
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.

   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.

   Problem Solving Notes:

   (a) *Read and Interpret:* We want to know if we modify Ford-Fulkerson to solve a MinFlow problem that in turn gives us a solution to MaxCut.

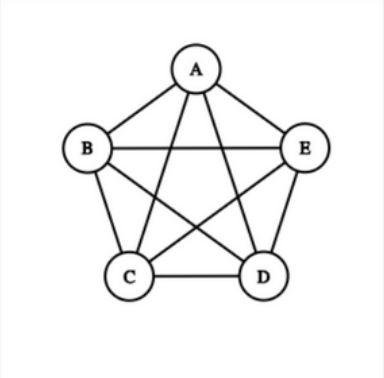
   (b) *Information Needed:* Does an algorithm giving MinFlow solve MaxCut for us?

   (c) *Solution Plan:* Test out some examples to see if it works, try to prove if it does.

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.

   SOLUTION: This does not work. To see why, consider running this algorithm on the following graph (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.

Problem Solving Notes:

(a) *Read and Interpret:* We want to know if a BFS-based partitioning gives us a max cut.

(b) *Information Needed:* Can we design a (possibly small) graph that has a MaxCut different from that returned by BFS (which is usually independent of the number of edges connecting different levels).

(c) *Solution Plan:* Test out some examples to see if it works, try to prove if it does.

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.]*

SOLUTION: First, we initialize two sets S_1 and S_2. Then, we iterate over all vertices in any order. At each vertex v, we add it to the set that has fewer of of v's neighbors. At the end, our cut is $\{S_1, S_2\}$.

We know that the cut returned by this approach will have at least $1/2$ of the edges in the graph crossing it. This is because at each step, when we add a new vertex v to either set, we consider all of v's neighbors. In doing so, we are really considering all of the new edges we are adding to the subgraph induced by $S_1 \cup S_2$. We choose to add v to the set such that at least half of these edges will cross the cut. Once we have added all nodes to either set, we have considered all edges in the graph, so at least half of all edges in the graph will cross the

cut.

For each vertex v, we are doing an $O(deg(v))$ operation to check which sets its neighbors are in, and an $O(1)$ operation to add it to one of two sets. So the overall runtime of our algorithm is $\sum_{v \in V} (O(deg(v)) + O(1)) = O(m + n)$.

Problem Solving Notes:

(a) *Read and Interpret:* We want to design an algorithm to return an *approximate* max cut of the graph. We know that choosing our partition greedily (based on some criteria) would give us a cut that is at least $1/2$ the size of the max cut.

(b) *Information Needed:* Given that we've already divided some k vertices into partitions, how do I greedily assign the next vertex into one of the partitions?

(c) *Solution Plan:* We decide to maintain a property that more edges cross the partition between the already partitioned vertices than the edges that don't cross the partition. In the end this results in a patition with the desired property.

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.

SOLUTION: 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.

Problem Solving Notes:

1. *Read and Interpret:* We have a pre-requisite graph on a set of tasks and we need to find the set of tasks to do that maximises the net reward for us.

2. *Information Needed:* We want to modify the reward of a task node into the capacity of edges. We want to make sure that partitioning the tasks into a chosen and a not-chosen set of tasks incurs a cut that corresponds to the rewards that I did not get.

3. *Solution Plan:* We first make sure that nodes outside the source partition contribute to the cut with a value negative of the reward of the task. Then we add edges with infinite weights so that the min cut algorithm never choses a task without choosing its pre-requisite.

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?

SOLUTION:

(a) 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 ∞.

(b) 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).

Problem Solving Notes:

(a) *Read and Interpret:* We want to use the max flow algorithm for calculating max flows on atypical graphs.

(b) *Information Needed:* Assumptions made by the max flow algorithm.

(c) *Solution Plan:* Modify the given graphs so that max flow in modified graphs have a direct correspondance with the max flow of the original graphs.

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).

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.

Problem Solving Notes:

1. *Read and Interpret:* We want to design an algorithm so that each person gets paid the amount they owe in total from other people without having to pay anything back to anyone.

2. *Information Needed:* The amount each person owes to the group or is owed by the group.
3. *Solution Plan:* Create a flow graph so that maximum flow equals the offset amount over everyone and flow values correspond to the amount each person pays other people.

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.

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.

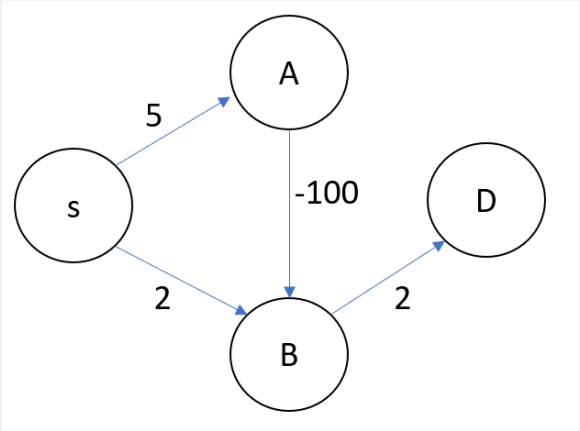
Problem Solving Notes:

1. *Read and Interpret:* This question is asking for why the shortest path cannot be computed with negative cycles. How do negative edge weights affect the cost of a path?
2. *Information Needed:* What happens if you make a cycle with negative paths?
3. *Solution Plan:* Play around with negative cycles. What happens if you keep going around the cycle?

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:

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.

Problem Solving Notes:

1. *Read and Interpret:* This question is asking for a graph that has negative edges (no cycles) where Dijkstra's algorithm wouldn't work.
2. *Information Needed:* How does Dijkstra's update the nodes in each iteration? What is one way that Dijkstra's would fail with negative edge weights?
3. *Solution Plan:* Create a graph where Dijkstra's will return a path, but there is actually a shorter path using negative edge weights.

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.

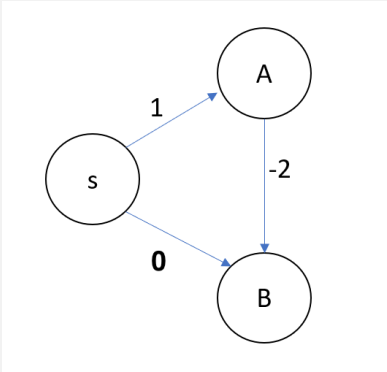
```
Negative−Dijkstra(G, s):
    minWeight = minimum edge weight in G
    for e in E: # iterate through all edges in G
        modifiedWeight(e) = w(e) − minWeight
    modifiedG = G with weights modifiedWeight
    T = Dijkstra(modifiedG, s) # run Dijkstra with modifiedWeight to get a SSSP Tree
```

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.



SOLUTION:

The shortest path from s to B is $S \longrightarrow A \longrightarrow B$, but after we add 2 to each edge, $S \longrightarrow A \longrightarrow B$ will have total weight of 3 whereas $S \longrightarrow B$ will have weight 2, so the solution found by Negative-Dijkstra is will be $S \longrightarrow B$ 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.

Problem Solving Notes:

1. *Read and Interpret:* Are we able to fix Dijkstra's problem with negative edge weights if we add the amount equivalent to the minimum value to all the edges to ensure they are all positive?
2. *Information Needed:* How would this affect Dijkstra's algorithm?
3. *Solution Plan:* See if you can come up with a counterexample of when this might not work.

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.

Problem Solving Notes:

1. *Read and Interpret:* Would the given Negative Prim algorithm return the correct MST or not?
2. *Information Needed:* How does Prim's algorithm work? How would increasing the edge weight by the same amount affect the algorithm?
3. *Solution Plan:* How does Negative-Prim compare with the original Prim algorithm? Can the correctness of the original Prim also work here?