# CS 161 (Stanford, Fall 2025) Section 8 Extra

## 1 The Max-Cut Problem

In this question we will investigate several approaches to the Max-Cut problem. Recall that Max-Cut is the following: given an undirected, unweighted graph $G = (V, E)$, find a partition of the vertices into subsets $S$ and $V \setminus S$ that maximizes the number of edges crossing the cut.

The Max-Cut problem is known to be NP-Hard, so we do not expect a polynomial-time exact algorithm. Parts (1)–(2) will ask you to show why certain natural ideas fail, and in part (3) you will design a greedy approximation algorithm.

1. **Modified Ford–Fulkerson.** Assume all edges have weight 1. Enumerate all pairs $(s, t)$, and for each pair compute the *minimum $s$–$t$ flow*, thinking of it as a way to obtain a Max-Cut (by analogy with the Min-Cut = Max-Flow theorem).

   *Find a counterexample demonstrating that this approach does* **not** *compute a Max-Cut, and explain why it fails.*

   Solution

   **Counterexample:** In any graph, for any choice of $s$ and $t$, the minimum $s$–$t$ flow is always the trivial flow that sends 0 units across every edge. This is strictly smaller than any nonzero feasible flow, so the algorithm always outputs the same meaningless value regardless of the graph structure.
   Thus, unlike Max-Flow/Min-Cut, there is no "Min-Flow = Max-Cut" relationship. Computing a minimum flow does not reveal anything about the maximum cut.

   **Why this is a counterexample:** Even on a simple graph such as a triangle, the minimum flow is 0, but the maximum cut is 2. The algorithm therefore fails to distinguish different graphs or cuts and cannot recover a Max-Cut.

2. **Modified BFS.** Initialize sets $S_1$ and $S_2$ to be empty. Start BFS from an arbitrary node, placing it in $S_1$. Whenever BFS discovers a new node, place it in the *opposite* set from its parent. Return $\{S_1, S_2\}$ as the cut.

   *Find a counterexample showing that this BFS-based partition does* **not** *necessarily give a Max-Cut.*

   Solution

   **Counterexample:** Consider the complete graph $K_5$ on vertices $\{A, B, C, D, E\}$. If BFS starts at any vertex $v$, the algorithm places $v$ in $S_1$ and all other nodes in

$S_2$. The cut then contains only 4 crossing edges (those incident to $v$).
However, a maximum cut in $K_5$ is achieved by splitting the vertices as

$$S_1 = \{A, B\}, \quad S_2 = \{C, D, E\},$$

which yields 6 crossing edges. Thus the BFS partition is far from optimal.

**Why this is a counterexample:** BFS levels depend only on graph distance, not on maximizing edge crossings. In dense graphs the BFS layering forces almost all vertices into the same set, resulting in a small cut.

3. **Greedy Approximation Algorithm.**

Since Max-Cut is NP-Hard, we do not expect a polynomial-time algorithm that always finds the optimal cut. However, we can efficiently compute a cut whose size is at least a 1/2-approximation of the maximum.

Design a greedy algorithm running in $O(m + n)$ time that always returns a cut of size at least $\frac{1}{2} \cdot \text{OPT}$, where OPT is the size of the maximum cut. Provide an English description, an informal correctness justification, and a runtime analysis.

Solution

**Algorithm:** Initialize two empty sets $S_1$ and $S_2$. Process the vertices in any order. For each vertex $v$, count how many of its already-processed neighbors lie in $S_1$ and how many lie in $S_2$. Place $v$ into the set containing *fewer* of its neighbors.

**Why it is a $1/2$-approximation:** When we process a vertex $v$, all edges incident to previously processed neighbors become "decided." By placing $v$ on the side with fewer neighbors, we ensure that at least half of these edges cross the cut (since one side has at most half of them). Summing this over all vertices shows that at least half of all edges in the graph cross the final cut.
Since the maximum cut can include at most all $m$ edges, this yields a 1/2-approximation.

**Runtime:** For each vertex $v$, counting neighbors in each set takes $O(\deg(v))$. Summing over all vertices gives

$$\sum_{v \in V} O(\deg(v)) = O(m),$$

and we perform $O(1)$ extra work per vertex. Thus the total runtime is $O(m+n)$.

# 2 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 each person should either pay or receive money (not both). In other words, you cannot have everyone that owes money pay one person, and that person distributes the money back to everyone that is owed money.

**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 $\infty$ 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.

# 3 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 Prim?

Since Prim's algorithm is very similar to Dijkstra, we want to now consider a similar algorithm *Negative-Prim* for computing minimum spanning trees 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 is 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 each edge $e \in E$ (iterate through all edges in $G$), set

$$modifiedWeight(e) = w(e) - minWeight.$$

- Let $modifiedG$ be $G$ with weights $modifiedWeight$.
- $T$ = Prim($modifiedG, s$) (run Prim's algorithm starting from $s$).

- Update $T$ with edges that correspond 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

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 produced by Negative-Prim is identical to the original Prim algorithm. Since we know Prim's algorithm is correct in graphs 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 notes 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?