9 Approximation Algorithms

In Ch. 8, we saw that for many problems, it is unlikely that there exists a polynomial-time algorithm that optimally solves the problem. So when these problems arise in the real world, what do we do? One solution is to use an algorithm that is "slow" in theory (e.g., has worst-case running time $\Omega(2^n)$) and hope that it's fast enough in practice.

Another idea is to favor speed over optimality by using an approximation algorithm. A *c-approximation algorithm* always returns a solution whose value ALG is within a factor c of the optimal value OPT.[1] For minimization problems, $c \geq 1$, so ALG satisfies

$$\mathsf{OPT} \leq \mathsf{ALG} \leq c \cdot \mathsf{OPT}.$$

For maximization problems, we flip all three inequality symbols above (so $c \leq 1$ and $\mathsf{OPT} \geq \mathsf{ALG} \geq c \cdot \mathsf{OPT}$). This value of c is often referred to as the *approximation ratio* of the algorithm, because it bounds the ratio between ALG and OPT. In Section 7.2, we saw an LP-based, 2-approximation algorithm for the Vertex Cover problem. In this chapter, we'll see more examples, starting with another 2-approximation algorithm for Vertex Cover.

9.1 Vertex Cover

Problem statement. Perhaps you can recall the Vertex Cover problem: The input is an undirected graph $G = (V, E)$, and our goal is to find a smallest subset of vertices such that every edge is incident to at least one vertex in the subset.

Algorithm. Pick an arbitrary edge $e = \{u, v\}$ and add both of its endpoints to S (initially empty). Remove u, v, e, and all other edges incident to u or v from the graph. Repeat this process until the graph has no edges, and return S.

Theorem 9.1. *The set S is a vertex cover and satisfies $|S| \leq 2 \cdot |\mathsf{OPT}|$.*

Proof. Notice that the edges chosen by the algorithm form a matching (i.e., no two chosen edges share an endpoint). Thus, if the matching has k edges, then $|\mathsf{OPT}| \geq k$ because each vertex in OPT can cover at most one edge in the matching. Since S contains two vertices per edge in the matching, we must have $|S| = 2k$, so $|S| \leq 2 \cdot |\mathsf{OPT}|$, as desired. □

9.2 Load Balancing

Problem statement. There are n jobs labeled $\{1, \ldots, n\}$ and m machines. Each job j has a length $\ell_j > 0$. When we assign the jobs to machines, the *load* on a machine is the sum of lengths of jobs assigned to that machine. Our goal is to find an assignment that has the smallest *makespan*, which is defined as the maximum load across all machines.

[1] Note that a 1-approximation algorithm always returns an optimal solution.

Algorithm. The algorithm is greedy: iterate through the set of jobs (in any order), assigning each one to the machine with the smallest load so far.[2]

Theorem 9.2. *The greedy algorithm is a 2-approximation algorithm.*

Proof. Before we prove anything about the algorithm, let us first consider an optimal solution OPT that achieves makespan T^*. Since OPT assigns every job to a machine, including the longest job, we know $T^* \geq \max_j \ell_j$. Also, in any assignment, the average load on a machine is $\sum_j \ell_j/m$. So in OPT, the load on the most-loaded machine is at least the average load, which means $T^* \geq \sum_j \ell_j/m$.

Now we consider our greedy algorithm; suppose it achieves makespan T on some machine i. Let k denote the last job assigned to i, and consider the moment in the algorithm immediately before it assigned k to i. At this point, the load on i was $T - \ell_k$, and because we chose to assign k greedily, this was the smallest load among the m machines. Thus, the total load is at least $m \cdot (T - \ell_k)$. Combining this with our bounds on T^*, we have

$$T - \ell_k \leq \frac{1}{m} \sum_{j=1}^{n} \ell_j \leq T^*,$$

which means $T \leq T^* + \ell_k \leq 2T^*$, as desired. □

9.3 Metric k-Center

Problem statement. Let X denote a set of n points. For any two points u, v, we are given their distance $d(u, v) \geq 0$. (Distances are symmetric, and the distance between any point and itself is 0.) We also know that d satisfies the *triangle inequality*, which states for any three points u, v, w, we have

$$d(u, v) \leq d(u, w) + d(w, v).$$

We are also given $k \geq 1$. We want to select k points in X that act as "centers" of clusters. Once k centers have been selected, the clusters are defined by assigning every point to its closest center. The *radius* of a cluster is the maximum distance from its center to a point in the cluster. Our goal is to select the centers such that the maximum radius is minimized.[3]

Algorithm. The algorithm is a fairly natural greedy algorithm. Pick an arbitrary point as the first center. Then repeat the following $k - 1$ times: find the point whose distance to its closest center is maximized, and add that point as a center.

Theorem 9.3. *The algorithm above is a 2-approximation algorithm.*

[2]Note that this algorithm only needs to see one job at a time, and it never changes its mind. Algorithms with this property are known as *online* algorithms; designing such algorithms is an active area of research.

[3]We can also think of this as placing k balls of the same radius that cover all the points, and we want the radius to be as small as possible.

Proof. Let $S \subseteq X$ denote the algorithm's solution. Also, let S^* and r^* denote an optimal solution and its maximum radius. For any center $y \in S^*$, let $C^*(y)$ denote the set of points in X assigned to y by the optimal solution. Now consider any point $p \in X$, and let t^* denote its closest center in S^*.

1. Suppose the algorithm chose a center $t \in C^*(t^*)$. Then we have

$$d(p,t) \leq d(p,t^*) + d(t^*,t) \leq r^* + r^* = 2r^*,$$

   where the first inequality is the triangle inequality and the second inequality holds because p and t are both assigned to t^* by OPT while r^* denotes the maximum radius in OPT. Thus, the distance from p to its closest center in S is at most $2r^*$.

2. Now suppose the algorithm didn't choose a center in $C^*(t^*)$. By the pigeonhole principle, there must exist some $u^* \in S^*$ such that $C^*(u^*)$ contains (at least) two centers $u_1, u_2 \in S$. Again, we have the following:

$$d(u_1,u_2) \leq d(u_1,u^*) + d(u^*,u_2) \leq r^* + r^* = 2r^*.$$

   Let's assume u_1 was added to S before u_2. When u_2 was added, it was the farthest point in X from its closest center, and we know $d(u_1,u_2) \leq 2r^*$. Thus, all points in X are within $2r^*$ from their closest center, so the maximum radius in S is at most $2r^*$.

In both cases, we have shown that the maximum radius in S is at most $2r^*$, as desired. □

9.4 Maximum Cut

Problem statement. Let $G = (V, E)$ be an undirected graph where each edge e has integral weight $w(e) \geq 1$. For any $S \subseteq V$, let $\delta(S)$ denote the set of edges with exactly one endpoint in S. Also, for any $F \subseteq E$, let $w(F)$ denote the total weight of edges in F. Our goal is to find a cut S that maximizes $w(\delta(S))$.

Algorithm. The algorithm is an example of a *local search* algorithm. Start with an arbitrary cut S. If there exists $u \in V$ such that moving u across the partition $(S, V \setminus S)$ would increase $w(\delta(S))$, then make the move. Repeat this process until no such u exists.[4]

Theorem 9.4. *The algorithm above is a* $1/2$*-approximation algorithm.*

Proof. Consider any vertex $u \in V$ at the end of the algorithm. Let $\alpha(u)$ denote the total weight of edges incident to u that contribute to $w(\delta(S))$. Consider how $w(\delta(S))$ would change if we moved u across the partition $(S, V \setminus S)$. The weight would decrease by $\alpha(u)$ and increase by $w(\delta(u)) - \alpha(u)$. Since the algorithm terminated, the net gain is at most 0, so $\alpha(u) \geq w(\delta(u))/2$.

Now consider summing $\alpha(u)$ over all $u \in V$; this value is exactly $2 \cdot w(\delta(S))$. Similarly, the sum of $w(\delta(u))$ over all $u \in V$ is exactly $2 \cdot w(E)$. Combining these, we have

$$2 \cdot w(\delta(S)) = \sum_{u \in V} \alpha(u) \geq \frac{1}{2} \sum_{u \in V} w(\delta(u)) = w(E) \geq \mathsf{OPT}.$$ □

[4]It might not be immediately clear that this algorithm even terminates; we leave this as an exercise.

9.5 Exercises

9.1. Consider the following greedy algorithm for Vertex Cover: add the vertex with highest degree to the solution, remove the edges incident to this vertex, and repeat this process on the resulting graph until no edges remain. Show that this algorithm does not always return an optimal solution.

9.2. For the load balancing problem (Sec. 9.2), give a 3/2-approximation algorithm that runs in $O(n^2)$ time. (Hint: The algorithm is similar to the greedy algorithm.)

9.3. Show how the algorithm in Sec. 9.3 can be implemented in $O(nk)$ time.

9.4. Prove that the algorithm in Sec. 9.4 always terminates. Then prove that the algorithm terminates in $O(n^4)$ time if every edge has weight 1.

9.5. Consider the greedy algorithm for the load balancing problem from Sec. 9.2.

(a) Show that the approximation ratio is actually at most $2 - 1/m$ (rather than 2).

(b) Give an instance of the problem such that the makespan of ALG is exactly $(2 - 1/m)$ times as large as the makespan of OPT.

9.6. In this exercise, we give another proof of Theorem 9.3.

(a) Let S denote the algorithm's solution, let u denote the point in X farthest from its closest center in S, and let r denote the distance from u to its closest center. Prove that the distance between any two points in S is at least r.

(b) Prove that r is at most twice the optimal radius.

9.7. Consider the following greedy algorithm for Selecting Compatible Intervals: repeatedly add the shortest interval that does not conflict with any interval already added. Prove that this is a 1/2-approximation algorithm.

9.8. Suppose there exists a polynomial-time α-approximation algorithm for the metric k-center problem (Sec. 9.3), where $\alpha < 2$. Show how we can use this algorithm to solve the DomSet problem from Sec. 8.3 in polynomial time.

9.9. Recall that a matching in an undirected graph $G = (V, E)$ is a subset of edges $F \subseteq E$ such that no two edges in F share an endpoint. Give a linear-time 1/2-approximation for the problem of finding a matching of maximum size.

9.10. Recall the maximum independent set problem: given an undirected graph, find a subset of vertices of maximum size such that no two vertices in the subset share an edge. Suppose every vertex in the graph has degree at most Δ. Give a linear-time $1/\Delta$-approximation algorithm.

9.11. Recall the minimum cut problem from Ch. 6. Suppose now there are k *terminal* vertices, and our goal is to find a minimum-capacity subset of edges such that removing these edges separates the k terminals into different connected components. Give a $(2 - 2/k)$-approximation algorithm for this problem. (Hint: Start with the goal of a 2-approximation by using a minimum s-t cut algorithm multiple times.)