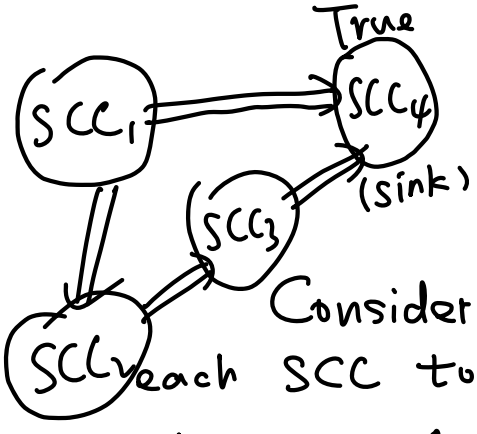


5. (a) Proof: If G has SCC containing x and $\bar{x}$, then there's a path from x to $\bar{x}$ while also one path from $\bar{x}$ to x. By construction rules, it exists

$$x \Rightarrow a \Rightarrow \cdots \Rightarrow \bar{x} \quad \text{and} \quad \bar{x} \Rightarrow b \Rightarrow \cdots \Rightarrow x$$

so $x \Rightarrow \bar{x}$ and $\bar{x} \Rightarrow x$, $\bar{x} \Leftrightarrow x$, which can not hold for any assignment of x.

(b) Proof: Do as hinted. Pick a sink SCC of G_I and assign the variables so that all literals in sink SCC are True.



But actually, let's do a more generative proof:

Considering assignment of all literals. We shrink each SCC to a point, getting a DAG. Then topo sort the DAG. Assume the topo order of SCC containing x is t_x. We do assignments as below:

(*) { ① if $t_x < t_{\bar{x}}$, assign x = False
② if $t_x > t_{\bar{x}}$, assign x = True

Situation that $t_x = t_{\bar{x}}$ won't exist.

Then we prove the assignment will not break any implications:

Firstly, it won't break imp between x and $\bar{x}$. It's obvious since our assignment rule guarantee the fact that only "False $\Rightarrow$ True" will happen to imp of x and $\bar{x}$.

Secondly, assume we have $x \Rightarrow y$. Assume that it happens that x = True and y = False. We have:

$t_{\bar{x}} < t_x$ (x = True),

$t_y < t_{\bar{y}}$ (y = False),

$t_x \leq t_y$ ($x \Rightarrow y$, and either x and y are in same SCC or x is visited earlier than y according to topo sort)

So we have $t_{\bar{x}} < t_x \leq t_y < t_{\bar{y}}$. Noticing that $\bar{y} \Rightarrow \bar{x}$ also holds, we have $t_{\bar{y}} < t_{\bar{x}}$. Contradiction happens!

So the assumption that x = True and y = False fails, which means no implication will be broken. Our assignment rule will definitely contribute to a legal assignment!

(c) By (a) and (b), we export the algorithm as below:

① Construct G_I of 2-SAT instance

② Run SCC on G_I (Tarjan Algorithm version of SCC)

③ Shrink each SCC to one node and run Topological Sort on the DAG (Making use of Tarjan Algorithm)

④ For each x, check whether it has $\bar{x}$ in same SCC, if yes, return "Unsatisfable".

⑤ Make assignment according to topological sort order

by (*)

⑥ Output the assignment.

Runtime Analysis: Tarjan Algorithm makes use of stack and DFS, taking $O(2n+2m)$ time. The topo sort and assignment also takes $O(2n+2m)$. So the overall runtime is $O(2n+2m)$.

Reference:

en.Wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm