CS 161 (Stanford, Winter 2026) Section 9

1 Max-Cut

In this question we'll try to come up with algorithms for the **Max-Cut** problem, which is just like Min-Cut but with the opposite objective: we're given an *undirected, unweighted* graph $G = (V, E)$, and our goal is to find a partition of the vertices into subsets $S, V \setminus S$ that maximizes the *number of edges* going from S to $V \setminus S$.

1.1 Modified Ford-Fulkerson

Siggi the Studious Stork comes up with the following algorithm to solve the Max-Cut problem. Is Siggi's algorithm correct?

Assume that all edges have weight 1. Enumerate over all candidate pairs of (s, t). For each pair find the *minimum s-t* flow, using the idea that a MinFlow corresponds to a MaxCut (consider the MinCut = MaxFlow theorem we saw in class, just reversing min and max).

If the algorithm is correct, please provide an informal explanation. If the algorithm is incorrect, please provide a counter-example with a brief explanation.

Solution

This does not work because the idea that MinFlow = MaxCut is not correct. On any graph, every choice of s and t has the same minimum flow, which sends 0 units of flow on every edge. So a minimum flow does not help us find a maximum cut.

1.2 Modified BFS

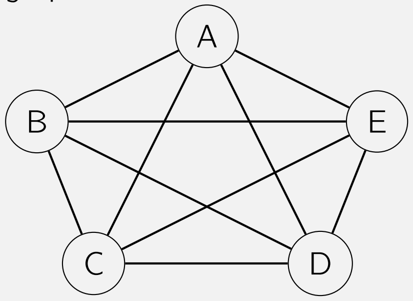
Ollie the Overachieving Ostrich comes up with a different approach to solve the Max-Cut problem. Is Ollie's algorithm correct?

Initialize two empty sets S_1 and S_2. Run BFS on the graph starting at a random node, adding the start node to S_1. Then at each step of BFS, add the current node to the opposite set as its parent (the node it was discovered from). Terminate once all nodes have been discovered, and return $\{S_1, S_2\}$ as the cut.

If the algorithm is correct, please provide an informal explanation. If the algorithm is incorrect, provide a counter-example and an explanation of why it is a counter-example.

Solution

This does not work. To see why, consider running this algorithm on a fully connected graph with 5 nodes:



If we start BFS starting at any vertex v, then we will deterministically return $S_1 = \{v\}$, $S_2 = V \setminus \{v\}$, with only 4 edges crossing the cut. However, a max cut on this graph is something like $S_1 = \{A, B\}$, $S_2 = \{C, D, E\}$, with 6 edges crossing the cut.

2 Class Selection

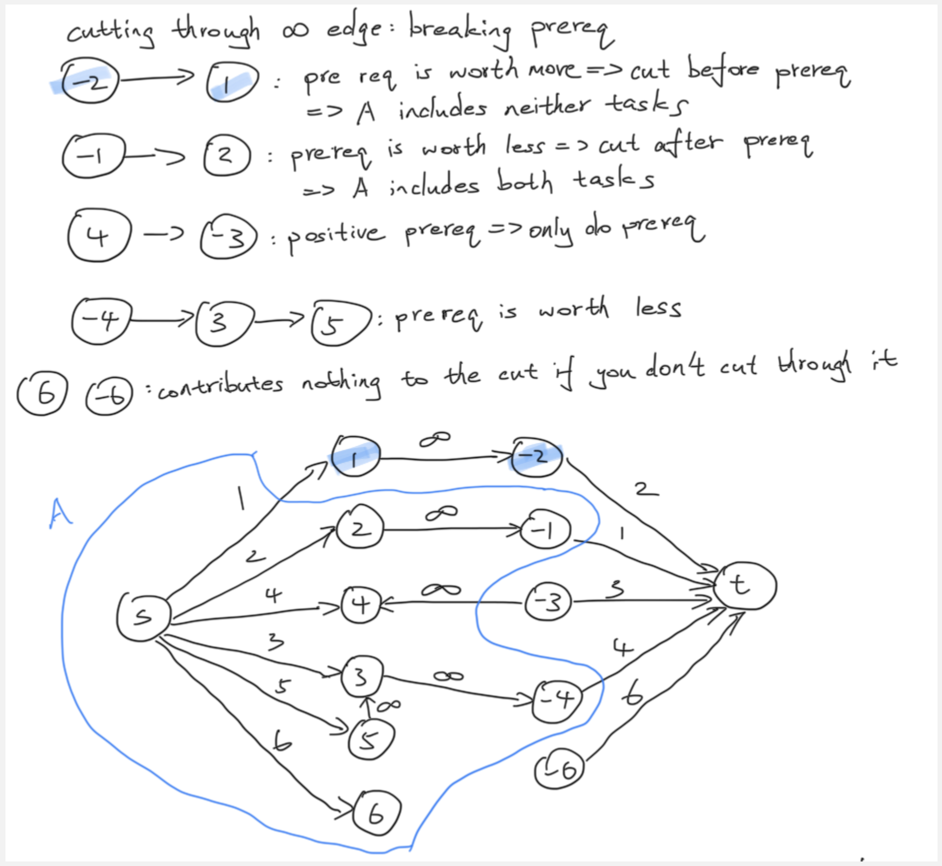
Suppose you have a set of k classes $c_1, ..., c_k$. There are certain classes such that c_i is a prerequisite of c_j. Each class c_i also has an integer reward r_i, which may be negative. Find an optimal subset of classes to complete to maximize your reward.

Solution

The high level idea why we want to translate this into a flow-cut problem is that the the cut is a separation of vertices, and we want to separate classes into ones that we choose and ones that we don't. Minimizing the cut, then, should translate into minimizing the positive rewards that we missed (because we wanted to avoid a larger negative pre-requisite), or the negative rewards that we kept (because it led to a larger positive post-requisite).

Have a vertex v_i for each class c_i. Draw an edge from $v_j \to v_i$ with weight ∞ if c_i is a prereq of c_j; this corresponds to the constraint that we cannot include c_j without including c_i . If $r_i \geq 0$, add an edge from $s \to v_i$ with weight r_i; this corresponds to a penalty to the size of the cut if we don't take v_i with a positive reward. If $r_i < 0$, add an edge from $v_i \to t$ with weight $-r_i$; this corresponds to a penalty to the size of the cut if we take v_i with a negative reward. Let (A, B) be a min s-t cut, where A contains s. Then $A - \{s\}$ is an optimal set of classes. This problem is also known as the "closure" problem.

See below for a visualization:



cutting through ∞ edge: breaking prereq

$-2 \rightarrow 1$: pre req is worth more => cut before prereq => A includes neither tasks

$-1 \rightarrow 2$: prereq is worth less => cut after prereq => A includes both tasks

$4 \rightarrow -3$: positive prereq => only do prereq

$-4 \rightarrow 3 \rightarrow 5$: prereq is worth less

6 -6 : contributes nothing to the cut if you don't cut through it

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 class with a positive reward), or from a vertex with a negative reward to t (which happens when we include a class 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 classes 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 classes is a constant. Therefore, the total reward $\sum A$ is maximized when the size of the cut is minimized.

3 Max Flow Potpourri

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

3.1

Suppose that instead of having a single source s and a single sink t, we have multiple sources $S = \{s_1, s_2, \ldots, s_k\}$ and multiple sinks $T = \{t_1, t_2, \ldots, t_\ell\}$. 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 ∞.

3.2

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).

4 Expense Settling

You've gone out to eat with k friends, where friend i paid c_i for the group's expenses. The expenses should be split equally amongst the friends. You would like to develop an algorithm to ensure that everyone gets paid back fairly, but without going through one person (that is, each person should either pay or receive money but not both).

Solution

Calculate the per person cost, $c = \frac{\sum c_i}{k}$. People who paid more than c need to get paid back, while people who paid less need to pay others. Create a graph with a source node s, sink node t, and one node per person v_i. If $c_i > c$, this person needs to get paid back, and we draw an edge from $v_i \to t$ with weight $c_i - c$. If $c_i < c$, this person needs to pay other people, and we draw an edge from $s \to v_i$ with weight $c - c_i$. We connect all pairs of vertices $v_i \to v_j$ with edge weight ∞ if $c_i < c$ and $c_j > c$. We find the max flow from source to sink in the graph, and the flow along an edge will represent how much people pay one another.

5 Fear of Negativity

Do our graph algorithms work when the weights are negative? Let's answer that in this problem. Assume that the graph is directed and that all edge weights are integers.

5.1 Negative Cycles

A "negative cycle" is a cycle where the sum of the edge weights is negative. Why can't we compute shortest paths in a graph with negative cycles?

Please provide an informal explanation.

Solution

If we have a negative cycle, each time we go around the cycle, we will decrease our path length. We will never have a "shortest" path since for each path we can find (that can be reached from any vertices in the negative cycle), we can just go around the cycle another time and get an even "shorter" path.

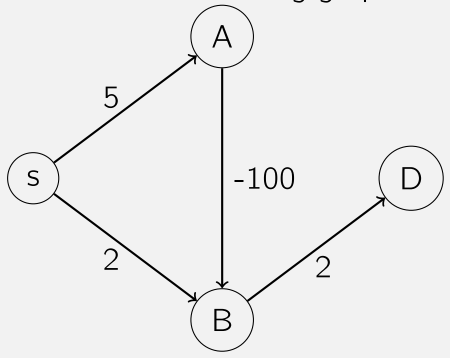
5.2 Negative Dijkstra

Even if a graph contains no negative cycles (but still contains negative edges) we might still be in trouble. Please draw a graph G, which contains both positive and negative edges but does not contain negative cycles, and specify some source $s \in V$ where `Dijkstra(G, s)` does not correctly compute the shortest paths from s.

Please provide a graph G with no more than 4 vertices (including a source node s), and an example of a shortest path from s that is not correctly computed using Dijkstra's algorithm.

Solution

Consider the following graph:



Here Dijkstra's algorithm incorrectly computes the distance from s to D. The shortest distance is supposed to be -93 (if you take the path $s \to A \to B \to D$) but instead Dijkstra's algorithm reports that the distance to D is 4.
The key is to exploit the fact that each node is only extracted from the queue once. Dijkstra's algorithm first extracts s from the queue, setting $d[A] = 5$, $d[B] = 2$. Next, it extracts B, which set $d[D] = 4$. Next, it extracts D, which produces no change in distances. Next, it extracts A, setting $d[B] = -95$, and terminating the algorithm. Ideally we would be able to relax edge (B, D) one more time, but we can't because B has already been extracted from the queue. This causes the distances and shortest paths to be incorrectly computed.

5.3 A Fix for Negative Dijkstra?

Consider the algorithm `Negative-Dijkstra` for computing shortest paths through graphs with negative edge weights (but without negative cycles). This algorithm adds some number to all of the edge weights to make them all non-negative, then runs `Dijkstra` on the resulting graph, and argues that the shortest paths in the new graph are the same as the shortest paths in the old graph.

```
Negative–Dijkstra(G, s):
    minWeight = minimum edge weight **in** G
    **for** e **in** E: # *iterate through all edges in G*
        modifiedWeight(e) = w(e) − minWeight
    modifiedG = G **with** weights modifiedWeight
    T = Dijkstra(modifiedG, s) # *run Dijkstra with modifiedWeight to get a SSSP Tree*
    update T to use weights w that corresponds to graph G
    **return** T
```

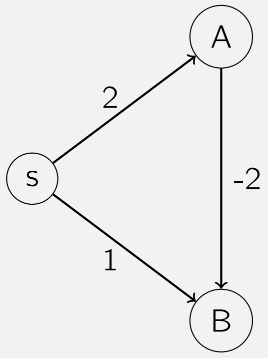
(Note that an "SSSP tree", or a "single-source-shortest-path tree", is analogous to a breadth-first-search tree in that paths in the SSSP tree correspond to shortest paths in the graph.

Here we assume that Dijkstra's algorithm has been modified to output such a tree.)

Prove or disprove: Negative-Dijkstra *always* computes single-source shortest paths correctly in graphs with negative edge weights.

To prove the algorithm correct, show that for all $u \in V$, a shortest path from s to u in the original graph lies in T. To disprove the algorithm, exhibit a graph with negative edges, but no negative cycles, where Negative-Dijkstra outputs the wrong "shortest" path, and explain why the algorithm fails.

Solution



Consider the following graph:
The shortest path from s to B is $S \longrightarrow A \longrightarrow B$. After we add 2 to each edge, $S \longrightarrow A \longrightarrow B$ will have total weight of 4 whereas $S \longrightarrow B$ will have weight 3, so the solution found by Negative-Dijkstra is $S \longrightarrow B$ rather than the correct answer $S \longrightarrow A \longrightarrow B$.
The intuition here is that since we are adding the same value for each edge, the longer paths get penalized more; in particular, a path of length ℓ gets a weight increase of $|\text{min weight}| \cdot \ell$, so that if the shortest path has many edges, it might still look bad due to the reweighting, and will be overlooked.

5.4 Negative Prim?

Since Prim's algorithm is very similar to Dijkstra, we want to now consider a similar algorithm `Negative-Prim` for computing minimum spanning tree in graphs with negative edge weights. Again, this algorithm adds some number to all of the edge weights to make them all non-negative, then runs `Prim's algorithm` on the resulting graph, and argues that the Minimum Spanning Tree in the new graph are the same as the MST in the old graph. You can assume that all the edge weights are unique integers.

```
Negative–Prim(G, s):
    minWeight = minimum edge weight in G
    for e in E: # iterate through all edges in G
        modifiedWeight(e) = w(e) − minWeight
    modifiedG = G with weights modifiedWeight
    T = Prim(modifiedG, s) # run Prim's algorithm starting from s
    update T with edges that corresponds to graph G
    return T
```

Please give either an informal explanation of why `Negative-Prim` computes the correct MST, or a counter-example of an undirected graph with negative edge weights where `Negative-Prim` does not output the correct minimum spanning tree, as well as an explanation of why it is a valid counter-example.

Solution

Since Prim's algorithm is a greedy algorithm that compares the edge weights between all neighbours, increasing the edge weight by the same amount for all the edges does not change the relative values between edges. Therefore, the tree produces by `Negative-Prim` is identical to the original Prim algorithm. Since we know Prim's algorithm is correct in graph with negative weight, `Negative-Prim` is correct as well.
Note: The proof of correctness for Prim we went through in class works regardless of edge weight being positive or negative, the lecture note include a proof with more details.