COS 330: Great Ideas in Theoretical Computer Science

Fall 2026

Problem Set 5

Module: Optimization and Approximation

Below is a reminder of key aspects of the PSet:

- **The only goal of this PSet is to help you develop your problem-solving skills.** Your performance on this PSet will not directly contribute to your grade, but will indirectly improve your ability to do well on the exams.
- **Your performance does not directly impact your grade**, so you may use any resources you like (collaboration, AI, etc.) to help you complete the PSet.
- **AI is not equally welcome in every part of this PSet**, and we say so explicitly where it matters. Problems 1 and 2 (except part e) are exam-style and should be attempted alone first. Problem 3 explicitly calls for AI assistance in part (d).
- **None of this is an integrity rule** – PSets are not graded directly. It's advice about what will actually develop your skills and in the process prepare you well for the exams, which are closed-book and closed-AI.
- **See the course website** for a full walkthrough of AI tools and recommendations, as well as the input files needed for Problem 3.
- **An AI-helper is available for Problem 1, Problem 2(a)–(d), Problem 3(a)–(c)** – unlike a general assistant, it gives hints and pushes back instead of handing you the answer. See the link above for more information.

Aligning Expectations

The 🗎 symbol marks an exam-style problem. We highly recommend writing up a full solution as if you were on an exam.

Problem 1 : Another Binary String Problem 🗎

You are given a binary string $s = s[1]s[2]\cdots s[n]$, where $n \geq 1$, and a positive integer C. We want to cut the string into nonempty contiguous segments and make all bits within each segment equal. Each segment costs C, plus one for each bit that must be changed.

A segment $S = [L, R]$ consists of the positions $L, L+1, \ldots, R$, whose bits are $s[L], s[L+1], \ldots, s[R]$. The fewest bit changes needed to make this segment all zeros or all ones is

$$\text{err}(S) = \min\{\#\text{zeros in } S,\ \#\text{ones in } S\},$$

where #zeros in S means the number of zeros in segment S, and #ones in S means the number of ones in that segment. Its cost is $C + \text{err}(S)$. A partition must cover the entire string, with no gaps or overlaps. Our goal is to choose the cuts to minimize the total cost: for a partition into $S_1, \ldots, S_k$, this is

$$kC + \sum_{i=1}^{k} \text{err}(S_i).$$

Consider the following **greedy algorithm**, which keeps adding bits to the current segment as long as its error stays at most C:

1. Start at the first uncovered position L (initially $L = 1$), with a one-bit segment $[L, L]$. Set $R = L$.
2. While $R < n$ and $\text{err}([L, R+1]) \leq C$, increase R by one. Thus, the next bit is added only if the enlarged segment still has error at most C.
3. Output the segment $[L, R]$. If $R = n$, stop. Otherwise, set $L = R+1$ and return to step 1 to start a new segment.

By maintaining the counts of zeros and ones in the current segment, this algorithm runs in $O(n)$ time. The threshold C is a choice made by greedy: a valid partition may contain segments with error greater than C.

For example, when $C = 1$ and $s = 011011$, greedy cuts the string as $011 \mid 011$. Each segment has error 1, so its total cost is $(1+1)+(1+1) = 4$. An optimal partition keeps the whole string as one segment: its error is 2, giving total cost $1+2 = 3$.

Write $G_1, \ldots, G_g$ for greedy's segments, in order, so g is the number of greedy segments. Write $B_1, \ldots, B_b$ for the segments of an optimal partition, so b is the number of optimal segments ("B" stands for "best"). The following parts establish that greedy's total cost is at most twice the optimal total cost.

(a) Show that

$$gC \leq \sum_{j=1}^{g} \text{err}(G_j) + C.$$

Approaching the Problem

The main difficulty of part (a) is to understand all the definitions and notation. Take some time to really process all the definitions. Look at the small example and make sure you understand the costs. Write a formula for the total cost of greedy and the total cost of an optimal solution (note that there is a formula for the cost of a partition in the problem statement). Only after you do all of this should you start thinking about how to prove the inequality.

Small Hint

Every segment except possibly the last is maximal. How much can the error increase when you append one bit?

(b) Let S be a segment, and let $S_1, \ldots, S_t$ be nonempty, non-overlapping segments contained in S. They need not cover all of S. Show that

$$\sum_{u=1}^{t} \mathrm{err}(S_u) \;\leq\; \mathrm{err}(S).$$

(c) Show that

$$\sum_{j=1}^{g} \mathrm{err}(G_j) \leq (b-1)C + \sum_{j=1}^{b} \mathrm{err}(B_j).$$

Small Hint

Partition the G_j intervals into two groups: those fully contained within a single optimal segment B_i, and those crossing a boundary between optimal segments. Analyze the groups separately, using part (b).

(d) Conclude that the greedy algorithm is a 2-approximation. In other words, show that the total cost of the greedy solution is at most twice the total cost of an optimal solution.

Small Hint

You need to combine the results from parts (a) and (c) to get a bound on the greedy cost in terms of the optimal cost.

Problem 2 : A Worse Approximation

Exam-Style – Parts (a)–(d)

Attempt parts (a)–(d) alone first and write up full solutions as if you were on an exam. Part (e) is an optional challenge.

Recall the Vertex Cover problem which you saw in class. Given a simple undirected graph $G = (V, E)$ (with $|V| = n \geq 1$ and $|E| = m$), a vertex cover is a subset of vertices $C \subseteq V$ such that for every edge $(u, v) \in E$, at least one of u or v is in C. The goal is to find a vertex cover of minimum size.

Consider the following greedy algorithm, GREEDY, for the vertex cover problem:

- Initialize the vertex cover C to be the empty set.
- While there are edges remaining in the graph:
  - Pick the vertex v with the maximum degree (i.e., the largest number of incident edges), breaking ties arbitrarily.
  - Add v to the vertex cover C.
  - Remove v and all edges incident to v from the graph.

(a) Prove that GREEDY produces a vertex cover of G.

(b) In class, we saw the following 2-approximation, ALG:

- Find any maximal matching in G, M (M is a set of edges).
- Add both endpoints of M to the vertex cover.

For every even $n > 1$, construct an example of a graph with n vertices where the size of the vertex cover produced by GREEDY is exactly half the size of the vertex cover produced by ALG.

(c) Assume $m > 0$, and let OPT be the size of a minimum vertex cover of G. Let m_i be the number of edges remaining after i iterations of GREEDY, so $m_0 = m$. Once GREEDY stops, set $m_i = 0$ for all subsequent i.

(i) Show that whenever $m_i > 0$, the vertex selected in the next iteration has degree at least m_i/OPT in the remaining graph.

(ii) Use (i) to show that, for every $i \geq 0$,

$$m_i \leq m\left(1 - \frac{1}{\text{OPT}}\right)^i.$$

(d) Show that GREEDY has an approximation ratio of $1 + 2\ln n$. That is, show that for every graph, if that graph admits a vertex cover of size K, GREEDY produces a vertex cover of size at most $(1 + 2\ln n) \cdot K$.

Small Hint

Use $1 - x \leq e^{-x}$ for $x \in [0, 1]$.

Optional – Challenge: Part (e)

This part is optional and considerably harder than the Problems 1 and the rest of problem 2 – we don't suggest attempting it just to practice for the exam. It's here for those who want to engage more deeply with the material; if you're considering an IW/thesis in CS theory, it'll give you a taste of what that might be like. Feel free to discuss it with your TA/UCA coach.

Do not use AI to solve this problem. This is a challenge problem, so if you want to work on it you should do so on your own.

(e) Prove that GREEDY does not guarantee a $o(\log n)$-approximation. That is, construct a family of graphs where there is a vertex cover of size $f(n)$, but GREEDY produces a vertex cover of size $g(n)$, and $g(n)/f(n) = \Omega(\log n)$.

Problem 3 : Making the Cut

Suppose we have n tasks to work on, and we will work on them in two batches, an early batch and a late batch. Some tasks get easier if other tasks are completed first, so we have preferences for which tasks should be early and which should be late. These preferences can conflict, so our goal is to accommodate as many of them as possible.

To make this formal, consider a directed graph $G = (V, E)$ with $n \geq 1$ vertices, corresponding to tasks, and m edges, corresponding to preferences. An edge (u, v) represents a preference for u to be early and v to be late. There are no self-loops or repeated edges, but both (u, v) and (v, u) may be present. A division is specified by the early batch $S \subseteq V$ (the late batch is $V \setminus S$). Its value is

$$c(S) = |\{(u, v) \in E : u \in S,\ v \notin S\}|.$$

An edge going from the late batch to the early batch does not count. Batch sizes are unrestricted, and either batch may be empty.

Our goal is to maximize the number of satisfied preferences, in other words to find S that maximizes $c(S)$. Write OPT for the largest possible value of $c(S)$.

(a) Put each vertex in the early batch independently with probability $1/2$, and in the late batch otherwise. Prove that this is a 4-approximation in expectation, i.e., that $\mathbb{E}[c(S)] \geq \text{OPT}/4$.

We will now construct a better polynomial-time approximation algorithm using linear programming.

(b) Let's formulate an LP relaxation of the problem. Assign a variable x_v to each vertex v and a variable z_{uv} to each edge (u, v). We want $x_v = 1$ if v is early and $x_v = 0$ if it is late, and we want

$z_{uv} = 1$ if the preference (u, v) is satisfied and $z_{uv} = 0$ otherwise. Consider the following program:

$$\begin{aligned} \text{maximize} \quad & \sum_{(u,v)\in E} z_{uv} \\ \text{subject to} \quad & z_{uv} \le x_u, \quad z_{uv} \le 1 - x_v && \text{for each } (u,v) \in E, \\ & 0 \le x_v \le 1 && \text{for each } v \in V, \\ & 0 \le z_{uv} \le 1 && \text{for each } (u,v) \in E. \end{aligned}$$

If we restrict x_v and z_{uv} to be 0/1 variables, this program is equivalent to the original problem. Let's relax the integrality constraints and allow x_v and z_{uv} to take any value in $[0, 1]$. Let U be the optimal value of this LP. Prove that

$$\text{OPT} \le U \le m \qquad \text{and} \qquad U \ge \frac{m}{2}.$$

(c) Suppose you solved the LP in (b) and obtained an optimal solution (x^*, z^*). Put each vertex v in the early batch independently with probability

$$p_v = \frac{1}{4} + \frac{x^*_v}{2},$$

and in the late batch otherwise. Prove that $\mathbb{E}[c(S)] \ge U/2$. Deduce that this is a 2-approximation in expectation.

We will now consider a harder generalization of the problem, where a division needs to serve several workloads.

Suppose the input is now $r \ge 1$ directed graphs $G_1 = (V, E_1), \ldots, G_r = (V, E_r)$ on the same vertex set, each with $m_j = |E_j| \ge 1$ and no self-loops or repeated edges. All workloads are known in advance, but we must choose one single division S for all of them. Let $c_j(S)$ be its value in G_j. Our objective is to maximize

$$Q(S) = \min_{1\le j\le r} \frac{c_j(S)}{m_j},$$

the fraction of preferences satisfied in the worst workload.

Aligning Expectations – Using AI in This Question

Part (d) asks you to use AI assistance for implementation and experiments. Decide what you want to implement and measure, then ask the assistant to help carry out those plans. You are responsible for checking that the code implements your methods.

(d) The course website will provide six test cases (`input_01.json` through `input_06.json`), along with loading instructions. Each file contains one collection of four directed workload graphs on a common vertex set. Input 1 has 8 vertices; inputs 2–6 have 96, 96, 192, 96, and 192 vertices, respectively. Use the small case to understand the problem and check your implementation, and the five larger cases to investigate your methods.

First (before using any AI), stare at the LP in (b) and the rounding in (c). See how you can easily adapt it to this multi-workload setting (even if the theoretical guarantees are not obvious).

Then, have an AI assistant implement the LP and rounding. To do this you will need a linear programming solver library (you don't want to implement the LP solver yourself; you also don't want AI to implement the LP solver). There are several free options available; do some research (using AI) and pick one[1].

Treat this method as your baseline. Inspect the larger inputs, using AI to help measure their properties, and design at least one alternative method or modification. Explain what you expect each idea to exploit, then have AI help implement and test it. Your alternatives need not satisfy the guarantees in (a) and (c).

Report what you learned about the inputs, what you tried, how you measured it, and which method you recommend. Include unsuccessful attempts and explain what your experiments do and do not establish. You need not beat the baselines everywhere. Write your explanations in your own words.

[1]Even though AI is doing most of the programming here, the programming language choice is actually important for this. A higher-level language (like Python) will lead to more readable code and easier debugging and testing. These kinds of languages (like Python) also have more resources available for linear programming, which also means the AIs are better at implementing them.