CS 161 (Stanford, Fall 2025) Section 8

1 Warm up - True or False

Decide whether you think the following statement is true or false. If it is true, give a short explanation. If it is false, give a counterexample.

Let G be an arbitrary flow network, with a source s, a sink t, and a positive integer capacity c_e on every edge e, and let (A, B) be a minimum s-t cut with respect to these capacities $\{c_e : e \in E\}$. Now suppose we add 1 to every capacity; then (A, B) is still a minimum s-t cut with respect to these new capacities $\{1 + c_e : e \in E\}$.

Solution

This is false. Consider a graph with nodes s, v_1, v_2, v_3, w, t and edges (s, v_i), (v_i, w) for each i and an edge (w, t). Let the capacity of edge (w, t) be 4 and let all other edges have capacity 1. Then a minimum s-t cut is obtained by setting $A = \{s\}$ and $B = \{V - A\}$ which has capacity 3. But if we add 1 to every edge, then this cut has capacity 6, more than the capacity of 5 on the cut with $B = \{t\}$ and $A = \{V - t\}$.

2 Max Flow Potpourri

How would you use a max flow algorithm to handle the following situations?

1. Suppose that instead of having a single source s and a single sink t, we have multiple sources $S = \{s_1, s_2, ..., s_k\}$ and multiple sinks $T = \{t_1, t_2, ..., t_\ell\}$. How can you find the max flow in the graph from sources to sinks?

   Solution

   Create a meta source node s' connected to all source nodes with edge weight ∞. Likewise, create a meta sink node t' where all sink nodes are connected to t' with weight ∞.

2. Suppose that in addition to edges having max flow capacities, vertices also have a limit to their capacity; that is, each vertex v_i has capacity c_i. How can you find the max flow from a source s to sink t in this graph?

   Solution

   Replace each vertex v_i with v_i and v_i', where there is an edge $v_i \to v_i'$ with weight c_i. Replace any edge (v_i, w) going out of v_i by (v_i', w).

3 Task Selection

Suppose you have a set of k tasks $t_1, \ldots, t_k$. There are certain tasks such that t_i is a prerequisite of t_j. Each task t_i also has an integer reward r_i, which may be negative. Find an optimal subset of tasks to complete to maximize your reward.

Solution

Have a vertex v_i for each task t_i. Draw an edge from $v_j \to v_i$ with weight ∞ if t_i is a prereq of t_j. If $r_i \geq 0$, add an edge from $s \to v_i$ with weight r_i. If $r_i < 0$, add an edge from $v_i \to t$ with weight $-r_i$. Let (A, B) be a min s-t cut, where A contains s. Then $A - \{s\}$ is an optimal set of tasks. This problem is also known as the "closure" problem.

Proof of correctness (not required):
First, note that the weight of an edge only counts towards the size of the cut if the edge is coming out of A (ie. the edge is from a vertex in A to a vertex not in A).
Consider a minimizing cut, which must have no "∞" edges coming out of A. Therefore, we know that the edges coming out of A are either from s to a vertex with a positive reward (which happens when we do not include a task with a positive reward), or from a vertex with a negative reward to t (which happens when we include a task with a negative reward). Below, we will call vertices with a positive reward "positive" and vertices with a negative reward "negative". We have the size of the cut:

$$|cut| = \sum \text{reward of positive } v \notin A - \sum \text{reward of negative } v \in A$$

Note that both positive rewards and negative rewards contribute a positive number to the size of the cut (since edges are all positive). On the other hand, the sum of the rewards of all the vertices in A gives us the total reward of all the tasks we pick:

$$\sum A = \sum \text{reward of positive } v \in A + \sum \text{reward of negative } v \in A$$

Algebra tells us that

$$\begin{aligned} & |cut| + \sum A \\ = & \sum \text{reward of positive } v \notin A - \sum \text{reward of negative } v \in A \\ & + \sum \text{reward of positive } v \in A + \sum \text{reward of negative } v \in A \\ = & \sum \text{reward of positive } v \notin A + \sum \text{reward of positive } v \in A \\ = & \sum \text{reward of all positive } v \text{ in the graph} \end{aligned}$$

The sum of all the positive rewards among all the tasks is a constant. Therefore, the total reward $\sum A$ is maximized when the size of the cut is minimized.

4 Tiling Partial Checkerboard

Suppose you are given an $n \times n$ checkerboard with some of the squares deleted. You have a large set of dominoes, each of which is just the right size to cover two adjacent squares of the checkerboard. Describe and analyze an algorithm to determine whether one can completely tile the board with dominoes. Each domino must cover exactly two adjacent undeleted squares, and each undeleted square must be covered by exactly one domino.

Your input is a two-dimensional $n \times n$ array $Deleted$, where $Deleted[i, j]$=True if and only if the square in row i and column j has been deleted. Your output is a single bit; you do not have to compute the actual placement of dominoes.

Solution

Let G be the bipartite graph whose vertices are the black and white squares of the partial checkerboard, and whose edges join pairs of squares that share a side (and therefore can be covered by one domino). This graph G has $O(n^2)$ vertices and $O(n^2)$ edges. The board can be tiled with dominoes if and only if G has a perfect matching, that is, there is a subset M of the edges of G such that each vertex of G touches exactly one edge in M.
Proof:

- First, suppose G has a perfect matching M. By definition of the edges in G, each edge in M corresponds to a pair of squares that can be covered by a domino. By definition of “perfect matching," every square on the board is covered by exactly one such domino. Thus, by definition, these dominoes tile the board.
- Conversely, suppose the board can be tiled with dominoes. Each domino in the tiling corresponds to an edge in G; call the resulting set of edges M. By definition of “tiling,” each vertex of G is incident to exactly one edge in M. Thus, by definition, M is a perfect matching.

Building G from the input array requires $O(n^2)$ time, and a maximum matching in G can be computed using e.g., Ford-Fulkerson in $O(|f||E|)$ time, where $|f|$ is the value of maximum flow. See lecture 15 notes section 8.1 for how to use Ford-Fulkerson to find a perfect matching. Since the maximum flow has value at most $O(n^2)$, altogether, this algorithm runs in $O(n^4)$ time.