22 (100 PTS.) Bane studies graph theory.

In the Dark Knight Rises, Bane traps the world's dumbest police force in Gotham's underground tunnel system by waiting for every officer to march into the tunnels, and then sealing the exits. The tunnel system consists of the straight tunnels (represented by edges) and the connection points where multiple tunnels intersect and are connected together (represented by vertices).

However, a deleted scene shows Bane's true genius. Bane couldn't just seal the entrances because then if Batman digs through a single entrance, all the cops will get out. He could just blow up every single tunnel but that's too expensive. After all he's not Batman!

Bane realizes he can save some money and selectively destroy tunnels(e)/intersections(v) resulting in two or more disjoint tunnel sub-systems. Therefore, if the entrance to one subsystem is breached, only the cops in that sub-system will get out. We call these tunnels/intersections *critical*.

You can assume the tunnel system is initially connected.

Solution:

We model Gotham's tunnel network as a graph where the intersections are represented as vertices and the tunnels are represented as edges.

22.A. (30 PTS.) Design a linear time algorithm to check if a tunnel is *critical*. In other words check if the removal of this tunnel results in two disjoint sub-systems.

Solution:

To see whether $e = (u, v)$ is a cut-edge, we can remove e from G and use any basic search algorithm to determine whether u is still connected to v. If it is, then e is not a cut-edge, otherwise e is a cut-edge.

It takes $O(n + m)$ time to perform the basic search starting at u. If we perform this once for each of the m edges, the algorithm takes $O(m^2)$ time.

22.B. (50 PTS.) Design a linear-time algorithm which identifies every critical tunnel.

Solution:

This critical tunnel can be thought of as a cut-edge and hence, we need to find all the cut-edges in the graph.

Empirically, we can determine a few things about cut-edges.

1. If $e = (u, v)$ is a cut-edge, then e must belong to any spanning tree of G.
2. Eliminating e would create two disjoint sub-graphs, one containing u and one containing v.
3. e is only a cut-edge if there exists no back edge from the subtree containing v to the subtree containing u

We can tell if there is a back-edge by recording the time DFS first reaches a particular vertex as well as the time of the vertices it is connected to. Let T be a DFS tree. We denote the pre-visit time of a vertex u in T by $pre(u)$. For each node u define:

$$low(u) = \min \begin{cases} pre(u) \\ pre(w) \text{ where } (v, w) \text{ is a back edge for } v \in T_u. \end{cases}$$

To clarify, we remark that $low(u)$ will be $\geq pre(u)$ if there is not back edge going out of T_u. The following pre-processing procedure is a modification of DFS that for each u, records the pre-visit time $pre(u)$, the predecessor vertex $pred(u)$, and the value $low(u)$.

```
LowDFS($G$):
    for all $u \in V$
        $edges \leftarrow \oslash$
        $pred(u) \leftarrow NULL$
    pick an arbitrary vertex $u \in V$
    LowDFSAUX($G, u$)
```

```
LowDFSAUX($G, u$):
    mark $u$ as a visited vertex
    $time \leftarrow time + 1$
    $pre(u) \leftarrow time$
    $low(u) \leftarrow pre(u)$
    for each vertex $v$ in $Adj(u)$
        if $v$ is not marked as visited
            $pred(v) \leftarrow u$
            LowDFSAUX($G, v$)
            $low(u) \leftarrow minlow(u), low(v))$
            if ($low(v) \geq pre(u)$)
                $edges = edges \cup (u, v)$
        elseif $v \neq pred(v)$
            $low(u) \leftarrow min(low(u), pre(v))$
```

The preprocessing procedure is a DFS with predecessor and pre-visit time, with additional constant amount of work at every step in order to calculate $low(u)$. Thus the running time of the algorithm is exactly the same as DFS: $O(m + n)$.

22.C. (20 PTS.) Bane theorizes it might be more cost effective to destroy intersection points rather than entire tunnels (this would fragment the system more effectively as well). Develop a linear time algorithm that will identify every critical intersection.

Solution:

This problem is referred to as finding articulation points. We surmise that if vertex u is a articulation point, then removing the vertex will break the graph into two unconnected graphs. This means that if we do a DFS on G there will be no back edges from a vertex in the subtree with root u to the rest of the graph.

The idea here is similar to that in the previous part. The major difference is that the vertex can be a root node which separates both halves of the DFS tree. Additionally, the vertex must have children. If it doesn't, removing it won't segment the graph. We make the following changes to our algorithm from above:

```
LowDFS($G$):
  for all $u \in V$
      $ap(u) \leftarrow NULL$
      $pred(u) \leftarrow NULL$
  pick an arbitrary vertex $u \in V$
  LowDFSAUX($G, u$)
```

```
LowDFSAUX($G, u$):
  mark $u$ as a visited vertex
  $time \leftarrow time + 1$
  $pre(u) \leftarrow time$
  $low(u) \leftarrow pre(u)$
  $children = 0$
  for each vertex $v$ in $Adj(u)$
      if $v$ is not marked as visited
          $children{+}{+}$
          $pred(v) \leftarrow u$
          LowDFSAUX($G, v$)
          $low(u) \leftarrow minlow(u), low(v)$
          if $(pred[u] = NULL \;\&\&\; children > 1)$
              ap(u) = true
          if $(pred[u] \neq NULL \;\&\&\; low(v) \geq pre(u))$
              ap(u) = true
      elseif $v \neq pred(v)$
          $low(u) \leftarrow min(low(u), pre(v))$
```

Using similar reasoning as in the previous part, the total running time is $\boldsymbol{O(m+n)}$, or linear in the size of the input.

23 (100 PTS.) Poke-vertices: Gotta visit them all

Let G be a directed acyclic graph with a unique source s and unique sink t.

23.A. (20 PTS.) A Hamiltonian path in G is a directed path that contains every vertex in G. Describe an algorithm to determine whether G has a Hamiltonian path.

Solution:

Perform a topological sort of the given DAG G in $O(n+m)$ time. Next, verify that any two consecutive vertices v_i, v_{i+1} in this ordering are connected by an edge (v_i, v_{i+1}). If so, then this is the required Hamiltonian path. Otherwise, there is no Hamiltonian path. The running time of this algorithm is $O(n+m)$.

Correctness. If there is an Hamiltonian path, then there is only one topological ordering of the DAG, which is the path itself, and that is what the algorithm computes.

Lemma 8.1. *A DAG G has a Hamiltonian path $\iff$ it has only one topological ordering.*

Proof: If a DAG G has a Hamiltonian path then it defines a topological ordering, and clearly it is unique.

The other direction is more interesting. Consider the algorithm that computes the Topological ordering of G, by repeatedly computing a source vertex s in G, output it, and removing s from G. (Here a source is a vertex with no incoming edge.) Clearly, if s is a single source, then it must be the first vertex is any topological ordering of G. If there are two vertices s_1, s_2 that are sources in the same iteration, then the graph can not be Hamiltonian since both s_1 and s_2 needs to be the first vertices on the Hamiltonian path. Thus, if the topological ordering is not unique, then at some iteration this algorithm would reach a graph with two or more sources, and this would imply that the original DAG is not Hamiltonian. ■

23.B. (30 PTS.) Suppose the edges of G have weights. Describe an efficient algorithm to find the path from s to t with the maximum **average** weight. Here, for a path π with k edges, and total weight of the edges being w, the average weight is w/k.

Solution:

Let $f(v,i)$ be the weight of the maximum path from v to t. Let $s = v_1, \ldots, v_n = t$ be the topological ordering of G (if s and t are not first/last in this ordering, we can delete the vertices that are not in between them in the graph).

We set $f(t,0) = 0$, and $f(u,0) = -\infty$, for all $u \in V \setminus \{t\}$. We now define

$$f(v_i, k) = \max_{(v_i,v_j)\in \mathsf{E}(G)} \bigl(w(v_i \to v_j) + f(v_j, k-1) \bigr).$$

By handling the vertices in the reverse ordering of their topological ordering, $f(\cdot, k)$ can be computed in $O(m)$ time from $f(\cdot, k)$. We repeat this for $k = 1, \ldots, n-1$). The desired quantity is

$$\max_{k=1}^{n-1} f(s,k)/k.$$

Using our regular machinery, we can turn this now into a dynamic program. The running time of the algorithm is $O(n(n+m))$.

23.C. (20 PTS.) Suppose some of the vertices of G are designated as ***special***. A special vertex v has a positive weight $w(v) > 0$ associated with it. A non-special vertex has weight zero. Describe an algorithm, as fast as possible, that computes the maximum weight path from s to t.

Solution:

Boring. Let $s = v_1, \ldots, v_n = t$ be the topological ordering of the vertices of G. Define $g(v_n) = w(t)$, and more generally,

$$g(v_i) = \max_{(v_i,v_j)\in \mathsf{E}(G)} \bigl(w(v_i) + g(v_j)\bigr).$$

By handling the vertices in the reverse ordering of their topological ordering, $g(\cdot)$ can be computed in $O(m)$ time. The output of the algorithm is $g(s)$.

Using our regular machinery, we can turn this now into a dynamic program. The running time of the algorithm is $O(n+m)$.

23.D. (30 PTS.) Describe an algorithm that returns if the number of different paths from s to t in G is odd or even. To earn any points, your algorithm should not compute the number of such paths (because the number of such paths is potentially exponential, and working with such big numbers is expensive).

Solution:

Let start with the basic inefficient algorithm, and then modify it to be the desired solution. So, let $s = v_1, \ldots, v_n = t$ be the topological ordering of G. Let $h(v)$ be the number of different paths from v to t in G.

We set $h(t) = 1$. We now define

$$h(v_i) = \sum_{(v_i,v_j)\in \mathsf{E}(G)} h(v_j).$$

It is easy to verify that the above indeed computes the values of $h(\cdot)$ correctly. In particular, $h(s)$ is indeed the number of disjoint paths from s to t. The problem is that if we implement this algorithm, in the same way as done above, its running time is $O(n(n+m))$, since the numbers involved are of size $2^{\Omega(n)}$, in the worst case. Such numbers requires n bits to handle, and adding two of them takes $\Theta(n)$ time.

Luckily, what we want to compute is $h(x) \bmod 2$. Furthermore, we have the super hyper duper bigly useful identity, that for any integers x, y, we have

$$(x+y) \bmod 2 = \Bigl((x \bmod 2) + (y \bmod 2)\Bigr) \bmod 2$$

Thus, defining $\widehat{h}(v) = h(v) \bmod 2$, the above recurrence becomes:

$$\widehat{h}(v_i) = h(v_i) \bmod 2 = \Bigl(\sum_{(v_i,v_j)\in \mathsf{E}(G)} h(v_j)\Bigr) \bmod 2 = \sum_{(v_i,v_j)\in \mathsf{E}(G)} \widehat{h}(v_j) \bmod 2.$$

Since $\widehat{h}(\cdot)$ is a single bit, it follows that we can compute $\widehat{h}(s)$, in $O(n+m)$ time, using the same approach as described above.

24 (100 PTS.) The revolution will not be televised, it will be a question on the homework.

There are n rebels that are currently planning, well, a "party". A rebel v can send messages to only one other rebel, designated as contact(v). Here, if a rebel is given a message, they will send it to their contact, and it would be propagated in this fashion to everyone reachable.

24.A. (20 PTS.) Describe an algorithm, as fast as possible, that computes a rebel v, such that if you send a message to v, the message gets propagated to all the rebels. If there is not such vertex v, the algorithm should output "no solution".

Solution:

The graph is a collection of cycles, and paths that might go into the cycles.

So, we compute for each node their incoming degree. If there is a single vertex with no incoming edge, then we check if it can reach all vertices by doing a **DFS** from it. If there are two or more such vertices, then there is no way to send a message to everyone using a single source.

The remaining possibility is that all the vertices have exactly one incoming edge. Namely, the graph is a union of cycles. We compute the number of cycles by doing DFS. If the number of cycles is one, then any vertex in this graph can send message to everyone else. Otherwise, there is no such vertex.

The overall running time of this algorithm is $O(n)$.

24.B. (20 PTS.) Describe an algorithm, as fast as possible, that computes the minimal number of rebels that needs to be sent directly a message, before the message can be propagated to all the rebels.

Solution:

Use the above algorithm to decide if one can send the message to a single vertex. Otherwise, every vertex that has no incoming edge must be a source. We do DFS from these vertices, to compute all the vertices that would get their messages from these sources. All remaining vertices in the graph are cycles. We compute these cycles using **DFS**, sending a message to a single rebel inside each cycle.

The running time is of this algorithm is $O(n)$.

24.C. (30 PTS.) A more realistic situation is that a rebel v can contact a group of rebels $R(v)$. Describe an algorithm, as fast as possible, that decides if there is a rebel such that if you send them a message, they can propagate it to everyone. Let $m = \sum_{v=1}^{n} |R(v)|$ be the total size of the input. You can safely assume in analyzing the input that $m \geq n$. Provide a running time in terms of m.

Solution:

Compute the strong connected components of the graph, and the meta graph $\mathsf{G}^{\mathrm{SCC}}$ which is a DAG. If this DAG has a single source (i.e., only one vertex with no incoming edges), then any vertex in the SCC of this source, is a rebel that can broadcast to everyone. Clearly, this all can be done in $O(n + m)$ time.

24.D. (30 PTS.) (Hard.) We are back to the setting where each rebel can send only a single message to their contact. Describe an algorithm, as fast as possible, that reassigns the minimal number of contacts, so that a message can be sent to *any* single rebel, and it will be propagated to all rebels. Here, a (single) *reassignment* is assigning a rebel a different contact person than their current one. Prove the correctness of your algorithm. (Here, an algorithm without formal proof of correctness is worth no points.)

Solution:

What is the question? Our purpose here is to create one big happy cycle, by reassigning edges.

The insight. Look on the cycle after the reassignment. If we delete every reassigned edge, then what remains is a collection of paths.

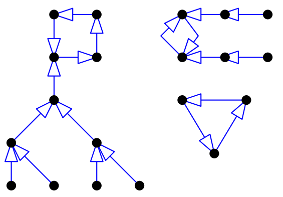
What is *really* the question? How to remove a minimal number of edges, so that the graph becomes a collection of paths.

Definitions. A ***head*** of path is the vertex with no outgoing edges, and the ***tail*** is the vertex with no incoming edges.

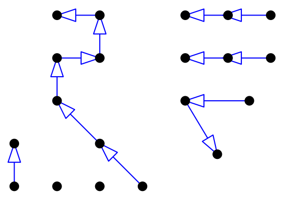
The algorithm.

(A) If the graph is a single cycle, then there is nothing to do.
(B) The algorithm scans the graph, and builds for every vertex a list of its incoming edges.
(C) The algorithm identifies all the cycles in the graph. This can be done by computing the SCC of the graph. Alternatively, copy the graph, and repeatedly delete the vertices that have no incoming edges – what remains are the cycles. Carefully implemented this can be done in $O(n)$ time. Mark all the edges that belong to cycles as ***cycle edges***.
(D) For every vertex v that has $k > 1$ incoming edges, we delete a cycle edge incoming into v, if such an edge exists.
(E) For every remaining cycle, the algorithm removes an arbitrary edge from it.
(F) For every vertex v that has $k > 1$ incoming edges, we delete arbitrarily $k - 1$ of its incoming edges.
(G) The remaining graph is now a collection of, say, t paths (see below for reasoning). The algorithm turns it into a single cycle by doing t assignments. The algorithm connects the head vertex of the ith path into the tail vertex of $(i+1)$th path, for $i = 1, \dots, t-1$. Finally, the algorithm connects the head of the tth path to the tail of the first path.

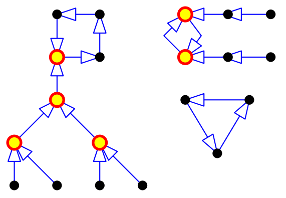
Discussion. Consider the following example for a possible input:



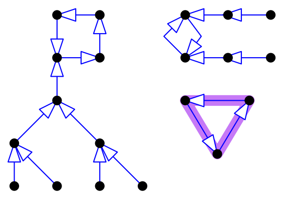
In our case, after we delete all the reassigned edges in the optimal solution, the remaining graph might look like:



A ***bad*** vertex is a vertex that has strictly more than one incoming edge into it. In the above example, the bad vertices are:



A cycle is ***pure*** if it contains no bad vertices. In our running example, we have only one pure cycle:



Step (D) deletes a least one edge from any non-pure cycle.

Step (E) deletes an arbitrary edge from every pure cycle.

Thus, in the beginning of (F), what remains is a collection of reversed trees, as there are no longer any cycles.

After (F) is done, all vertices have zero or one incoming edge. Namely, the graph is a collection of cycles and paths. But there are no cycles, as the algorithm already broke all of them in earlier steps.

Clearly, the running time of this algorithm is $O(n)$, and it clearly generates a single cycle.

Lemma 8.2. *The above algorithm performs the minimum number of reassignments.*

Proof: Every bad vertex, in the original graph, with incoming degree $k > 1$, must have $k - 1$ of its incoming edges reassigned in the optimal solution (because of the insight that if we remove the reassigned edges, we remain with a collection of paths). Also, we must reassign at least one edge from each pure cycle. Importantly, a pure cycle can not contain a bad vertex, so these two quantities are disjoint.

Adding these two quantities together, we get a lower bound α on the number of reassignments needed. However, the above algorithm uses exactly α reassignments, thus establishing the claim. ∎