1. Suppose you are given a directed graph $G = (V, E)$, two vertices s and t, a capacity function $c : E \to \mathbb{R}^+$, and a second function $f : E \to \mathbb{R}$. Describe and analyze an algorithm to determine whether f is a maximum (s, t)-flow in G. *[Hint: Don't make any "obvious" assumptions!]*

   Solution: Verifying that f is a maximum flow requires four steps:

   - Check that $f(u \to v) \geq 0$ for every edge $u \to v$.
   - Check that $f(u \to v) \leq c(u \to v)$ for every edge $u \to v$.
   - Check that $\sum_u f(u \to v) = \sum_w f(v \to w)$ for every node v except s and t.
   - Check that there is no path from s to t in the residual graph G_f.

   The first three steps can be done in $O(V + E)$ time by brute force. In the last step, we need $O(V + E)$ time to build G_f, plus $O(V + E)$ time to perform a whatever-first search starting at s. Thus, the total running time is $\boldsymbol{O(V + E)}$. ■

   Rubric: 10 points = 2 for each step + 2 for time analysis. ($O(E)$ time is fine, because flow graphs are always connected.)

2. Suppose you are given a flow network G with ***integer*** edge capacities and an ***integer*** maximum flow f^* in G. Describe algorithms for the following operations:

   (a) INCREMENT(e): Increase the capacity of edge e by 1 and update the maximum flow.

   Solution: To INCREMENT the edge e, we first increase the capacity $c(e)$ by 1, and then perform one iteration of the Ford-Fulkerson augmenting path algorithm: Build the residual graph G_f, look for a path from s to t in G_f, and if such a path is found, push 1 unit of flow along it.

   To prove the algorithm is correct, there are two cases to consider.

   - Suppose e crosses every minimum cut. Then increasing $c(e)$ increases the minimum cut cost by 1, and thus, by the maxflow-mincut theorem, increases the maximum flow value by 1. Because all the capacities are integers, a successful iteration of Ford-Fulkerson pushes at least 1 unit of flow through the graph. Thus, at most one iteration is required to restore a maximum flow.
   - Otherwise, increasing $c(e)$ does not change the minimum cut cost, and thus does not change the maximum flow value, so f^* is still a maximum flow.

   The algorithm runs in $O(E)$ ***time***. ■

   Rubric: 5 points = 2 for algorithm + 2 for proof + 1 for time analysis. This is neither the only correct algorithm nor the only proof of correctness for this algorithm.

(b) DECREMENT(e): Decrease the capacity of edge e by 1 and update the maximum flow.

Solution: Suppose we are asked to DECREMENT the edge $u \to v$. Assume $f^*(u \to v) = c(u \to v)$, since otherwise, we can simply decrease $c(u \to v)$ by 1 and return. We also assume that $c(u \to v) > 0$, since otherwise, decreasing $c(u \to v)$ is impossible.

First, we send one unit of flow backward through $u \to v$, either along a path from t to s or along a cycle in in the residual graph G_{f^*}. Temporarily add an edge $s \to t$ to the residual graph G_{f^*}, find a path from u to v in this new larger graph, push one unit of flow along the corresponding edges of G, and finally decrease $f(u \to v)$ by 1.

Now we decrease $c(u \to v)$ by 1. The current flow f is still *feasible*, but it may not be a *maximum* flow. To restore a maximum flow, we run one iteration of Ford-Fulkerson, to push at most one more unit of flow from s to t.

To prove the algorithm is correct, we need two prove two claims:

- ***There is a path from u to v in the residual graph plus $s \to t$.*** Temporarily add an edge $t \to s$ to G with $f^*(t \to s) = |f^*|$. Now follow one unit of flow out of v through an arbitrary sequence of directed edges, traversing each edge $x \to y$ at most $f^*(x \to y)$ times. Because flow is conserved at every vertex (including s and t), this walk must eventually reach u. Reversing this walk gives us a walk in the residual graph from u to v. Any walk from u to v contains a path from u to v. (In fact, we can just use the walk directly.)
- ***Decrementing $c(u \to v)$ decreases the maximum flow value by at most 1.*** We can follow the proof from part (a). If $u \to v$ crosses some minimum cut, then decrementing $c(u \to v)$ decreases the cost of the minimum cut by 1, and therefore decreases the value of the maximum flow by 1. Otherwise, the minimum cut cost does not change, so the maximum flow value also does not change.

The algorithm runs in ***$O(E)$ time***. ■

Rubric: 5 points = 2 for algorithm + 2 for proof + 1 for time analysis. This is neither the only correct algorithm nor the only proof of correctness for this algorithm.

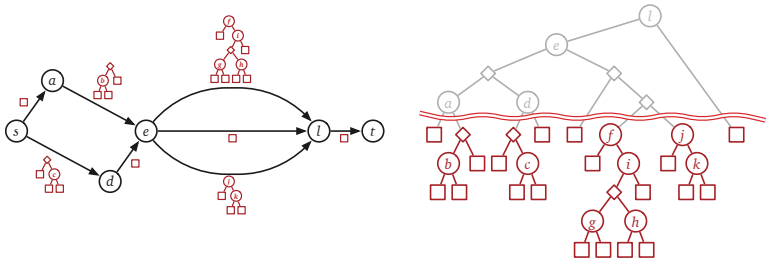
3. (a) Suppose you are given a directed graph G with two special vertices s and t. Describe and analyze an algorithm that either builds a decomposition tree for G or correctly reports that G is not (s,t)-series-parallel. *[Hint: Build the tree from the bottom up.]*

Solution: As suggested by the hint, the algorithm builds a decomposition tree for G upward from the leaves, through a series of reductions, defined as follows:

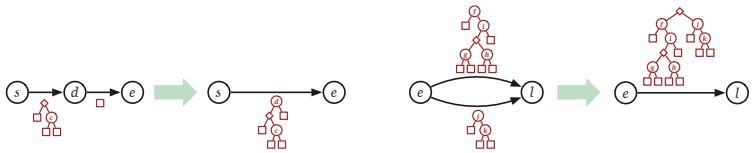
- A *series reduction* replaces two edges $u \to v$ and $v \to w$, where no other edge is incident to v, with a single edge $u \to w$.
- A *parallel reduction* merges any pair of edges with the same tail and the same head into a single edge.

Unless G has only one edge, at least one reduction is possible. (**Proof:** The definition of series-parallel implies a decomposition tree T for G; consider the deepest node in that tree that is not a leaf.) Because each reduction decreases the number of edges, the algorithm terminates after exactly $E-1$ reductions.

In the middle of the algorithm, each edge e of the remaining graph corresponds to a series-parallel subgraph of the original input graph G, which we denote $G(e)$. Each edge e stores a points to a decomposition tree $T(e)$ of $G(e)$. At the start of the algorithm, each tree $T(e)$ is just a single leaf; at the end of the algorithm, when only one edge e remains, $T(e)$ is a decomposition tree for the entire graph G.



When we perform a reduction from two edges $e^\flat$ and $e^\sharp$ to a single edge e, we create a new decomposition-tree node and direct its child pointers to $T(e^\flat)$ and $T(e^\sharp)$; this new node become the root of $T(e)$.



Finally, we need to describe how to *find* the reductions. We can find series reductions in $O(E)$ time, using a brute-force count of the edges entering and leaving every vertex. Parallel reductions are slightly trickier. In the preprocessing phase, we can sort the edges in every adjacency list by the index of the target vertex, in $O(E \log V)$ time. Whenever we perform a series reduction, we insert the new edge into the adjacency list so that this sorted order is maintained, in $O(V)$ time. By keeping the adjacency lists sorted, we can find parallel reductions in $O(E)$ time using a brute-force scan.

Each iteration of the algorithm requires $O(V+E) = O(E)$ time, and the algorithm terminates after $E-1$ iterations. Thus, the entire algorithm runs in $\boldsymbol{O(E^2)}$ ***time***.

This is enough for full credit.

With a bit more effort, we can reduce the running time to $O(E)$ as follows.

First we need to be more careful about the graph data structure. I assume that G is initially presented in a standard adjacency list. In a preprocessing phase, we apply several standard modifications to the data structure:

- We create a second edge list for each vertex v, storing all edges *entering* v. Thus, each directed edge is stored twice in the data structure.
- We make the edge lists at each vertex *doubly*-linked.
- Within each edge record, we store the indices of its endpoints, a pointer to the other record for the same edge, and pointers to the next and previous edge with the same head/tail.

We can perform these modifications in $O(V + E)$ time by careful brute force. With these modifications in place, we can insert any edge or delete any edge (given a pointer to either of its edge records) in $O(1)$ time.

Next we need a faster method to identify series and (especially) parallel reductions. Our modified algorithm maintains a bag of vertices, initially containing every vertex. (The bag is an arbitrary data structure supporting insertions and deletions in $O(1)$ time, for example, a stack or a queue.) At each iteration, we remove an arbitrary vertex from the bag and process it as follows ***in $O(1)$ time***:

```
PROCESS(v):
  if v has already been deleted by a serial reduction
      ⟨⟨do nothing⟩⟩
  else if v = s or v = t
      ⟨⟨do nothing⟩⟩
  else if deg_in(v) = 1 and deg_out(v) = 1
      perform a series reduction at v
      put the endpoints of the new edge into the bag
  else
      choose three arbitrary edges incident to v
      if any two of these edges are parallel
          perform a parallel reduction on those two edges
          put the endpoints of the new edge into the bag
```

Notice that the PROCESS algorithm may not actually perform a reduction. Each reduction inserts two new vertices into the bag.

The overall reduction algorithm now looks like this:

```
DECOMPOSITIONTREE(G):
  preprocess the graph as described above
  for every vertex v
      put v into the bag
  while the bag is non-empty
      take an arbitrary vertex v out of the bag
      PROCESS(v)
  perform any remaining parallel reductions
  return the resulting decomposition tree
```

Before arguing correctness, let's consider the running time. Altogether we perform at most $V + 2E$ insertions into the bag: V in the initial for loop, plus two for each reduction inside PROCESS. It follows that the while loop ends after at most

$V + 2E = O(E)$ iterations. Because each call to PROCESS requires only $O(1)$ time, we conclude that the entire algorithm runs in ***O(E) time***.

To prove the algorithm correct, we need to establish a few claims.

Claim 1. *Let G be any series-parallel graph with at least three vertices. At least one vertex in G exactly one in-neighbor and exactly one out-neighbor.*

Proof: Consider any decomposition tree T. Let α be the deepest serial node in T, and let v be the corresponding vertex in G. The two subgraphs joined at v each consist entirely of parallel edges between v and another vertex. Thus, all edges in G coming into v come from the same vertex, and all edges in G leaving v go to the same vertex. □

Claim 2. *Let G be any series-parallel graph, and let v be any vertex of G. Whenever the set of neighbors of v changes, v is inserted into the bag.*

Proof: The set of neighbors of v changes only if we perform a series reduction of the form $v \to x \to y \rightsquigarrow v \to y$ or $y \to x \to v \rightsquigarrow y \to v$. In both cases, PROCESS$(x)$ inserts v (and y) into the bag. □

Claim 3. *At all times during the execution of DECOMPOSITIONTREE(G), the bag contains every vertex of G with exactly one in-neighbor and exactly one out-neighbor.*

Proof: Consider any vertex v not equal to s or t. Claim 2 implies that the first time v has only two neighbors, either at the start of the algorithm or after a series reduction, v is inserted into the bag. When v is taken out of the bag, PROCESS(v) either performs a series reduction at v, thereby removing v from the graph, of performs a parallel reduction at v and reinserts v into the bag. Thus, once v has only two neighbors, v remains in the bag until it is deleted. □

Claims 1 and 3 immediately implies that when the bag is empty, the only remaining vertices are s and t. At this point, there may be several parallel edges between s and t, but the algorithm performs parallel reductions to eliminate those parallel edges, leaving only a single edge $s \to t$.

Rubric: 10 points = 4 for how to perform series and parallel reductions + 4 for details of finding reductions to perform + 2 for time analysis. This is not the only correct solution. Full credit for any polynomial-time algorithm.

+5 extra credit points for a correct $O(E)$-time algorithm = 2 for algorithm + 2 for proof of correctness + 1 for time analysis. This is not the only correct $O(E)$-time algorithm. $O(E)$ *expected* time (using hashing, for example) is fine.

+3 extra credit for a correct $O(E \log V)$-time algorithm = 1 for algorithm + 1 for correctness proof + 1 for time analysis.

Grade part (b) as a separate numbered problem.

(b) Describe and analyze an algorithm to compute a maximum (s,t)-flow in a given (s,t)-series-parallel flow network with arbitrary edge capacities. *[Hint: In light of part (a), you can assume that you are actually given the decomposition tree. First compute the maximum-flow value, then compute an actual maximum flow.]*

Solution: Let T denote the decomposition tree of the input graph G. To simplify presentation, I will always use the words "vertex" and "edge" to refer to features of G, and the word "node" to refer to nodes of the decomposition tree T. The record for each node α in T has several fields:

- $\alpha.vertex$: The corresponding vertex if α is a serial node.
- $\alpha.edge$: The corresponding edge if α is a leaf.
- $\alpha.left$: The left child of α, or NULL if α is a leaf.
- $\alpha.right$: The right child of α, or NULL if α is a leaf.
- $\alpha.type$: series, parallel, or leaf.

For any node α in T, let $G(\alpha)$ denote the subgraph of G corresponding to the subgraph of T rooted at α. If α is a serial node, then $\alpha.vertex$ is the target vertex of $G(\alpha.left)$ and the source vertex of $G(\alpha.right)$. Children of parallel nodes are ordered arbitrarily.

For every node α in the decomposition tree, we define two values:

- $cap(\alpha)$ is the maximum amount of flow that can be sent through $G(\alpha)$.
- $flow(\alpha)$ is the actual amount of flow sent through $G(\alpha)$ by a maximum flow through G.

The difference between these is that $cap(\alpha)$ considers the subgraph $G(\alpha)$ in isolation, while $flow(\alpha)$ considers $G(\alpha)$ as a piece of the larger graph G.

The "capacity" function $cap(\alpha)$ satisfies the following recursive definition:

$$cap(\alpha) = \begin{cases} c(\alpha.edge) & \text{if } \alpha \text{ is a leaf} \\ \min\{cap(\alpha.left), cap(\alpha.right)\} & \text{if } \alpha \text{ is a serial node} \\ cap(\alpha.left) + cap(\alpha.right) & \text{if } \alpha \text{ is a parallel node} \end{cases}$$

We can compute $cap(\alpha)$ for every node α in $O(E)$ time by postorder traversal, memoizing each value $cap(\alpha)$ into a new field $\alpha.cap$.

Then we can compute the "flow" function $flow(\alpha)$ for every node α in $O(E)$ time using a *preorder* traversal as follows:

- If α is the root of T, then $flow(\alpha) = cap(\alpha)$.
- If α is a serial node, then $flow(\alpha.left) = flow(\alpha.right) = \alpha.flow$.
- If α is a parallel node, then we can assign

$$flow(\alpha.left) \leftarrow \min\{flow(\alpha), cap(\alpha.left)\}$$
$$flow(\alpha.right) \leftarrow flow(\alpha) - flow(\alpha.left).$$

(Here I've made an arbitrary choice to push as much flow as possible through the "left" subgraph of a parallel split; this is not the only reasonable choice.) Finally, for each leaf α, we assign the flow value $f(\alpha.edge) \leftarrow flow(\alpha)$ to the corresponding edge.

Altogether, the algorithm runs in ***O(E) time***. Complete pseudocode is shown on the next page.

```
⟨⟨Assumes G is (s,t)-series-parallel⟩⟩
MaxFlow(G, s, t):
  T ← DecompositionTree(G)

  ⟨⟨Compute the value of the maximum flow bottom-up⟩⟩
  ⟨⟨α.cap is the capacity of subgraph G(α)⟩⟩
  for all nodes α in T in postorder
      if α is a leaf
          α.cap ← c(α.edge)
      if α is a serial node
          α.cap ← min{α.left.cap, α.right.cap}
      if α is a parallel node
          α.cap ← α.left.cap + α.right.cap

  ⟨⟨Compute the actual maximum flow top-down⟩⟩
  ⟨⟨α.flow is how much G's maximum flow sends through G(α)⟩⟩
  T.root.flow ← T.root.cap
  for all nodes α in T in preorder
      if α is a leaf
          f(α.edge) ← α.flow
      if α is a serial node
          α.left.flow ← α.flow
          α.right.flow ← α.flow
      if α is a parallel node
          α.left.flow ← min{α.flow, α.left.cap}
          α.right.flow ← α.flow − α.left.flow
```

■

Rubric: Graded as a separate numbered problem.
10 points = 5 for computing the value of the maximum flow (scaled dynamic programming rubric) + 5 for computing the actual flow values at the edges (= 1½ for specifying the flow function in English + 2½ for computing the flow function + 1 for time analysis). As usual, it is not necessary to give explicit pseudocode.