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 $V \setminus S$.

We've given two possible algorithms for the Max-Cut problem below. For each one, give a short explanation of why it does or doesn't find the max-cut.

1. **Modified Ford-Fulkerson.** 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 and an explanation of why it is a counter-example.

2. **Modified BFS.** 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.

3. **Greedy Algorithm.** Next you will design a **greedy** algorithm that runs in time $O(m + n)$ on G, an undirected and unweighted graph, and returns a cut of size at least $1/2$ times the maximum cut.

   Please provide an English description of your algorithm, an informal justification of correctness, and a runtime analysis.

   [Hint: You can always return a cut with at least $1/2$ *of all the edges in the graph.]*

Task Selection

Suppose you have a set of k tasks $t_1, ..., t_k$. 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.

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?

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?

Expense Settling

You've gone on a trip 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).

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.

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.

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.

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" paths, and explain why the algorithm fails.

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.