Homework 11: Complexity Theory

Due: December 4th, 2025

Problem 1. Let Maximum-Clique be the following problem: INSTANCE: a graph G (given by an adjacency list), and a number k. QUESTION: does the graph G have a clique of size $\geq k$?

1. Suppose that you have a black-box that solves the Clique problem in $O(1)$ time. Give an efficient algorithm which, for any input graph G, finds the maximum clique in G.
2. Clearly state the (asymptotic) time complexity of the algorithm and the number of queries made to the black-box.

Solution.

1. Algorithm

   We assume access to a black-box $\mathcal{O}$ which, on input (G, k), answers in $O(1)$ time whether G contains a clique of size at least k.

   We want to compute the *maximum* clique size and also output an actual maximum clique.

   To determine the maximum clique size we can perform binary search on $k \in \{1, \ldots, |V|\}$:

   (a) Let $\ell = 1$, $r = |V|$.
   (b) While $\ell < r$:
      - Let $m = \lfloor (\ell + r + 1)/2 \rfloor$.
      - Query $\mathcal{O}(G, m)$.
      - If the oracle answers YES, set $\ell = m$; otherwise set $r = m - 1$.
   (c) Output $K_{\max} = \ell$.

   After binary search, $K_{\max}$ is the size of a maximum clique.

   To construct this maximum size clique we can do this greedily by trying to include each vertex when possible.

   Initialize $C = \emptyset$, and let V' be the vertex set. For each vertex $v \in V$:

   - Temporarily set $C' = C \cup \{v\}$.
   - Let G' be the induced subgraph on the vertices adjacent to all vertices in C'.
   - Query $\mathcal{O}(G', K_{\max} - |C'|)$.
   - If YES, update $C := C'$ and continue; otherwise discard v.

   At the end, C is a clique of size $K_{\max}$.

   For the analysis: The idea is that the oracle solves a decidable decision problem, collapsing an exponential branching process down to one decision per vertex.

- Binary search on k makes $O(\log n)$ oracle queries.
- The reconstruction phase tests each of the n vertices once, and each test calls the oracle exactly once.
- Thus the total number of oracle queries is

$$O(n + \log n) = O(n).$$

- All other computation (checking neighbors, forming induced subgraphs conceptually) runs in polynomial time.

Because the oracle runs in $O(1)$ time, the total running time of the algorithm is

$$O(n(n + m)) = O(n + \text{poly}(n)) = \text{poly}(n).$$

□

Problem 2. Let G be a complete weighted graph in a metric space.

1. A Minimum Bottleneck Spanning Tree (ST) in G, $MBST(G)$, is a spanning tree that minimizes the maximum edge weight:

$$\text{MBST}(G) = \arg \min_{T \in \mathcal{T}(G)} \max_{e \in T} w(e).$$

   Show that a minimum total-weight spanning tree (an MST) in G is also a minimum bottleneck spanning tree.

2. A Minimum Bottleneck TSP in G, $MBTSP(G)$, is a tour that visits each vertex exactly once and minimizes the maximum edge weight on the tour. Design a 3-approximation algorithm for the Minimum Bottleneck TSP.

Solution.

1. MST must also be an MBST by the following: Let T be an MST of G. Let e^* denote the heaviest edge in T:

$$w(e^*) = \max_{e \in T} w(e).$$

   We argue by contradiction. Suppose T is *not* a minimum bottleneck spanning tree. Then there exists some spanning tree T' such that

$$\max_{e \in T'} w(e) < w(e^*).$$

   Consider adding e^* to T'. This creates a cycle C. Since every edge in T' has weight smaller than $w(e^*)$, all edges in $C \setminus \{e^*\}$ have weight strictly less than $w(e^*)$.

   But in the MST T, if we remove e^*, T disconnects into two components; all edges crossing this cut have weight *at least* $w(e^*)$ (by the Cut Property of MSTs). Yet in T', the cycle contains an edge crossing this same cut with strictly smaller weight than $w(e^*)$, contradicting the Cut Property for the MST T.

   Thus no tree can have strictly smaller bottleneck than T, and the MST is an MBST.

2. The construction of the 3-approximation for MBTSP:

   Let T be an MST of G. Let

$$b = \max_{e \in T} w(e)$$

   be its bottleneck value, which by part (1) is the optimal bottleneck for any spanning structure.

   We build a TSP tour using preorder traversal of the MST. The standard doubling-tree algorithm gives a tour H of total length at most $2 \cdot \text{MST}(G)$, but we care about bottleneck, not total weight.

   We show that every edge of H has weight at most $3b$.

   First, consider any step in the preorder walk from vertex u to vertex v:

- If (u, v) is an edge of T, then its weight is at most b.
- If we "jump" directly from u to v (shortcutting repeated visits), then in the original walk the path between u and v was a simple path in T of at most two edges of weight at most b each:

  $$d(u, v) \leq d(u, \text{LCA}) + d(\text{LCA}, v) \leq 3b,$$

  using the triangle inequality that for any three vertices x, y, z in the graph, the direct path from x to z can never be longer than a detour through y, and the fact that G is metric.

Thus every shortcut edge has weight at most $3b$.

Since any TSP solution must have bottleneck at least b, we obtain a 3-approximation:

$$\max_{e \in H} w(e) \leq 3b \leq 3 \cdot (\text{optimal bottleneck value}).$$

□

Problem 3. A HITTING-SET problem is defined on a set U and a collection of subsets $S_1, \ldots, S_n \subseteq U$. The goal is to find the smallest subset $T \subseteq U$ such that $T \cap S_i \neq \emptyset$ for all i.

Design a polynomial-time $O(\log n)$-approximation for the HITTING-SET problem.

Solution. We design the greedy algorithm by repeatedly choosing an element $u \in U$ that hits the largest number of currently-unhit sets S_i.

Initialize $T = \emptyset$ and mark all sets S_i as unhit.

Repeat:

1. Let u be an element of U appearing in the largest number of unhit sets.
2. Add u to T.
3. Mark all sets containing u as hit.

Stop when all sets are hit.

Hitting-set is equivalent to set cover if we view each element $u \in U$ as "covering" those sets S_i in which u appears. The greedy algorithm above is exactly the standard greedy algorithm for set cover.

Greedy achieves an approximation ratio of H_n ($n-$th harmonic number):

$$H_n = 1 + \frac{1}{2} + \cdots + \frac{1}{n} = O(\log n).$$

Thus the greedy solution T satisfies

$$|T| \leq O(\log n) \cdot |T_{\mathrm{OPT}}|.$$

The running time is polynomial, since each greedy choice can be implemented in $O(|U| + \sum |S_i|)$ time with appropriate data structures.

□