Homework Assignment 2

Submission Guidelines.

- Latex submissions are preferred but not mandatory. However, please write your solutions on unruled paper and upload a clearly scanned copy. Please do not use poorly lit photographs. Keep your solutions brief and to the point. Meaningless rambles fetch negative credits.
- Collaboration is allowed in teams of up to two students, but each student must write and submit their own individual solutions in their own words. Also mention the name of your collaborator.
- "Bibles" such as textbooks, lecture notes from other courses, or online forums (e.g., Stack Exchange) may not be used as references for your write-up.
- Limited use of large language models (LLMs) is permitted, but you may not directly copy-paste text from them; use them only for clarification and then produce your own reasoning and explanations.

Problem 1. In the s-t directed edge-disjoint paths problem, the input is a directed graph $G = (V, E)$, a source vertex s, and a sink vertex t. The goal is to output a maximum-cardinality set of edge-disjoint s-t paths $P_1, \ldots, P_k$. (I.e., P_i and P_j should share no edges for each $i \neq j$, and k should be as large as possible.)

(a) (10 points) Prove that this problem reduces to the maximum flow problem. That is, given an instance of the disjoint paths problem, show how to

(i) produce an instance of the maximum flow problem such that

(ii) given a maximum flow to this instance, you can compute an optimal solution to the disjoint paths instance.

Solution Sketch.

(i) Constructing the Instance

Given the input graph $G = (V, E)$ with source s and sink t, we construct a flow network $G' = (V, E')$ as follows:

(a) The vertices and edges of G' are identical to those in G.

(b) Assign a capacity $u_e = 1$ to every edge $e \in E$.

(c) The source and sink remain s and t.

Runtime: We iterate through the set of edges E exactly once to assign capacities. This takes $O(m)$ time, which is linear.

Rubric : +3 for correct description and +2 for runtime.

(ii) Computing the Optimal Solution

Let f be an integral maximum flow computed on G'. Since all capacities are 1, the flow on any edge f_e must be either 0 or 1. The value of the flow $|f| = k$ corresponds to the maximum number of edge-disjoint paths.

To recover the paths $P_1, \ldots, P_k$:

Algorithm 1 Path Reconstruction (Polynomial Time)

```
Initialize 𝒫 ← ∅.
while there exists an s-t path in G' using only edges where f_e = 1 do
    Find a path P from s to t using DFS/BFS on edges with f_e = 1.
    Add P to 𝒫.
    For every edge e ∈ P, set f_e ← 0 (effectively removing the edge).
end while
return 𝒫
```

Correctness: Since capacities are unit (1), no edge can carry more than 1 unit of flow. Thus, any edge used in path P_i cannot be used in P_j (where $i \neq j$), ensuring the paths are edge-disjoint. The maximality of the flow ensures we find the maximum number of such paths.

Runtime: Finding a single path takes $O(m)$. We repeat this k times. Since $k \leq m$, the total runtime is $O(km)$ or $O(m^2)$, which is polynomial.

Rubric : +2 for algorithm description, +2 for correctness, +1 for runtime.

Your implementations of steps (i) and (ii) should run in linear and polynomial time, respectively.

(b) (10 points) Can you achieve linear time also for (ii)? Include a description of the algorithm and brief proof of correctness and runtime.

Remark: Linear time always means $O(m + n)$

Solution Sketch. We can achieve a linear time complexity of $O(m + n)$ for the reconstruction step by avoiding the repeated scanning of edges. This is essentially a DFS search through the graph subgraph of G' with unit flow on edges. You can also simply do the following - run a DFS from s until you hit t. In this process, we need to do two additional things - (a) Once you explore an edge uv, add it to the current path and delete it from G' (this ensures edge disjointness). (b) if you ever backtrack via an edge say uv before reaching t, then you may delete v safely. Once you find a path to t, return the path and start a fresh search from s.

Below is a formalization of this, which I do not expect you to write.

Algorithm 2 Linear Time Path Reconstruction

Input: Flow assignment f on graph G.
Construct subgraph G_{flow} containing only edges where $f_e = 1$.
For each vertex v, maintain a pointer $current\_edge[v]$ to the first neighbor in G_{flow}.
Initialize $\mathcal{P} \leftarrow \emptyset$.
while s has outgoing edges in G_{flow} **do**
    Initialize path $P = \{s\}$, current vertex $u = s$.
    while $u \neq t$ **do**
        Let (u, v) be the edge pointed to by $current\_edge[u]$.
        Append (u, v) to P.
        Advance $current\_edge[u]$ (effectively deleting (u, v) from future visits).
        $u \leftarrow v$
    end while
    Add P to $\mathcal{P}$.
end while
return $\mathcal{P}$

Runtime Analysis:

- Creating G_{flow} takes $O(m)$.
- Every edge carrying flow ($f_e = 1$) is visited exactly once. Once an edge is added to a path, the edge is deleted.
- The total work is proportional to the number of edges with flow plus the number of vertices visited. Since total flow edges $\leq m$, the runtime is $O(m + n)$.

Rubric : +7 for the correct algorithm (details like DFS, deletion of a visited edge and deletion of a vertex while backtracking carry weightage), +3 for justifying the runtime properly

Problem 2. Consider a directed graph $G = (V, E)$ with source s and sink t for which each edge e has a positive integral capacity u_e. Recall from Lecture that a *blocking flow* in such a network is a flow $\{f_e\}_{e \in E}$ with the property that, for every s-t path P of G, there is at least one edge of P such that $f_e = u_e$. For example, our first (broken) greedy algorithm terminates with a blocking flow (which, as we saw, is not necessarily a maximum flow).

Algorithm 3 Dinic's Algorithm

initialize $f_e = 0$ for all $e \in E$
while there is an s-t path in the current residual network G_f **do**
construct the layered graph L_f, by computing the residual graph G_f and running breadth-first search (BFS) in G_f starting from s, stopping once the sink t is reached, and retaining only the forward edges[1]
compute a blocking flow g in L_f ▷ augment the flow f using the flow g
for all edges (v, w) of G for which the corresponding forward edge of G_f carries flow ($g_{vw} > 0$) **do**
increase f_e by g_e
end for
for all edges (v, w) of G for which the corresponding reverse edge of G_f carries flow ($g_{wv} > 0$) **do**
decrease f_e by g_e
end for
end while

The termination condition implies that the algorithm can only halt with a maximum flow. Its running time is therefore $O(n \cdot BF)$, where BF is the amount of time required to compute a blocking flow in the layered graph L_f. We know that $BF = O(m^2)$ — our first broken greedy algorithm already proves this — but we can do better.

Consider the following algorithm, inspired by depth-first search, for computing a blocking flow in L_f:

Algorithm 4 Blocking Flow Algorithm

Initialize. Initialize the flow variables g_e to 0 for all $e \in E$. Initialize the path variable P as the empty path, from s to itself. Go to **Advance**.

Advance. Let v denote the current endpoint of the path P. If there is no edge out of v, go to **Retreat**. Otherwise, append one such edge (v, w) to the path P. If $w \neq t$ then go to **Advance**. If $w = t$ then go to **Augment**.

Retreat. Let v denote the current endpoint of the path P. If $v = s$ then halt. Otherwise, delete v and all of its incident edges from L_f. Remove from P its last edge. Go to **Advance**.

Augment. Let Δ denote the smallest residual capacity of an edge on the path P (which must be an s-t path). Increase g_e by Δ on all edges $e \in P$. Delete newly saturated edges from L_f, and let $e = (v, w)$ denote the first such edge on P. Retain only the subpath of P from s to v. Go to **Advance**.

And now the analysis:

(a) (5 points) Argue formally that every iteration of the main loop increases $d(f)$, the length (i.e., number of hops) of a shortest s-t path in G_f.

Solution.

Proof. Let f be the flow at the start of an iteration, and let L_f be the layered graph constructed from G_f. L_f contains only edges (u, v) such that $\text{level}(v) = \text{level}(u) + 1$, where $\text{level}(u)$ is the breadth-first distance from s to u. Thus, every path from s to t in L_f has length exactly $d(f)$.

The algorithm computes a blocking flow g in L_f and updates the total flow to $f' = f + g$. A blocking flow saturates at least one edge on every directed s-t path in L_f.

Consider any shortest s-t path in the new residual graph $G_{f'}$. This path can consist of:

(a) **Edges from L_f that were not saturated:** These edges advance the level by exactly $+1$.

(b) **Edges from G_f that were not in L_f:** These edges (u, v) satisfy $\text{level}(v) \leq \text{level}(u)$ (otherwise they would be in L_f). They do not provide "shortcuts" relative to the layering.

(c) **Reverse edges created by g:** If flow is pushed on $(u, v) \in L_f$, a reverse edge (v, u) appears in $G_{f'}$. Since $(u, v) \in L_f$, we know $\text{level}(v) = \text{level}(u) + 1$. Thus, the reverse edge goes from level $i + 1$ to level i, strictly decreasing the distance to s.

Since g is a blocking flow, every path of length $d(f)$ in L_f is broken (contains at least one saturated edge). Therefore, any new path from s to t in $G_{f'}$ must use edges that either maintain the level, go backwards in level, or traverse edges not in the shortest-path subgraph. Consequently, the new shortest path distance must be strictly greater than $d(f)$. □

Rubric : Roughly $+1$ for arguing about each category of edges and $+2$ for a generally sound argument.

(b) (10 points) Prove that the running time of the algorithm, suitably implemented, is $O(mn)$. (As always, m denotes $|E|$ and n denotes $|V|$.)

Hint: How many times can **Retreat** be called? How many times can **Augment** be called? How many times can **Advance** be called before a call to **Retreat** or **Augment**?

Solution Sketch. We analyze the running time of Algorithm 2 (the DFS-based Blocking Flow Algorithm) by counting the total number of operations performed by its three main procedures: **Advance**, **Retreat**, and **Augment**.

- **Augment:** An augmentation occurs when the path P reaches t. For each Augment call, the algorithm finds the bottleneck capacity Δ, updates flow on all edges in P, and deletes *at least one* saturated edge from L_f. Thus we can conclude the following.
  - Since each augmentation deletes at least one edge, and there are m edges total, there can be at most m augmentations.
  - The path P has length at most n.
  - Total work for Augment: $O(mn)$.

  Rubric : $+3$ for correctness
- **Retreat:** A retreat occurs when the current vertex v has no outgoing edges. The vertex v (or the edge leading to it) is effectively removed from consideration because it cannot lead to t.
  - In this implementation, we delete edges incident to v or remove the edge from P. We can charge the cost of a retreat to the edge being deleted/removed from the graph.

– Since each edge is deleted at most once, the total work for Retreat is $O(m)$.

Rubric : +3 for correctness

- **Advance:** An advance adds an edge (v, w) to the path P. Once added, this edge will eventually meet one of two fates:
  (a) It becomes part of an **Augment** operation. In this case, the cost of the Advance is absorbed by the $O(n)$ cost of one call to Augment.
  (b) It triggers a **Retreat** (if w turns out to be a dead end). In this case, the edge is deleted. We charge this Advance step to the edge deletion.

  Since every Advance step leads to either an augmentation (at most m times, length n) or an edge deletion (at most m times), the total complexity is dominated by the augmentations.

Rubric : +2 for each of the above points - so +4 in all. **Total Runtime:** $O(mn) + O(m) = O(mn)$.

(c) (10 points) Prove that the algorithm terminates with a blocking flow g in L_f.

[For example, you could argue by contradiction.]

Solution Sketch. We prove the following more general statement - for any vertex v, if Retreat is called on v, then there does not exist a path in L_f from v to t. We prove this by induction on the distance of v from t. Firstly, note that Retreat is never called on t and hence the induction hypothesis holds. Now consider a vertex v such that vt is an edge in L_f. Now if Retreat has been called on v, it is only due to the fact that there are not outgoing edges from v in L_f. In particular, the edge vt does not exist - this is the base case. N

Now consider this hypothesis to be true for all vertices that are k hops away from t in L_f and let v be a vertex which is $k+1$ hops away such that Retreat has been called on v. This can only happen if all the edges out of v to the next layer has been deleted. There could be two cases - (a) All edges out of v are saturated (in which case the deletion has occurred in the Augment step) which implies no path from v to t or (b) there existed some non-saturated edge vv' that had been deleted since Retreat was called on v'. By induction hypothesis, there is no path from v' to t and hence there cannot be a path from v to t through v'.

Rubric : +2 for setting up the induction/contradiction correctly. +3 for each of the cases . +2 for finally observing that the algorithm halts if and only if Retreat is called on s. (in case of alternate proofs, I will apply my objective/subjective judgment).

Problem 3. In lecture we proved a bound of $O(n^3)$ on the number of push and relabel operations needed by the Push-Relabel algorithm (where, in each iteration, we select the highest vertex with excess to Push or Relabel) before it terminates with a maximum flow. In the problem, you will give a detailed implementation of this algorithm that runs in $O(n^3)$ time.

(a) (5 points) First prove the running time bound assuming that, in each iteration, you can identify the highest vertex with positive excess in $O(1)$ time.

Solution Sketch. To analyze the complexity rigorously, we first explicitly define the main loop of the algorithm to pinpoint exactly when the vertex selection occurs.

Algorithm 5 Highest-Label Push-Relabel Main Loop

```
1: while there exists a vertex v ∈ V \ {s, t} with excess e(v) > 0 do
2:     u ← vertex with e(u) > 0 maximizing h(u)          ▷ Assumed O(1)
3:     if there exists an admissible edge (u, v) in G_f then
4:         PUSH(u, v)
5:     else
6:         RELABEL(u)
7:     end if
8: end while
```

Let K be the total number of iterations of the **while** loop. The total running time T is the sum of the selection cost and the action cost over all K iterations:

$$T = \sum_{i=1}^{K} (\text{Cost of Selection} + \text{Cost of Action})$$

1. Total Selection Cost: Since the selection happens exactly once per iteration and takes $O(1)$ by assumption:

$$\text{Total Selection Cost} = K \times O(1)$$

2. Total Iterations (K): Every iteration performs exactly one operation. We bound the total count of each operation type:

- **Relabels:** Since $h(v) \leq 2n$, each vertex is relabeled $O(n)$ times. Total: $O(n^2)$.
- **Saturating Pushes:** Each edge is saturated $O(n)$ times. Total: $O(nm)$.
- **Non-Saturating Pushes:** Using the standard phase analysis (where a phase is the period between relabels), there are at most n non-saturating pushes per phase. With $O(n^2)$ phases, Total: $O(n^3)$.

Thus, the total number of iterations is $K = O(n^2) + O(nm) + O(n^3) = O(n^3)$.

3. Total Runtime: Substituting $K = O(n^3)$ into the time equation:

$$T = \underbrace{(O(n^3) \times O(1))}_{\text{Selection}} + \underbrace{(O(nm) + O(nm) + O(n^3))}_{\text{Actions}}$$
$$T = O(n^3)$$

(b) (15 points) The hard part is to maintain the vertices with positive excess in a data structure such that the amortized time for identifying the vertex with excess and highest label is $O(n^3)$. Can you get away with just a collection of buckets (implemented as lists) ?

Solution Sketch.
The Data Structure

- **Buckets (B):** An array of size $2n$, where $B[k]$ is a doubly linked list containing all active vertices currently at height k.

- **Highest-Label-Ptr (ptr):** An integer pointer tracking the index of the highest non-empty bucket.

2. Key Operations

- **Insert (v):** Add v to list $B[h(v)]$. If $h(v) > ptr$, update $ptr \leftarrow h(v)$. ($O(1)$ time).
- **Extract Max:** If $B[ptr]$ is empty, decrement ptr until a non-empty bucket is found. Remove and return the head of $B[ptr]$.

3. Amortized Analysis The total running time is determined by two factors:

1. **List Maintenance ($O(n^3)$):** This is the easier one to see. Insertion and deletion from doubly linked lists are $O(1)$ operations. These occur exactly once per Push or Relabel. Since there are $O(n^3)$ total operations (established in Part A), the total time spent manipulating lists is $O(n^3)$.
2. **Pointer Scanning ($O(n^2)$):** This is the most critical part in the analysis since naive argument would lead to $O(n)$ ptr movements per iteration. However, the simple observation is that any pointer movement can be essentially charged to relabels. We formalize as follows The **Extract Max** operation lazily decrements ptr. While a single search might take $O(n)$, we bound the *aggregate* movement of ptr over the entire execution:
   - **Increases:** ptr only increases when a vertex is relabeled. The sum of all height increases for all vertices is bounded by $O(n^2)$. Therefore, the total upward movement of ptr is $O(n^2)$.
   - **Decreases:** Since $ptr \geq 0$, the total number of decrements cannot exceed the total increments. Thus, the total downward scanning work is also $O(n^2)$.

Conclusion: The total time complexity for the data structure is $O(n^3) + O(n^2) = O(n^3)$.

Rubric : +5 for the correct data structure. +2 for stating the important operations insert and Extract. +4 for analyzing each of the operations in reasonable detail with analysis.