35 Approximation Algorithms

35.1 The vertex-cover problem

35.1-1

> Give an example of a graph for which $\text{APPROX-VERTEX-COVER}$ always yields a suboptimal solution.

(Omit!)

35.1-2

> Prove that the set of edges picked in line 4 of $\text{APPROX-VERTEX-COVER}$ forms a maximal matching in the graph G.

(Omit!)

35.1-3 ⋆

> Professor Bündchen proposes the following heuristic to solve the vertex-cover problem. Repeatedly select a vertex of highest degree, and remove all of its incident edges. Give an example to show that the professor's heuristic does not have an approximation ratio of 2. ($Hint:$ Try a bipartite graph with vertices of uniform degree on the left and vertices of varying degree on the right.)

Consider a bibartite graph with left part L and right part R such that L has 5 vertices of degrees $(5, 5, 5, 5, 5)$ and R has 11 vertices of degrees $(5, 4, 4, 3, 2, 2, 1, 1, 1, 1, 1)$ (the graph is easy to draw and the figure is omitted here).

Clearly there exists a vertex-cover of size 5 (the left vertices). The idea is to show that the proposed algorithm chooses all the vertices on the right part, resulting in the approximation ratio of $11 / 5 > 2$.

1. After choosing the first vertex in R, the degrees on L decrease to $(4, 4, 4, 4, 4)$.
2. After choosing the second vertex in R, the degrees on L decrease to $(4, 3, 3, 3, 3)$.
3. After choosing the third vertex in R, the degrees on L decrease to $(3, 3, 2, 2, 2)$.
4. After choosing the fourth vertex in R, the degrees on L decrease to $(2, 2, 2, 2, 1)$.
5. After choosing the fifth vertex in R, the degrees on L decrease to $(2, 2, 1, 1, 1)$.
6. After choosing the sixth vertex in R, the degrees on L decrease to $(1, 1, 1, 1, 1)$.

Now the algorithm still has to choose 5 more vertices.

35.1-4

> Give an efficient greedy algorithm that finds an optimal vertex cover for a tree in linear time.

(Omit!)

35.1-5

> From the proof of Theorem 34.12, we know that the vertex-cover problem and the NP-complete clique problem are complementary in the sense that an optimal vertex cover is the complement of a maximum-size clique in the complement graph. Does this relationship imply that there is a polynomial-time approximation algorithm with a constant approximation ratio for the clique problem? Justify your answer.

(Omit!)

35.2 The traveling-salesman problem

35.2-1

> Suppose that a complete undirected graph $G = (V, E)$ with at least 3 vertices has a cost function c that satisfies the triangle inequality. Prove that $c(u, v) \geq 0$ for all $u, v \in V$.

Let $u, v \in V$. Let w be any other vertex in V. Then by the triangle inequality we have

$$\begin{aligned} c(w, u) + c(u, v) &\geq c(w, v) \\ c(w, v) + c(v, u) &\geq c(w, u). \end{aligned}$$

Adding the above inequalities together gives

$$\backslash\text{cancel}c(w, u) + \backslash\text{cancel}c(w, v) + c(u, v) + c(v, u) \geq \backslash\text{cancel}c(w, v) + \backslash\text{cancel}c(w, u).$$

Since G is undirected we have $c(u, v) = c(v, u)$, and so the above inequality gives $2c(u, v) \geq 0$ from which the result follows.

35.2-2

> Show how in polynomial time we can transform one instance of the traveling-salesman problem into another instance whose cost function satisfies the triangle inequality. The two instances must have the same set of optimal tours. Explain why such a polynomial-time transformation does not contradict Theorem 35.3, assuming that $P \neq NP$.

(Omit!)

35.2-3

> Consider the following ***closest-point heuristic*** for building an approximate traveling-salesman tour whose cost function satisfies the triangle inequality. Begin with a trivial cycle consisting of a single arbitrarily chosen vertex. At each step, identify the vertex u that is not on the cycle but whose distance to any vertex on the cycle is minimum. Suppose that the vertex on the cycle that is nearest u is vertex v. Extend the cycle to include u by inserting u just after v. Repeat until all vertices are on the cycle. Prove that this heuristic returns a tour whose total cost is not more than twice the cost of an optimal tour.

(Omit!)

35.2-4

> In the ***bottleneck traveling-salesman problem***, we wish to find the hamiltonian cycle that minimizes the cost of the most costly edge in the cycle. Assuming that the cost function satisfies the triangle inequality, show that there exists a polynomial-time approximation algorithm with approximation ratio 3 for this problem. (Hint: Show recursively that we can visit all the nodes in a bottleneck spanning tree, as discussed in Problem 23-3, exactly once by taking a full walk of the tree and skipping nodes, but without skipping more than two consecutive intermediate nodes. Show that the costliest edge in a bottleneck spanning tree has a cost that is at most the cost of the costliest edge in a bottleneck hamiltonian cycle.)

(Omit!)

35.2-5

> Suppose that the vertices for an instance of the traveling-salesman problem are points in the plane and that the cost $c(u, v)$ is the euclidean distance between points u and v. Show that an optimal tour never crosses itself.

(Omit!)

35.3 The set-covering problem

35.3-1

> Consider each of the following words as a set of letters: $\{\text{arid, dash, drain, heard, lost, nose, shun, slate, snare, thread}\}$. Show which set cover GREEDY-SET-COVER produces when we break ties in favor of the word that

> appears first in the dictionary.

(Omit!)

35.3-2

> Show that the decision version of the set-covering problem is NP-complete by reducing it from the vertex-cover problem.

(Omit!)

35.3-3

> Show how to implement GREEDY-SET-COVER in such a way that it runs in time $O\left(\sum_{S\in\square}|S|\right)$.

(Omit!)

35.3-4

> Show that the following weaker form of Theorem 35.4 is trivially true:
>
> $$|\square| \le |\square^*| \max\{|S| : S \in \square\}.$$

(Omit!)

35.3-5

> GREEDY-SET-COVER can return a number of different solutions, depending on how we break ties in line 4. Give a procedure $\text{BAD-SET-COVER-INSTANCE}(n)$ that returns an n-element instance of the set-covering problem for which, depending on how we break ties in line 4, GREEDY-SET-COVER can return a number of different solutions that is exponential in n.

(Omit!)

35.4 Randomization and linear programming

35.4-1

> Show that even if we allow a clause to contain both a variable and its negation, randomly setting each variable to 1 with probability $1/2$ and to 0 with probability $1/2$ still yields a randomized $8/7$-approximation algorithm.

(Omit!)

35.4-2

> The ***MAX-CNF satisfiability problem*** is like the MAX-3-CNF satisfiability problem, except that it does not restrict each clause to have exactly 3 literals. Give a randomized 2-approximation algorithm for the MAX-CNF satisfiability problem.

(Omit!)

35.4-3

> In the MAX-CUT problem, we are given an unweighted undirected graph $G = (V, E)$. We define a cut $(S, V - S)$ as in Chapter 23 and the ***weight*** of a cut as the number of edges crossing the cut. The goal is to find a cut of maximum weight. Suppose that for each vertex v, we randomly and independently place v in S with probability $1/2$ and in $V - S$ with probability $1/2$. Show that this algorithm is a randomized 2-approximation algorithm.

\gdef\ExE \gdef\ProbP \gdef\optopt

We first rewrite the algorithm for clarity.

```
APPROX-MAX-CUT(G)
    for each v in V
        flip a fair coin
        if heads
            add v to S
        else
            add v to V - S
```

This algorithm clearly runs in linear time. For each edge $(u, v) \in E$, define the event A_{uv} to be the event where edge (i, j) crosses the cut $(S, V - S)$, and let $1_{A_{uv}}$ be the indicator random variable for A_{uv}.

The event A_{ij} occurs if and only if the vertices u and v are placed in different sets during the main loop in APPROX-MAX-CUT. Hence,

$$\begin{aligned} \text{\Prob}\{A_{uv}\} & = \text{\Prob}\{u \in S \wedge v \in V - S\} + P\{u \in V - S \wedge v \in S\} \\ & = \frac{1}{2} \cdot \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{2} \\ & = \frac{1}{2}. \end{aligned}$$

Let $\text{\opt}$ denote the cost of a maximum cut in G, and let $c = |(S, V - S)|$, that is, the size of the cut produced by APPROX-MAX-CUT. Clearly $c = \sum_{(u,v) \in E} 1_{A_{uv}}$. Also, note that $\text{opt} \le |E|$ (this is tight iff G is bipartite). Hence,

$$\begin{aligned} \text{\Ex}[c] & = \text{\Ex}\left[\sum_{(u,v)\in E} 1_{A_{uv}}\right] \\ & = \sum_{(u,v)\in E} \text{\Ex}[1_{A_{uv}}] \\ & = \sum_{(u,v)\in E} \text{\Prob}\{A_{uv}\} \\ & = \frac{1}{2}|E| \\ & \ge \frac{1}{2}\text{\opt} \end{aligned}$$

Hence, $\text{\Ex}\left[\frac{\text{\opt}}{c}\right] \le \frac{|E|}{1/2|E|} = 2$, and so APPROX-MAX-CUT is a randomized 2-approximation algorithm.

35.4-4

> Show that the constraints in line (35.19) are redundant in the sense that if we remove them from the linear program in lines (35.17)–(35.20), any optimal solution to the resulting linear program must satisfy $x(v) \le 1$ for each $v \in V$.

(Omit!)

35.5 The subset-sum problem

35.5-1

> Prove equation (35.23). Then show that after executing line 5 of EXACT-SUBSET-SUM, L_i is a sorted list containing every element of P_i whose value is not more than t.

(Omit!)

35.5-2

> Using induction on i, prove inequality (35.26).

(Omit!)

35.5-3

> Prove inequality (35.29).

(Omit!)

35.5-4

> How would you modify the approximation scheme presented in this section to find a good approximation to the smallest value not less than t that is a sum of some subset of the given input list?

(Omit!)

35.5-5

> Modify the APPROX-SUBSET-SUM procedure to also return the subset of S that sums to the value z^*.

(Omit!)

Problem 35-1 Bin packing

> Suppose that we are given a set of n objects, where the size s_i of the ith object satisfies $0 < s_i < 1$. We wish to pack all the objects into the minimum number of unit-size bins. Each bin can hold any subset of the objects whose total size does not exceed 1.
>
> **a.** Prove that the problem of determining the minimum number of bins required is NP-hard. $(\text{Hint:}$ Reduce from the subset-sum problem.)
>
> The ***first-fit*** heuristic takes each object in turn and places it into the first bin that can accommodate it. Let $S = \sum_{i=1}^n s_i$.
>
> **b.** Argue that the optimal number of bins required is at least $\lceil S \rceil$.
>
> **c.** Argue that the first-fit heuristic leaves at most one bin less than half full.
>
> **d.** Prove that the number of bins used by the first-fit heuristic is never more than $\lceil 2S \rceil$.
>
> **e.** Prove an approximation ratio of 2 for the first-fit heuristic.
>
> **f.** Give an efficient implementation of the first-fit heuristic, and analyze its running time.

(Omit!)

Problem 35-2 Approximating the size of a maximum clique

> Let $G = (V, E)$ be an undirected graph. For any $k \ge 1$, define $G^{(k)}$ to be the undirected graph $(V^{(k)}, E^{(k)})$, where $V^{(k)}$ is the set of all ordered k-tuples of vertices from V and $E^{(k)}$ is defined so that $(v_1, v_2, \ldots, v_k)$ is adjacent to $(w_1, w_2, \ldots, w_k)$ if and only if for $i = 1, 2, \ldots, k$, either vertex v_i is adjacent to w_i in G, or else $v_i = w_i$.
>
> **a.** Prove that the size of the maximum clique in $G^{(k)}$ is equal to the kth power of the size of the maximum clique in G.
>
> **b.** Argue that if there is an approximation algorithm that has a constant approximation ratio for finding a maximum-size clique, then there is a polynomial-time approximation scheme for the problem.

(Omit!)

Problem 35-3 Weighted set-covering problem

> Suppose that we generalize the set-covering problem so that each set S_i in the family □ has an associated weight w_i and the weight of a cover □ is $\sum_{S_i \in □} w_i$. We wish to determine a minimum-weight cover. (Section 35.3 handles the case in which $w_i = 1$ for all i.)
>
> Show how to generalize the greedy set-covering heuristic in a natural manner to provide an approximate solution for any instance of the weighted set-covering problem. Show that your heuristic has an approximation ratio of $H(d)$, where d is the maximum size of any set S_i.

(Omit!)

Problem 35-4 Maximum matching

> Recall that for an undirected graph G, a matching is a set of edges such that no two edges in the set are incident on the same vertex. In Section 26.3, we saw how to find a maximum matching in a bipartite graph. In this problem, we will look at matchings in undirected graphs in general (i.e., the graphs are not required to be bipartite).
>
> **a.** A ***maximal matching*** is a matching that is not a proper subset of any other matching. Show that a maximal matching need not be a maximum matching by exhibiting an undirected graph G and a maximal matching M in G that is not a maximum matching. (Hint: You can find such a graph with only four vertices.)
>
> **b.** Consider an undirected graph $G = (V, E)$. Give an $O(E)$-time greedy algorithm to find a maximal matching in G.
>
> In this problem, we shall concentrate on a polynomial-time approximation algorithm for maximum matching. Whereas the fastest known algorithm for maximum matching takes superlinear (but polynomial) time, the approximation algorithm here will run in linear time. You will show that the linear-time greedy algorithm for maximal matching in part (b) is a 2-approximation algorithm for maximum matching.
>
> **c.** Show that the size of a maximum matching in G is a lower bound on the size of any vertex cover for G.
>
> **d.** Consider a maximal matching M in $G = (V, E)$. Let
>
> $$T = \{v \in V : \text{ some edge in } M \text{ is incident on } v\}.$$
>
> What can you say about the subgraph of G induced by the vertices of G that are not in T?
>
> **e.** Conclude from part (d) that $2|M|$ is the size of a vertex cover for G.
>
> **f.** Using parts (c) and (e), prove that the greedy algorithm in part (b) is a 2-approximation algorithm for maximum matching.

(Omit!)

Problem 35-5 Parallel machine scheduling

> In the ***parallel-machine-scheduling problem***, we are given n jobs, $J_1, J_2, \ldots, J_n$, where each job J_k has an associated nonnegative processing time of p_k. We are also given m identical machines, $M_1, M_2, \ldots, M_m$. Any job can run on any machine. A ***schedule*** specifies, for each job J_k, the machine on which it runs and the time period during which it runs. Each job J_k must run on some machine M_i for p_k consecutive time units, and during that time period no other job may run on M_i. Let C_k denote the ***completion time*** of job J_k, that is, the time at which job J_k completes processing. Given a schedule, we define $C_{\max} = \max_{1 \le j \le n} C_j$ to be the ***makespan*** of the schedule. The goal is to find a schedule whose makespan is minimum.

For example, suppose that we have two machines M_1 and M_2 and that we have four jobs J_1, J_2, J_3, J_4, with $p_1 = 2$, $p_2 = 12$, $p_3 = 4$, and $p_4 = 5$. Then one possible schedule runs, on machine M_1, job J_1 followed by job J_2, and on machine M_2, it runs job J_4 followed by job J_3. For this schedule, $C_1 = 2$, $C_2 = 14$, $C_3 = 9$, $C_4 = 5$, and $C_{\max} = 14$. An optimal schedule runs J_2 on machine M_1, and it runs jobs J_1, J_3, and J_4 on machine M_2. For this schedule, $C_1 = 2$, $C_2 = 12$, $C_3 = 6$, $C_4 = 11$, and $C_{\max} = 12$.

Given a parallel-machine-scheduling problem, we let $C^*_{\max}$ denote the makespan of an optimal schedule.

a. Show that the optimal makespan is at least as large as the greatest processing time, that is,

$$C^*_{\max} \ge \max_{1 \le k \le n} p_k.$$

b. Show that the optimal makespan is at least as large as the average machine load, that is,

$$C^*_{\max} \ge \frac{1}{m} \sum_{1 \le k \le n} p_k.$$

Suppose that we use the following greedy algorithm for parallel machine scheduling: whenever a machine is idle, schedule any job that has not yet been scheduled.

c. Write pseudocode to implement this greedy algorithm. What is the running time of your algorithm?

d. For the schedule returned by the greedy algorithm, show that

$$C_{\max} \le \frac{1}{m} \sum_{1 \le k \le n} p_k + \max_{1 \le k \le n} p_k.$$

Conclude that this algorithm is a polynomial-time 2-approximation algorithm.

(Omit!)

Problem 35-6 Approximating a maximum spanning tree

Let $G = (V, E)$ be an undirected graph with distinct edge weights $w(u, v)$ on each edge $(u, v) \in E$. For each vertex $v \in V$, let $\max(v) = \max_{(u,v) \in E} \{w(u, v)\}$ be the maximum-weight edge incident on that vertex. Let $S_G = \{\max(v) : v \in V\}$ be the set of maximum-weight edges incident on each vertex, and let T_G be the maximum-weight spanning tree of G, that is, the spanning tree of maximum total weight. For any subset of edges $E' \subseteq E$, define $w(E') = \sum_{(u,v) \in E'} w(u, v)$.

a. Give an example of a graph with at least 4 vertices for which $S_G = T_G$.

b. Give an example of a graph with at least 4 vertices for which $S_G \ne T_G$.

c. Prove that $S_G \subseteq T_G$ for any graph G.

d. Prove that $w(T_G) \ge w(S_G)/2$ for any graph G.

e. Give an $O(V + E)$-time algorithm to compute a 2-approximation to the maximum spanning tree.

(Omit!)

Problem 35-7 An approximation algorithm for the 0-1 knapsack problem

Recall the knapsack problem from Section 16.2. There are n items, where the ith item is worth v_i dollars and weighs w_i pounds. We are also given a knapsack that can hold at most W pounds. Here, we add the further assumptions that each weight w_i is at most W and that the items are indexed in monotonically decreasing order of their values: $v_1 \ge v_2 \ge \cdots \ge v_n$.

In the 0-1 knapsack problem, we wish to find a subset of the items whose total weight is at most W and whose total value is maximum. The fractional knapsack problem is like the 0-1 knapsack problem, except that we are allowed to take a fraction of each item, rather than being restricted to taking either all or none of each item. If we take a fraction x_i of item i, where $0 \le x_i \le 1$, we contribute $x_i w_i$ to the weight of the knapsack and receive value $x_i v_i$. Our goal is to develop a polynomial-time 2-approximation algorithm for the 0-1 knapsack problem.

In order to design a polynomial-time algorithm, we consider restricted instances of the 0-1 knapsack problem. Given an instance I of the knapsack problem, we form restricted instances I_j, for $j = 1, 2, \ldots, n$, by removing items $1, 2, \ldots, j-1$ and requiring the solution to include item j (all of item j in both the fractional and 0-1 knapsack problems). No items are removed in instance I_1. For instance I_j, let P_j denote an optimal solution to the 0-1 problem and Q_j denote an optimal solution to the fractional problem.

a. Argue that an optimal solution to instance I of the 0-1 knapsack problem is one of $\{P_1, P_2, \ldots, P_n\}$.

b. Prove that we can find an optimal solution Q_j to the fractional problem for instance I_j by including item j and then using the greedy algorithm in which at each step, we take as much as possible of the unchosen item in the set $\{j+1, j+2, \ldots, n\}$ with maximum value per pound v_i/w_i.

c. Prove that we can always construct an optimal solution Q_j to the fractional problem for instance I_j that includes at most one item fractionally. That is, for all items except possibly one, we either include all of the item or none of the item in the knapsack.

d. Given an optimal solution Q_j to the fractional problem for instance I_j, form solution R_j from Q_j by deleting any fractional items from Q_j. Let $v(S)$ denote the total value of items taken in a solution S. Prove that $v(R_j) \ge v(Q_j)/2 \ge v(P_j)/2$.

e. Give a polynomial-time algorithm that returns a maximum-value solution from the set $\{R_1, R_2, \ldots, R_n\}$, and prove that your algorithm is a polynomial-time 2-approximation algorithm for the 0-1 knapsack problem.

(Omit!)