Selected Algorithms for 330

Kevin Sun

Preface

This document is primarily written for those who are studying algorithms at a standard undergraduate level. It assumes familiarity with basic programming concepts (e.g., arrays, loops), discrete math (e.g., graphs, induction), and asymptotic notation. The chapters are intended to be read in order, and the last section of each chapter contains exercises that I recommend. To maintain conciseness, I have minimized things like context, applications, and (sometimes) even running time analysis in favor of showcasing the core ideas.

For a more thorough treatment of the material, I recommend the following textbooks. I think they are all very well-written, and this document is heavily inspired by them.

- *Algorithms* by Dasgupta, Papadimitriou, and Vazirani. This book is probably the most conversational one on this list: "instead of dwelling on formal proofs we distilled in each case the crisp mathematical idea that makes the algorithm work."
- *Algorithms* by Erickson.[1] This book has the strongest "eccentric professor" vibes, complete with historical footnotes, color commentary, and wondrous diagrams. The author also has a strong online presence (e.g., on Quora and StackExchange).
- *Algorithm Design* by Kleinberg and Tardos.[2] This book discusses real-world applications without sacrificing mathematical rigor. It also has "Solved Exercises" in each chapter, which discuss the intuition behind formulating an algorithmic solution.
- *Introduction to Algorithms* by Cormen, Leiserson, Rivest, and Stein. This book is widely regarded as the definitive reference for the field of algorithms. It contains a comprehensive set of topics, each of them covered in full mathematical detail.

Of course, there are many wonderful textbooks on algorithms not listed above. There are also resources that focus on algorithms for coding interviews or competitive programming. Although some of those topics (e.g., dynamic programming) overlap with ours, this document is primarily focused on writing proofs and pseudocode rather than actual code.

I hope these notes teach you something, or perhaps jog your memory about something you've learned before. Please feel free to contact me if you have any feedback.

— Kevin Sun
nusnivek@gmail.com
2021

[1] Freely available at `http://algorithms.wtf`.
[2] Slides freely available at `https://www.cs.princeton.edu/~wayne/kleinberg-tardos/`.

1 Basic Graph Algorithms

A graph $G = (V, E)$ consists of a set of vertices and edges. If the graph is directed, then each edge is of the form (u, v); otherwise, an edge between u and v is technically $\{u, v\}$, but people often use (u, v) because it looks a bit nicer (and we can always order the vertices arbitrarily). Unless otherwise stated, every graph we consider in this entire document is connected and has n vertices and m edges.

1.1 Breadth-First Search

Problem statement. Let $G = (V, E)$ be a graph and s be a vertex of G. Our goal is to find the distance (i.e., length of the shortest path) from s to all other vertices of G.

Algorithm. We start at s, visit its neighbors, visit their neighbors (that haven't already been visited), and so on, until every vertex has been visited. Since this procedure finishes each "layer" (defined by distance from s) entirely before proceeding to the next, it is known as *breadth*-first search (BFS).

Algorithm 1 Breadth-First Search (BFS)

```
1: Initialize d[s] = 0 and d[u] = ∞ for all u ∈ V \ {s}.
2: Initialize a queue Q = (s).
3: while Q is not empty do
4:     Dequeue a vertex u from Q.
5:     for each neighbor v of u do                ▷ visit the unvisited neighbors of u
6:         if d[v] = ∞ then
7:             Set d[v] = d[u] + 1 and add v to Q.
8: return d
```

Theorem 1.1. *At the end of Algorithm 1, for every vertex v, $d[v]$ denotes the distance between the starting vertex s and v.*

Proof. Let k denote the distance between s and v; we proceed by induction on k. If $k = 0$, then $s = v$ and $d[s] = 0$, so we are done. Otherwise, by induction, we assume that $d[u] = k-1$ for every vertex u that has distance $k - 1$ from s. Since the algorithm processes vertices layer by layer, there is a point at which the vertices in Q are exactly the ones with distance $k - 1$ from s. At least one such vertex them is a neighbor of v, so we can let u denote the first such vertex dequeued. When u is removed from Q, v still has not been added, so the algorithm sets $d[v] = d[u] + 1 = k - 1 + 1 = k$, as desired. □

Notice that every vertex enters and leaves Q exactly once. Processing a vertex u requires $\deg(u)$ time and $\sum_{u \in V} \deg(u) = 2m$, so the overall running time is $O(m + n) = O(m)$.

1.2 Depth-First Search

Problem statement. Let $G = (V, E)$ be a directed graph. Our goal is to determine whether or not G contains a (directed) cycle; if so, we should return one such cycle.

Remark. Before solving the problem, we first describe the general Depth-First Search (DFS) algorithm. In particular, we show how it labels every vertex u with two positive integers: $\mathsf{pre}[u]$ and $\mathsf{post}[u]$. These values allow us to solve multiple problems, including the cycle detection problem stated above.

The DFS algorithm uses an "explore" procedure (Algorithm 2), which recursively traverses some path until it gets "stuck" because it has already visited all neighbors of the current vertex. It then simply follows another path until it gets stuck again. This process repeats until all vertices have been visited (by possibly restarting the process from another vertex). The algorithm gets its name because it goes as "deep" as possible (away from the starting vertex) before turning around.

Algorithm 2 Explore (a vertex $x \in V$)

```
Mark x as visited, set pre[x] = t, and set t = t + 1.
for each edge (x, y) ∈ E do
    if y has not been marked as visited then
        Set p[y] = x and Explore (i.e., call this algorithm on) y.
Set post[x] = t and t = t + 1.
```

As shown above, DFS also maintains a global "clock" variable t, which is incremented whenever a vertex receives its pre or post value. Every vertex x receives its pre value when the exploration of x begins, and it receives its post value when the exploration ends. Also, when the exploration traverses an edge (x, y) we set $p[y] = x$, indicating that vertex y "came from" x, so x is the "parent" of y.

Algorithm 3 Depth-First Search (DFS)

```
Initialize all vertices as unmarked (i.e., unvisited) and t = 1.
for each vertex u ∈ V do
    if u has not been marked as visited then
        Set p[u] = u and call Algorithm 2 on u.
return pre, post, the parent pointers p
```

So the first vertex considered by DFS gets pre value 1, and the last vertex explored gets post value $2n$. Algorithm 2 is called on every vertex exactly once, and also examines every edge exactly once, so the overall running time of DFS is $O(m + n) = O(m)$. Before solving the cycle detection problem, we present the following definition.

Definition 1.2. *An edge (u, v) in G is a* back edge *if $\mathsf{pre}[v] < \mathsf{pre}[u] < \mathsf{post}[u] < \mathsf{post}[v]$.*

After running DFS, we have the pre and post values for all vertices, so we can easily find a back edge in linear time. Furthermore, every vertex has a parent, and if we orient the explored edges from parent to child, then we see that DFS constructs a forest of rooted trees. (A new tree is created whenever Algorithm 3 explores a new vertex.)

Theorem 1.3. *A directed graph has a cycle if and only if DFS creates a back edge.*

Proof. If (u, v) is a back edge, then while v was being explored, the algorithm discovered u. Since the algorithm only follows edges in the graph, this means there is a path from v to u in the graph. This path, combined with the edge (u, v), is a cycle.

Conversely, suppose the graph has a cycle C. Let v denote the first vertex of C explored by DFS, and let u denote the vertex preceding v in C. Then while v is being explored, u will be discovered, which will lead to (u, v) being a back edge. □

The proof of Theorem 1.3 gives our final algorithm for the cycle detection problem. If DFS does not create any back edges, then the graph is acyclic. Otherwise, if (u, v) is a back edge, then there must be a path from v to u, so adding (u, v) to this path forms a cycle.

1.3 Minimum Spanning Tree

Problem statement. Let $G = (V, E)$ be an undirected graph where each edge e has weight $w_e > 0$. Our goal is to compute $F \subseteq E$ such that (V, F) is connected and the total weight of edges in F is minimized. Any subset of E satisfying these two properties is a *minimum spanning tree* (MST) of G.

Remark. For simplicity, we assume that all of the edge weights in G are distinct. It follows, by a fairly straightforward exchange argument (similar to the proofs in this section), that G has exactly one MST. Thus, wwe will generally refer to *the* (rather than *an*) MST of G.

Algorithm. Instead of giving a single algorithm, we first prove two facts about MSTs known as the cut property and the cycle property. Using these two facts, we can prove the correctness of three natural greedy algorithms for the MST problem.

Lemma 1.4 (Cut Property)**.** *Let S be any subset of V (also known as a* cut*), and let e denote the lightest edge crossing S (i.e., among the edges with exactly one endpoint in S, e has minimum weight). The MST of G contains this edge e.*

Proof. Let T denote the MST of G, and for contradiction, suppose there exists a cut S such that the lightest edge e that crosses S is not in T. Notice that $T \cup \{e\}$ must contain a cycle C that includes e. Since e crosses S, some other edge $f \in C$ must cross S as well, and since e is the lightest edge crossing S, we must have $w(e) < w(f)$. Thus, adding e and removing f from T yields a spanning tree with less total weight than T, and this contradicts our assumption that T is the MST. □

Lemma 1.5 (Cycle Property)**.** *Let C be any cycle in G, and let f denote the heaviest edge in C. The MST of G does not contain this edge f.*

Proof. The proof is similar to that of the cut property. Let T denote the MST of G, and for contradiction, suppose there exists a cycle C whose heaviest edge f belongs to T. Removing f from T disconnects T into two trees, and also defines a cut of G. Some other edge e of C must cross this cut, and since f is the heaviest in C, we have $w(e) < w(f)$. Thus, adding e and removing f from T yields a spanning tree with less total weight than T, and this contradicts our assumption that T is the MST. □

To summarize: the cut property says that the lightest edge crossing any cut is in the MST, and the cycle property says that the heaviest edge in any cycle is not in the MST. As mentioned earlier, using these two properties, we can prove the correctness of multiple algorithms for MST.

1. **Kruskal's algorithm.** Sort the edges in order of increasing weight. Then iterate through the edges in this order, adding each edge e to F (initially empty) if (and only if) $(V, F \cup \{e\})$ remains acyclic.

2. **Prim's algorithm.** Let s be an arbitrary vertex and initialize a set $S = \{s\}$. Repeat the following $n-1$ times: (greedily) add the node v to S that minimizes the "attachment cost" defined as $\min_{(u,v):u \in S} w(u, v)$.[1]

3. **Reverse-Delete.** Sort the edges in order of decreasing weight. Then iterate through the edges in this order, removing each edge e from F (initially all of E) if (and only if) $(V, F \setminus \{e\})$ remains connected. (This is sort of a "backward" version of Kruskal's algorithm.)

Given the cut and cycle properties, the proofs of correctness for all three algorithms are fairly short and straightforward. Below, we prove the correctness of Kruskal's algorithm; we leave the other two proofs as exercises.

Theorem 1.6. *At the end of Kruskal's algorithm, F is a minimum spanning tree.*

Proof. Consider any edge $e = \{u, v\}$ added to F, and let S denote the set of vertices connected to u immediately before e was added. Notice that $u \in S$, $v \in V \setminus S$, and no edge crossing S has been considered by the algorithm yet (because it would have been added). Thus, e is the lightest edge crossing S, so by the cut property, e is in the MST.

Now we know F is a subset of the MST; it suffices to prove that F is spanning. For contradiction, suppose that there exists a cut $S \subset V$ such that no edge crossing S is in F. But the algorithm would have added the lightest edge crossing S, so we are done. □

1.4 Exercises

1. Recall that DFS returns two values $\mathsf{pre}[u]$ and $\mathsf{post}[u]$ for every vertex u of a directed graph. Consider any edge $e = (u, v)$; we saw what it means for e to be a back edge. What do you think should be the definition of a *tree* edge, *forward* edge, and *cross* edge? How is the situation different for undirected graphs?

2. Let G be a directed graph and s be a starting vertex. Give a linear-time algorithm that returns the number of shortest paths between s and all other vertices.[2]

3. An undirected graph $G = (V, E)$ is *bipartite* if there exists a partition of V into two nonempty groups such that no edge of E has one endpoint in each group.

[1] As we will see in the following section, this algorithm is similar to Dijkstra's algorithm.

[2] A *linear-time* algorithm is one whose running time is $O(m+n)$. If the graph is connected, then $n-1 \leq m$, so this bound can be simplified to $O(m)$.

(a) Prove that G is bipartite if and only if G contains no odd cycle.

(b) Give a linear-time algorithm to determine if G is bipartite, and if so, the algorithm should return a valid partition of V (i.e., the algorithm should label each vertex using one of two possible labels).

4. Let G be a directed acyclic graph (DAG). Use DFS to give a linear-time algorithm that orders the vertices such that for every edge (u, v), u appears before v in the ordering. (Such an ordering is known as a *topological sort* of G.)

5. Let G be a connected undirected graph. An edge e is a *bridge* if the graph $(V, E \setminus \{e\})$ is disconnected.

   (a) Give an $O(m^2)$-time algorithm to determine if G has a bridge, and if so, the algorithm should return one.

   (b) Assume G is bridgeless. Give a linear-time algorithm that orients every edge (i.e., turns it into a directed edge) in a way such that the resulting (directed) graph is strongly connected.

6. Let G be a directed graph. We say that vertices u and v are in the same *strongly connected component* (SCC) if they are strongly connected, i.e., there exists a path from u to v and a path from v to u. Consider the graph G^S whose vertices are the SCC's of G and there exists an edge from A to B if (and only if) there exists at least one edge (u, v) in G with $u \in A$ and $v \in B$.

   (a) Show that G^S is a DAG.

   (b) Give a linear-time algorithm to construct G^S using G. (Hint: First run DFS on the *reverse* graph of G to obtain post values. Then, use these post values while running DFS (or BFS) on G.)

7. Recall that the *distance* between two vertices is the length of the shortest path between them. The *diameter* of a graph is the maximum distance between any two vertices.

   (a) Give an $O(mn)$-time algorithm that computes the diameter of a graph.

   (b) Give a linear-time algorithm that computes the diameter of a tree. (Hint: One algorithm involves running BFS twice.)

8. Let G be a graph; assume that G contains a cycle. Our goal is to return the shortest cycle in G. Give an $O(m^2)$-time algorithm assuming G is undirected, and give an $O(mn)$-time algorithm assuming G is directed.

9. Let $G = (V, E)$ be an undirected graph with n vertices. Show that if G satisfies any two of the following properties, then it also satisfies the third: (1) G is connected, (2) G is acyclic, (3) G has $n - 1$ edges.

10. Prove that an undirected graph is a tree if and only if there exists a unique path between each pair of vertices.

11. Prove that a graph has exactly one minimum spanning tree if all edge weights are unique, and show that the converse is not true.

12. Consider the MST problem from Sec. 1.3.

    (a) Prove the correctness of Prim's algorithm, and show that it can be implemented to run in $O(n^2)$ time.

    (b) Prove the correctness of the Reverse-Delete algorithm.

13. Consider the MST problem, but suppose some of the edge weights are negative. Describe how to modify the graph such that an MST in the new graph is also an MST of the original graph.

14. Suppose all of the edges of an undirected graph G have weight $w > 0$. Give a linear-time algorithm that returns an MST of G.

15. Consider the following variant of the MST problem: our goal is to find a spanning tree such that the weight of the heaviest edge in the tree is minimized. In other words, instead of minimizing the sum, we are minimizing the maximum. Prove that a "normal" MST is also optimal for this new problem.

16. Let T be a spanning tree of a graph G (which has unique edge weights). Prove that T is the MST of G if and only if T satisfies the following property: for every edge $e \in E \setminus T$, e is the heaviest edge in the (only) cycle of $T \cup \{e\}$.

2 Greedy Algorithms

In this chapter, we consider optimization problems whose solution can be built iteratively as follows: in each step, there is a natural "greedy" choice, and the algorithm is to repeatedly make that greedy choice. In the analysis, we must show that a solution created this way is indeed an optimal solution to the problem.

Sometimes, there are multiple interpretations for what it means to be greedy, and we will have to figure out which choice is the correct one for the problem. And for many problems, there is no optimal greedy solution. But in this chapter, we will only be concerned with problems for which *some* greedy algorithm works.

2.1 Selecting Compatible Intervals

Problem statement. Let $\{1, \ldots, n\}$ denote a set of n events, where each event i occupies a time interval $[s(i), t(i)]$. A subset of events is *compatible* if no two events overlap in time. Our goal is to select a largest subset of compatible events.

Algorithm. Repeatedly select the event that finishes earliest. More formally, first sort the events by their end time so that $t(1) \leq t(2) \leq \cdots \leq t(n)$. Then add the first event to the solution and traverse the list until we reach a non-conflicting event. Add this event as well, and repeat this process until we've traversed the entire list.

Theorem 2.1. *The algorithm optimally solves the problem of selecting a largest subset of compatible intervals.*

Proof. Suppose our solution returns m events, which we denote by $\mathsf{ALG} = \{i_1, \ldots, i_m\}$. Let $\mathsf{OPT} = \{j_1, \ldots, j_{m^*}\}$ denote an optimal solution, which contains m^* events. Assume that ALG and OPT are both sorted in increasing finishing time, so i_1 is the event that finishes first among all n events. This means $t(i_1) \leq t(j_1)$, so if we only consider i_1 and j_1, ALG is "ahead" of OPT because it finishes at least as early.

In general, we can use induction to show that ALG "stays ahead" of OPT at every step, that is, $t(i_k) \leq t(j_k)$ for every $k \leq m$. We've shown the statement is true for $k = 1$. Assuming it is true for $k - 1$, we have $t(i_{k-1}) \leq t(j_{k-1})$. Since j_k starts after j_{k-1} ends, it also starts after i_{k-1} ends. This means j_k is a candidate after the algorithm picks i_{k-1}, so by the algorithm's greedy choice when picking i_k, we must have $t(i_k) \leq t(j_k)$.

In particular, this means $t(i_m) \leq t(j_m)$. Now, for contradiction, suppose $m^* \geq m + 1$. The event j_{m+1} starts after j_m ends, so it also starts after i_m ends. This means the algorithm could have picked j_{m+1} after picking i_m, but it actually terminated. This contradicts the fact that the algorithm processes the entire list of events. □

2.2 The Fractional Knapsack Problem

Problem statement. Suppose $[n] = \{1, \ldots, n\}$ denote a set of n items, where each item i has weight $w_i > 0$ and value $v_i > 0$. We have a knapsack with capacity B. Our goal is to determine $x_i \in [0, 1]$ for each item i to maximize the total value $\sum_{i=1}^{n} x_i v_i$, subject to the constraint that the total weight $\sum_{i=1}^{n} x_i w_i$ is at most B.

Remark. In the original (i.e., not fractional) knapsack problem, we cannot take a fraction of an item, i.e., we must set $x_i \in \{0, 1\}$ for every item i. For this problem, there is no natural greedy algorithm that always returns an optimal solution. However, it can be solved using a technique known as dynamic programming, as we show in Ch. 3.

Algorithm. Relabel the items in decreasing[1] order of "value per weight," so that

$$\frac{v_1}{w_1} > \frac{v_2}{w_2} > \cdots > \frac{v_n}{w_n}.$$

In this order, take as much of each item as possible (i.e., set each $x_i = 1$) until the total weight of selected items reaches the capacity of the knapsack B.

Theorem 2.2. *The greedy algorithm above returns an optimal solution to the fractional knapsack problem.*

Proof. Let $\mathsf{ALG} = (x_1, \ldots, x_n)$ denote the solution returned by the greedy algorithm, and let $\mathsf{OPT} = (x_1^*, \ldots, x_n^*)$ denote an optimal solution. For contradiction, assume $\mathsf{OPT} > \mathsf{ALG}$; let i denote the smallest index such that $x_i^* \neq x_i$. By design of the algorithm, we must have $x_i^* < x_i$. Furthermore, since $\mathsf{OPT} > \mathsf{ALG}$, there must exist some $j > i$ such that $x_j^* > 0$.

We now modify OPT as follows: increase x_i^* by $\epsilon > 0$ and decrease x_j^* by $\epsilon w_i / w_j$. Note that ϵ can be chosen small enough such that x_i^* and x_j^* remain in $[0, 1]$ after the modification. Let OPT' denote this modified solution; notice that OPT and OPT' have the same total weight so OPT' is feasible. Now consider the value of OPT':

$$\mathsf{OPT}' = \mathsf{OPT} + \epsilon \cdot v_i - \frac{\epsilon w_i}{w_j} \cdot v_j = \mathsf{OPT} + \epsilon \cdot \left(v_i - \frac{w_i}{w_j} \cdot v_j \right) > \mathsf{OPT},$$

where the inequality follows from $\epsilon > 0$ and $j > i$. Thus, OPT' is a feasible solution with total value greater than OPT, contradicting the fact that OPT is an optimal solution. □

2.3 Maximum Spacing Clusterings

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.) The *spacing* of a clustering (i.e., partition) of X is the minimum distance between any pair of points lying in different clusters. We are also given $k \geq 1$, and our goal is to find a clustering of X into k groups with maximum spacing.

Algorithm. Our algorithm is essentially Kruskal's algorithm for Minimum Spanning Tree (Sec. 1.3). In the beginning, each point belongs to its own cluster. We repeatedly add an edge between the closest pair of points in different clusters until we have k clusters. (This is equivalent to computing the MST and removing the $k - 1$ longest edges.)

Theorem 2.3. *The algorithm above returns a clustering with maximum spacing.*

[1] We can assume the v_i / w_i values are unique without loss of generality; we leave the proof as an exercise.

Proof. Let α denote the spacing of the clustering ALG produced by the algorithm, so α is the length of the edge that the algorithm would have added next, but did not. Now consider the optimal clustering OPT. It suffices to show that the spacing of OPT is at most α. If ALG $\neq$ OPT, then there must exist two points $u, v \in X$ such that in ALG, u and v are in the same cluster, but in OPT, u and v are in different clusters.

Now consider the path P between u and v along the edges in ALG. Since the algorithm adds edges in non-decreasing order of length, and the edge corresponding to α was not added, all of the edges in P have length at most α. Furthermore, since u and v lie in different clusters of OPT, there must be some edge on P whose endpoints lie in different clusters of OPT. Thus, the distance between these two clusters is at most α. □

2.4 Stable Matching

Problem statement. Let $G = (X \cup Y, E)$ be a bipartite graph where $|X| = |Y| = n$. The vertices in X represent students, and the vertices in Y represent companies. Each student has a preference list (i.e., an ordering) of the n companies, and each company has a preference list of the n students.

Our goal is to construct a matching function $M \colon Y \to X$ such that each company is matched with exactly one student, and there are no instabilities. An *instability* is defined as a (student, company) pair (x, y) such that y prefers x over $M(y)$, and x prefers y over the company y' such that $M(y') = x$.[2]

Remark. It is not obvious that a stable matching even exists for every possible set of preference lists. In this section, we not only prove this holds, but we do this by presenting an algorithm (known as the Gale-Shapley algorithm) that efficiently returns a stable matching.

Algorithm. If we focus our attention on a particular student x, then the greedy choice to make would be to match that student to their favorite company. So in fact, our algorithm does this in the first round: it assigns every student to their favorite company. If every company receives exactly one applicant, then it's not difficult to show that we'd be done. But if a company receives more than 1 applicant, then it *tentatively* gets matched with that student, and the other applicants return to the pool.

In each subsequent round, an unmatched student applies to their favorite company that hasn't rejected them yet, that company gets tentatively matched with their favorite, and the rejected student (if one exists) returns to the pool. Note that a company never rejects an applicant unless a better one (from the company's perspective) applies.

Theorem 2.4. *Algorithm 4 returns a stable matching.*

Proof. We first show that the algorithm terminates and the output is a matching. Notice that no student applies to the same company more than once, so the algorithm terminates in most n^2 iterations. Now suppose the algorithm ended with an unmatched student x. Whenever a company y receives an applicant, y is (tentatively) matched for the rest of the

[2] If we view X as men and Y as women, then an instability consist of a man and a woman who both wish they could cheat on their respective partners.

Algorithm 4 The Gale-Shapley algorithm

```
Initialize M(y) = ∅ for all y ∈ Y and S = X.
while S is not empty do
    Remove some student x from S.
    Let y denote x's favorite company that hasn't rejected them yet.
    if M(y) = ∅ or y prefers x to M(y) then
        Add M(y) to S and set M(y) = x.          ▷ x and y are tentatively matched
    else
        Return x to S.                           ▷ x is rejected by y
return the matching function M
```

algorithm. Since x must have applied to every company, in the end, every company must be matched. But $|X| = |Y|$, so it cannot be the case that every company is matched while x remains unmatched.

Now we show at M is stable. For contradiction, suppose (x, y) is an instability, where x is matched to some y' but prefers y, and y is matched to $M(y)$ but prefers x. Since x prefers y to y', x must have applied to y at some point. But, since y ends up matched with $M(y)$, y must have rejected x at some point in favor of someone y prefers more. The tentative match for y only improves over time (according to y's preference list), so y must prefer $M(y)$ to x. This contradicts our assumption that y prefers x to $M(y)$, so we are done. □

2.5 Exercises

1. Let $\{1, \ldots, n\}$ denote a set of n customers, where each customer has a distinct processing time t_i. In any ordering of the customers, the *waiting time* of a customer i is the sum of processing times of customers that appear before i in the order. Give an $O(n \log n)$-time algorithm that returns an ordering such that the sum (or average) of waiting times is minimized.

2. Consider the same setup as the problem in Sec. 2.1. Instead of sorting by end time, suppose we sort by non-decreasing start time, duration (i.e., length of the interval), or number of conflicts with other events. For each of these three algorithms, give an instance where the algorithm does not return an optimal solution.

3. Consider the same setup as the problem in Sec. 2.1. Now our goal is to partition the events into groups such that all of the events in a group are compatible, and the number of groups is minimized. Give an $O(n^2)$-time algorithm for this problem.

4. Consider the same setup as the problem in Sec. 2.1. We say a point x *stabs* an interval $[s, t]$ if $s \leq x \leq t$. Give an $O(n \log n)$-time algorithm that returns a minimum set of points that stabs all intervals.

5. Consider the fractional knapsack problem from Sec. 2.2. Prove that the assumption that all v_i/w_i values are unique can be made without loss of generality.

6. Suppose there are an unlimited number of coins in each of the following denominations: 50, 20, 1. The problem input is an integer n, and our goal is to select the smallest number of coins such that their total value is n. The greedy algorithm is to repeatedly select the denomination that is as large as possible.

   (a) Give an instance where the greedy algorithm does *not* return an optimal solution.

   (b) Now suppose the denominations are 10, 5, 1. Show that the greedy algorithm always returns an optimal solution.

7. Consider the stable matching problem described in Sec. 2.4. For any student x and company y, we say that (x, y) is *valid* if there exists a stable matching that matches x and y. For each student x, let $f(x)$ denote the highest company on x's preference such that $(x, f(x))$ is valid. Similarly, for any company y, let $g(y)$ denote the *lowest* valid student on y's preference list such that $(g(y), y)$ is valid.

   (a) Prove that the Gale-Shapley algorithm always matches x with $f(x)$.

   (b) Prove that the Gale-Shapley algorithm always matches y with $g(y)$.

8. An *independent set* of a graph is a subset of vertices such that no two share an edge. Let $T = (V, E)$ be a tree. Give an $O(n)$-time algorithm that returns a maximum independent set (i.e., one with the most vertices) of T. (In Sec. 3.4, we solve the weighted version of this problem.)

9. A *matching* in a graph is a subset of edges such that no two share an endpoint. Let $T = (V, E)$ be a tree. Give an $O(n)$-time algorithm that returns a maximum matching (i.e., one with the most edges) of T. (In Sec. 3.5 Exercise 13, we solve the weighted version of this problem.)

3 Dynamic Programming

In this chapter, we study an algorithmic technique known as dynamic programming (DP). The idea is to solve a sequence of increasingly larger subproblems by using previously computed solutions until we've solved the original problem. For all of these problems, there are two parts of a solution: the optimum *value* (e.g., the *length* of a subsequence), or the actual solution itself (e.g., a subsequence of an array). Technically, the algorithms only compute the value of the optimal solution, but as we'll see, recovering the solution itself is often relatively straightforward.

The proofs of correctness for all the algorithms in this chapter simply rely on the correctness of a recurrence. The recurrence tells us the structure of an optimal solution in terms of smaller subproblems. Once the recurrence is established, the only thing that remains is to solve the subproblems (using the recurrence) in an order such that when we solve one subproblem, we have already solved the required smaller subproblems. Thus, for clarity of presentation, we omit the formal "theorem-proof" approach in favor of a pre-algorithm analysis that establishes the recurrence.

3.1 Longest Increasing Subsequence

Problem statement. Let $A[1, \ldots, n]$ denote an array of integers. Our goal is to find the longest increasing subsequence (LIS) in A. For example, the LIS of $[4, 1, 5, 2, 3, 0]$ is $[1, 2, 3]$, and the LIS of $[3, 2, 1]$ is $[3]$ (or $[2]$, or $[1]$).

Analysis. Let $\mathsf{OPT}[i]$ denote the length of the longest increasing subsequence of A that must end at i, so $\mathsf{OPT}[1] = 1$. For $i > 1$, $\mathsf{OPT}[i]$ corresponds to appending $A[i]$ to the longest subsequence ending before i, but the last element of this subsequence must be less than $A[i]$. In other words, we have

$$\mathsf{OPT}[i] = 1 + \max_{\substack{1 \le j < i \\ A[j] < A[i]}} \mathsf{OPT}[j]. \tag{1}$$

(If none of the elements before i are smaller than $A[i]$, then $\mathsf{OPT}[i] = 1$.) The optimal solution OPT must end somewhere, so $\mathsf{OPT} = \max_i \mathsf{OPT}[i]$.

Algorithm. In the algorithm, we maintain an array $d[1, \ldots, n]$, and compute $d[i]$ according to Eq. (1). Computing $d[i]$ requires making $i - 1$ comparisons, so the overall running time is $O(n^2)$. Since we don't know where the optimal solution ends, we return $\max_i d[i]$.

Algorithm 5 Longest Increasing Subsequence

```
1: Initialize an array d[1, ..., n] containing all 1's.
2: for i = 2, ..., n do
3:     for j = 1, ..., i − 1 do
4:         if A[j] < A[i] and d[i] < 1 + d[j] then
5:             Set d[i] = 1 + d[j].          ▷ append A[i] to the LIS ending at j
6: return max_i d[i]
```

To obtain the actual longest subsequence, and not just its length, we can maintain an array p that tracks pointers. In particular, if $d[i]$ is set to $1 + d[j]$, then we set $p[i] = j$, indicating that the LIS ending at i includes j immediately before i. By following these pointers, we can recover the subsequence itself in $O(n)$ time.

3.2 The Knapsack Problem

Problem statement. Suppose there are n items $\{1, 2, \ldots, n\}$, where each item i has integer weight $w_i > 0$ and value $v_i > 0$. We have a knapsack of capacity B. Our goal is to select a subset of items with maximum total value, subject to the constraint that the total weight of selected items is at most B.[1]

Analysis. Let $\mathsf{OPT}[i][j]$ denote the optimal value for the subproblem in which we could only select from items $\{1, \ldots, i\}$ and the knapsack only has capacity j. Notice that $\mathsf{OPT}[i][j]$ either contains item i, or it doesn't (and "chooses" whichever one is more profitable). If it contains item i, then its value is v_i plus the optimal value obtained from items $\{1, \ldots, i-1\}$ with capacity $j - w_i$. Otherwise, its value is the optimal value obtained from items $\{1, \ldots, i-1\}$ with capacity j. Thus, the recurrence is

$$\mathsf{OPT}[i][j] = \max(v_i + \mathsf{OPT}[i][j - w_i], \mathsf{OPT}[i-1][j]) \tag{2}$$

and $\mathsf{OPT} = \mathsf{OPT}[n][B]$ denotes the value of the optimal solution to the original problem.

Algorithm. The algorithm fills out an array $d[1, \ldots, n][1, \ldots, B]$ according to Eq. (2). Notice that each row depends on the previous row, and we return $d[n][B]$ as our solution. (We implicitly assume $d[i][j] = 0$ if $j \leq 0$.) The array has nB entries, and computing each one takes $O(1)$ time, so the total running time is $O(nB)$.

Algorithm 6 Knapsack

```
Initialize an n × B array d containing all 0's.
for j = 1, ..., n do                                   ▷ initialize the first row
    if w_1 ≤ j then set d[1][j] = v_1.
for i = 2, ..., n do
    for j = 1, ..., B do
        Set d[i][j] = max(v_i + d[i − 1][j − w_i], d[i − 1][j]).   ▷ follow (2)
return d[n][B]
```

By tracing back through the array d starting at $d[n][B]$, we can construct the optimal subset of items. At any point, if $d[i][j] = d[i-1][j]$, then we can exclude item i from the solution. Otherwise, we must have $d[i][j] = v_i + d[i-1][j - w_i]$, and this means we include item i in the solution.

[1] Recall that we solved the fractional version of this problem in Sec. 2.2.

Remark. Despite having a running time of $O(nB)$, Algorithm 6 does *not* run in polynomial time. This is because the input length is proportional to $\log B$ rather than B. In other words, if B is doubled then the running time doubles, but the input length only increases by 1 bit.

3.3 Minimum Edit Distance

Problem statement. We are given two strings $A[1, \ldots, m]$ and $B[1, \ldots, n]$. There are three valid operations we can perform on a: insert a character, delete a character, and replace one character for another. Our goal is to convert A into B (or *match* them) using a minimum number of operations (known as the *edit distance*), where substitutions are free if (and only if) the two characters match.

Analysis. Let $\mathsf{OPT}[i][j]$ denote the edit distance between $A[1, \ldots, i]$ and $B[1, \ldots, j]$. Notice that $\mathsf{OPT}[i][j]$ must modify $A[1, \ldots, i]$ such that its last character is $B[j]$, so it must make one of the following three choices:

1. Insert $B[j]$ after $A[1, \ldots, i]$ for a cost of 1; then match $A[1, \ldots, i]$ with $B[1, \ldots, j-1]$.
2. Delete $A[i]$ for a cost of 1; then must match $A[1, \ldots, i-1]$ and $B[1, \ldots, j]$.
3. Replace $A[i]$ with $B[j]$ for a cost of 0 (if $A[i] = B[j]$) or 1 (otherwise); then match $A[1, \ldots, i-1]$ and $B[1, \ldots, j-1]$.

Thus, the subproblems satisfy the following recurrence:

$$\mathsf{OPT}[i, j] = \min \begin{cases} 1 + \mathsf{OPT}[i][j-1] \\ 1 + \mathsf{OPT}[i-1][j] \\ \delta(i, j) + \mathsf{OPT}[i-1][j-1] \end{cases} \tag{3}$$

where $\delta(i, j) = 0$ if $A[i] = B[j]$ and 1 otherwise. For the base cases, we have $\mathsf{OPT}[i][0] = i$ (i.e., delete all i characters of A to obtain the empty string) and $\mathsf{OPT}[0][j] = j$ (i.e., insert all j characters of B to obtain B).

Algorithm. The algorithm fills out an array $d[0, \ldots, m][0, \ldots, n]$ according to (3) and returns $d[m][n]$. Each of the $O(mn)$ entries takes $O(1)$ time to compute, so the total running time is $O(mn)$. To recover the optimal sequence of operations, we can use the same technique of tracing through the table, as we've seen in the previous sections.

3.4 Independent Set on Trees

Problem statement. Let $T = (V, E)$ be a tree on n vertices, where each vertex u has weight $w(u) > 0$. An *independent set* is a subset of vertices such that no two vertices in the subset share an edge. Our goal is to find an independent set S that maximizes $\sum_{u \in S} w(u)$; such as set is known as a *maximum independent set* (MIS).

Analysis. First, we establish some notation. We turn T into a rooted tree by arbitrarily selecting some vertex r as the root. For any vertex u, let $T(u)$ denote the subtree rooted at u (so $T = T(r)$), and let $C(u)$ denote the set of children of u. Also let $\mathsf{OPT}_{\mathsf{in}}[u], \mathsf{OPT}_{\mathsf{out}}[u]$ denote the MIS of $T(u)$ that includes/excludes u, respectively.

If u is a leaf, then $T(u)$ contains u and no edges, so $\mathsf{OPT}_{\mathsf{in}}[u] = w(u)$ and $\mathsf{OPT}_{\mathsf{out}}[u] = 0$. Now suppose u is not a leaf. The MIS corresponding to $\mathsf{OPT}_{\mathsf{in}}[u]$ must include u, so it cannot include any vertices in $C(u)$. This means

$$\mathsf{OPT}_{\mathsf{in}}[u] = w(u) + \sum_{v \in C(u)} \mathsf{OPT}_{\mathsf{out}}[v]. \tag{4}$$

On the other hand, the MIS corresponding to $\mathsf{OPT}_{\mathsf{out}}[u]$ cannot include u, but this means it is free to include/exclude every vertex in $C(u)$, which implies

$$\mathsf{OPT}_{\mathsf{out}}[u] = \sum_{v \in C(u)} \max(\mathsf{OPT}_{\mathsf{in}}[v], \mathsf{OPT}_{\mathsf{out}}[v]). \tag{5}$$

In the end, the maximum independent set of $T = T(r)$ either contains r or it doesn't, so we return $\max(\mathsf{OPT}_{\mathsf{in}}[r], \mathsf{OPT}_{\mathsf{out}}[r])$.

Algorithm. The algorithm maintains two arrays $d_{\mathsf{in}}[1, \ldots, n]$ and $d_{\mathsf{out}}[1, \ldots, n]$, calculated according to Eq. (4) and Eq. (5), respectively. When calculating these two values at a vertex u, we need to ensure that we have already calculated both values for every child of u. One way to do this is to run a DFS (see Sec. 1.2) starting at the root r, and process V in increasing post values. For clarity of presentation, we assume that the vertices are already ordered in this way (or some other that allows us to follow the recurrences).

Algorithm 7 MIS on a Tree

```
Initialize d_in, d_out as two arrays of length n containing all 0's.
for each u ∈ V do
    Let C(u) denote the set of children of u.
    if C(u) is empty then                                   ▷ u is a leaf
        Set d_in[u] = w(u) and d_out[u] = 0.
    else                                                    ▷ u has at least one child
        Set d_in[u] = w(u) + Σ_{v∈C(u)} d_out[v].
        Set d_out[u] = Σ_{v∈C(u)} max(d_in[v], d_out[v]).
return max(d_in[r], d_out[r])
```

In Algorithm 7, the time it takes to process each vertex u is proportional to the number of children of u. Thus, the overall running time is proportional to

$$\sum_{u \in V} |C(u)| = n - 1 = O(n).$$

Note that the sum holds because every vertex v, except for the root, has exactly one parent, so v appears in exactly one set of children. To recover the MIS of T itself, we can maintain

pointers in Algorithm 7. If $d_{\mathsf{in}}[r] > d_{\mathsf{out}}[r]$, then we include r in the solution, exclude all of its children, and recurse on its grandchildren. Otherwise, we exclude r from the solution and recurse on its children.

3.5 Exercises

1. Consider the knapsack problem from Sec. 3.2. Show that the following two greedy algorithms do not always return an optimal solution: (1) sort the items by non-increasing v_i and then iterate through the list, adding every item that fits (2) sort the items by non-increasing v_i/w_i (i.e., "value per weight") and then iterate through the list, adding every item that fits.

2. Suppose, in the knapsack problem, there is an unlimited quantity of every item. Give an $O(nB)$-time algorithm for this problem.

3. Let $A[1, \ldots, n]$ be an array of integers. Give an $O(n)$-time algorithm that finds a contiguous subarray whose sum is maximized.

4. Let $A[1, \ldots, n]$ be an array where $A[i]$ denotes the price of a stock on day i. A *trade* consists of buying the stock on some day i and selling it on some day $j > i$ for a profit of $A[j] - A[i]$. Give an $O(n)$-time algorithm that finds the trade that maximizes profit.

5. Consider the same setting as the previous exercise, but now you are allowed to make at most k trades, where $k < n/2$. Give an $O(nk)$-time algorithm for this problem. (Hint: First design an $O(n^2k)$-time algorithm, then reduce its running time.) (Bonus: what happens when $k \geq n/2$?)

6. Consider the same setup as Sec. 2.1, but now each interval i has weight $w(i) > 0$. Assume that the intervals are sorted such that $t(1) \leq t(2) \leq \cdots \leq t(n)$. You may also assume that you are given an array p where $p[j]$ is that largest i such that $t(i) < s(j)$ ($p[j] = 0$ if no such i exists.) Give an $O(n)$-time algorithm that finds a subset of compatible events whose total weight is maximized.

7. Suppose you are given a function f whose input can be any string and output is an integer. You are also given a string A of length n. Your goal is to partition A into substrings $(s_1, s_2, \ldots, s_k)$ (where $1 \leq k \leq n$) such that, $\sum_{i=1}^{k} f(s_i)$ is maximized. (You get to choose the value of k.) Give an $O(n^2)$-time algorithm for this problem.

8. Consider the same setup as Sec. 3.3, but now we want to find the longest common subsequence (LCS) of A and B. (Recall that a subsequence of a string is not necessarily contiguous.) Give an $O(mn)$-time algorithm for this problem.

9. A string is *palindromic* if it is equal to its reverse (e.g., "abcba"). Give an $O(n^2)$-time algorithm that finds the longest palindromic subsequence of a string of length n. Additionally, give an $O(n^2)$-time algorithm that finds the longest palindromic substring (i.e., contiguous subsequence), also of a string of length n.

10. Suppose we are given a rod of length n, and we can cut it into pieces of integer length. For each $i \in \{1, \ldots, n\}$, we are told the profit $p_i > 0$ we would earn from selling a piece of length i. Give an $O(n^2)$-time algorithm that returns the lengths we should cut to maximize our total profit.

11. Suppose there are an unlimited number of coins in integer denominations $x_1, x_2, \ldots, x_n$. We are given a value v, and our goal is to select a minimum subset of coins such that their total value is v. Give an $O(nv)$-time algorithm for this problem. (Recall from Sec. 2.5 Exercise 6 that the greedy algorithm is not always optimal.)

12. Give an $O(n)$-time algorithm for the MIS on Trees problem (Sec. 3.4) that only uses one DP table (rather than two).

13. A *matching* in a graph is a subset of edges such that no two edges share an endpoint. Let $T = (V, E)$ be a tree where each edge e has weight $w(e) > 0$. Give an $O(n)$-time algorithm that returns a maximum matching in T. (Recall that in Sec. 2.5 Exercise 8, we solved the unweighted version of this problem.)

14. Let $T = (V, E)$ be a tree where each edge e has weight $w(e) \geq 0$, and let $k \geq 2$ be an integer. Give an $O(nk^2)$-time algorithm that finds a connected subgraph of T containing at lesat k edges whose total weight is minimized. (Hint: First solve the problem when T is a binary tree.)

4 Paths in Graphs

In this chapter, we resume our study of graph algorithms from Ch. 1. In particular, we examine multiple solutions to the problem of finding the shortest path between two vertices. If all edges have the same length, then BFS solves this problem in linear time (see Sec. 1.1). However, things get more complicated when the edge lengths differ.

In Sec. 4.1, we solve the problem for directed acyclic graphs. In Sec. 4.2, we solve the problem for general graphs and nonnegative edge lengths. In Sec. 4.3, we remove the assumption that edge lengths are nonnegative. Finally, in Sec. 4.4, we show how we can simultaneously calculate the shortest path between all pairs of vertices.

Throughout this chapter, we let $\delta(s,t)$ denote the length of the shortest path from s to t. Furthermore, the algorithms we study only compute the *length* of a shortest path, rather than the path itself. But as we saw in Ch. 3, we can maintain parent pointers and trace them at the end to recover the shortest paths themselves.

4.1 Paths in a DAG

Problem statement. Let $G = (V, E)$ be a directed acyclic graph (DAG) where each edge $e = (u, v)$ has (possibly negative) length $\ell(u, v) = \ell_e$, and let s be a starting vertex. Our goal is to find the length of the shortest path from s to all other vertices.

Algorithm. First, topologically sort the vertices of G such that no edges enter the first vertex (see Sec. 1.4 Exercise 4).[1] Then, for each vertex v in this order, and each edge $e = (u, v)$ entering v, we "relax" e (more on this later) by setting $d[v] = \min(d[v], d[u] + \ell_e)$.

Algorithm 8 Shortest Paths in a DAG

```
1: Use DFS to topologically sort the vertices V.
2: Initialize d[v] = ∞ for all v ∈ V.
3: for each vertex v in topological order do
4:     if v = s then
5:         Set d[v] = 0.
6:     else
7:         for each edge (u, v) do
8:             Set d[v] = min(d[v], d[u] + ℓ(u, v)).        ▷ "relax" the edge (u, v)
9: return d
```

Theorem 4.1. *At the end of Algorithm 8, $d[v]$ contains the distance from s to v.*

Proof. Notice that Algorithm 8 is a dynamic program! Let $\mathsf{OPT}[v]$ denote the distance from s to v. Then $\mathsf{OPT}[s] = 0$ and for $v \neq s$, the recurrence is

$$\mathsf{OPT}[v] = \min_{(u,v)\in E} d(u) + \ell(u, v).^2$$

[1] Note that we might as well assume this first vertex is s, because no vertices that appear before s can be reached from s.

[2] Notice that this recurrence is very similar to the one for LIS, from Sec. 3.1.

This is because the shortest path from s to v must go to some intermediate vertex u (possibly s if $(s, v) \in E$) and then traverse the edge (u, v). Thus, $\mathsf{OPT}[v]$ is simply the best option across all possible edges entering v. Since the algorithm calculates d according to this recurrence, we've proven the theorem. □

Edge relaxations. To *relax* a (directed) edge (u, v) means setting $d[v] = \min(d[v], d[u] + \ell(u, v))$. This is a useful procedure: if u is the penultimate node on a shortest s-v path and $d[u] = \delta(s, u)$, then relaxing (u, v) results in $d[v] = \delta(s, v)$. We will see this process being used again in both Dijkstra's algorithm and the Bellman-Ford algorithm.

4.2 Dijkstra's algorithm

Problem statement. Let $G = (V, E)$ be a directed graph where each edge $e = (u, v)$ has length $\ell(u, v) = \ell_e \geq 0$, and let s be a starting vertex. Our goal is to find the length of the shortest path from s to all other vertices.

Algorithm. We present Dijkstra's algorithm; notice its similarities to both BFS (Sec. 1.1) and Prim's algorithm (Sec. 1.3). It starts by adding s to a set S, and in each step, adds a vertex in $V \setminus S$ whose distance from s through vertices in S is minimized.

Algorithm 9 Dijkstra's Algorithm

```
Initialize d[s] = 0 and d[v] = ∞ for all v ∈ V \ {s}.
Initialize sets S = ∅ and Q = V.
while |Q| ≥ 1 do                          ▷ Q contains the unprocessed vertices
    Let u = arg min_{x∈Q} d[x].
    Remove u from Q and add u to S.       ▷ S contains the processed vertices
    for each neighbor v of u do
        Set d[v] = min(d[v], d[u] + ℓ(u, v)).      ▷ "relax" the edge (u, v)
return d
```

Remark. As mentioned earlier, whenever we update the value $d[v]$, we can direct a pointer from v to its parent u that is being processed in this iteration of the while loop. By following these pointers back to s, we can recover the shortest path from s to every vertex in V.

Theorem 4.2. *At the end of Dijkstra's algorithm, for every vertex u, $d[u]$ contains the distance from the starting vertex s to u.*

Proof. We will show, by induction on $|S|$, that whenever a vertex u gets added to S, we have $d[u] = \delta(u)$ where $\delta(u)$ denotes the distance from s to u. When $|S| = 1$, then s is the only vertex in S, and we indeed have $d[s] = \delta(s) = 0$.

Now suppose $|S| = k$ and u is the $(k + 1)$-th vertex that we are adding to S. Let P denote the s-u path that the algorithm has found, and let P^* denote any s-u path. Since u

is not in S yet, P^* must leave S through some edge (x, y); let P_x^* denote the s-x subpath in P^*. By the inductive hypothesis, we have

$$d[x] + \ell(x, y) = \delta(x) + \ell(x, y) \leq \ell(P_x^*) + \ell(x, y) \leq \ell(P^*).$$

When x was added to S, we relaxed the edge (x, y), so $d[y] \leq d[x] + \ell(x, y)$. Finally, in this iteration, the algorithm is removing u from Q (rather than y), so $d[u] \leq d[y]$. Putting this all together, we have $d[u] \leq \ell(P^*)$, so P is indeed a shortest path. □

4.3 The Bellman-Ford algorithm

Problem statement. Let $G = (V, E)$ be a directed graph where each edge $e = (u, v)$ has (possibly negative) length $\ell(u, v) = \ell_e$, and let s be a starting vertex. Our goal is to find the length of the shortest path from s to all other vertices.

Remark. We assume that G has no cycle whose total weight is negative, because the existence of such a cycle muddles the meaning of "shortest path." To go from s to t, we could first go to the negative cycle and traverse it any arbitrarily large number of times. Then the resulting "path" has very negative cost, so it's not even clear what to return.[3]

Algorithm. The algorithm is quite simple: run for $n - 1$ iterations, and in each iteration, relax every edge of the graph. The total running time is $O(mn)$.

Algorithm 10 Bellman-Ford

```
Initialize d[s] = 0 and d[v] = ∞ for all v ∈ V \ {s}.
for n − 1 iterations do
    for each edge (u, v) ∈ E do
        Set d[v] = min(d[v], d[u] + ℓ(u, v)).        ▷ "relax" the edge (u, v)
return d
```

Before formally proving the theorem, let's intuitively analyze the algorithm. Consider a shortest path $P = (s, u_1, u_2, \ldots, u_k)$ from s to $t = u_k$. In the i-th iteration, the edge (u_{i-1}, u_i) is relaxed and $d[u_i]$ is set to $\delta(s, u_i)$. Thus, after $k \leq n - 1$ iterations, every vertex u_i on P satisfies $d(u_i) = \delta(s, u_i)$. In particular, $d(t) = \delta(s, t)$.

Theorem 4.3. *At the end of the Bellman-Ford algorithm, for every vertex v, $d[v]$ contains the distance from the starting vertex s to v.*

Proof. Let $\lambda(i, v)$ denote the length of the shortest *walk* (i.e., sequence of adjacent vertices with repetition allowed) from s to v that contains at most i edges. Since the shortest walk from s to v contains at most $n - 1$ edges, it is sufficient to prove that after i iterations, $d[v] \leq \lambda(i, v)$. If $i = 0$, then $d[s] = 0 \leq 0 = \lambda(i, s)$ and $\lambda(i, v) = \infty$ for all $v \in V \setminus \{s\}$, so

[3] Actually, vertices are not allowed to repeat in a path, so the shortest path is technically still well-defined. What we really mean is that there is no shortest *walk*, but overloading the term "path" this way is very common. Finding the actual shortest *path* is very difficult when the graph has negatively weighted edges.

$d[v] \leq \lambda(0, v)$. Now consider $i \geq 1$, a vertex v, and a shortest walk W from s to v containing at most i edges. If the last edge of W is (u, v), then

$$\lambda(i-1, u) + \ell(u, v) = \lambda(i, v).$$

By the inductive hypothesis, we have $d[u] \leq \lambda(i-1, u)$. So in the i-th iteration, when relaxing (u, v), we set $d[v] \leq \lambda(i-1, u) + \ell(u, v) = \lambda(i, v)$, as desired. □

4.4 The Floyd-Warshall algorithm

Problem statement. Let $G = (V, E)$ be a directed graph where each edge $e = (u, v)$ has (possibly negative) length $\ell(u, v) = \ell_e$. Our goal is to find the length of the shortest path between every pair of vertices. (As discussed in Sec. 4.3, we assume that the graph has no negative cycles.)

Remark. One solution for this problem is to run the Bellman-Ford algorithm (or Dijkstra's, if all lengths are nonnegative) from every vertex. However, this approach can be very slow; the following algorithm is both simple and faster when the graph has many edges.

Algorithm. Once again, we'll use dynamic programming. Suppose the vertices are numbered $1, 2, \ldots, n$. Let $\delta(u, v, r)$ denote the length of the shortest u-v path that only passes through vertices $\{1, 2, \ldots, r\}$, so $\delta(u, v) = \delta(u, v, n)$. Notice that $\delta(u, v, 0)$ is $\ell(u, v)$ if (u, v) is an edge, or ∞ otherwise. For $r \geq 1$, the path corresponding to $\delta(u, v, r)$ either passes through vertex r or it doesn't. Thus, we have the following recurrence:

$$\delta(u, v, r) = \min(\delta(u, v, r-1), \delta(u, r, r-1) + \delta(r, v, r-1)).$$

The Floyd-Warshall algorithm implements this recurrence in $O(n^3)$ time, and as usual, we can maintain pointers and follow them at the end recover the actual shortest paths.

Algorithm 11 Floyd-Warshall

```
1: Number the vertices so that V = {1, ..., n}.
2: for u = 1, ..., n do
3:     for v = 1, ..., n do
4:         Set d[u, v, 0] = ℓ(u, v) if (u, v) ∈ E and ∞ otherwise.
5: for r = 1, ..., n do
6:     for u = 1, ..., n do
7:         for v = 1, ..., n do
8:             Set d[u, v, r] = min(d[u, v, r − 1], d[u, r, r − 1] + d[r, v, r − 1]).
9: return d
```

4.5 Exercises

1. Let $G = (V, E)$ be a directed graph where each edge $e = (u, v)$ has an *integer* length $\ell(u, v) \geq 1$, and let s be a starting vertex. Show how BFS can be used to find the shortest path from s to all other vertices.

2. Consider the same setup as Sec. 4.1, but now our goal is to find the *longest* path from s to all other vertices. Give a linear-time algorithm for this problem.

3. Notice that Algorithm 8 looks at edges entering each vertex, while Algorithms 9 and 10 look at edges leaving each vertex. Redesign Algorithm 8 so that it also looks at edges leaving each vertex, and prove that the resulting algorithm is still correct.

4. Show that Dijkstra's algorithm can be implemented to run in $O(n^2)$ time (where n is the number of vertices in the graph, as usual).

5. Consider the following alternative to Bellman-Ford: add a large constant to every edge lengths so that all edge lengths are nonnegative, and then run Dijkstra's algorithm. Show that this algorithm does not necessarily return a correct solution.

6. Let G be a directed graph with nonnegative edge lengths, and let x be a vertex of G. An *x-walk* is a walk that passes through x. Give an $O(n^2)$-time algorithm that finds the shortest x-walk between every pair of vertices.

7. Let G be a directed graph with nonnegative edge lengths, and let s be a vertex of G. Give an algorithm that finds the shortest cycle containing s. The running time should be asymptotically the same as running Dijkstra's algorithm once.

8. Let G be a directed graph with nonnegative edge lengths, and let s, t be vertices of G. Give an algorithm that finds the shortest walk containing an even number of edges between s and t (or reports that no such walk exists). The running time should be asymptotically the same as running Dijkstra's algorithm once.

9. Give an $O(mn)$-time dynamic programming version of the Bellman-Ford algorithm (Algorithm 10), and prove that it is correct.

10. Consider the Bellman-Ford algorithm, but we run it for one extra iteration at the end. Prove that graph has a negative cycle reachable from s if and only if some d-value gets updated in the n-th iteration. Use this to design an $O(mn)$-time algorithm that detects whether or not a graph has a negative cycle.

11. Suppose you have already ran the Floyd-Warshall algorithm to obtain a matrix d where $d[u, v] = \delta(u, v)$ for every $u, v \in V$. Now suppose the length of an edge e decreases from w_e to w'_e. Give an $O(n^2)$-time algorithm to update the matrix d for this new graph. (Bonus: what if all lengths are positive, and the graph is undirected?)

5 Maximum Flows

In this chapter, we study network flows and cuts. A *flow network* is a directed graph $G = (V, E)$ (on n vertices and m edges, as usual), where each edge e has *capacity* $c_e \geq 1$, along with a designated *source* $s \in V$ and *sink* $t \in V$. For simplicity, we assume that every capacity is an integer, no edges enter s, and no edges leave t.

Any subset of vertices S is known as a *cut*, and the set of edges leaving (or *crossing*) the cut is denoted by $\delta^{\mathsf{out}}(S) = \{(u, v) \in E : u \in S, v \notin S\}$. Similarly, let $\delta^{\mathsf{in}}(S)$ denote the set of edges entering S. In this section, we will focus on a subset of cuts known as *s-t cuts*; these are cuts S that contain the source vertex s.

A *flow* is a function $f : E \to \mathbb{R}^+$; let $f^{\mathsf{out}}(S)$ denote the sum of flow on edges in $\delta^{\mathsf{out}}(S)$, and define $f^{\mathsf{in}}(S)$ analogously. When working with a flow, we often implicitly assume that it is *feasible*, that is, it satisfies the following conditions:

1. Capacity constraints: For each edge $e \in E$, we have $0 \leq f(e) \leq c_e$.
2. Conservation: For each vertex $u \in V \setminus \{s, t\}$, we have $f^{\mathsf{in}}(u) = f^{\mathsf{out}}(u)$.

The *value* of a flow f is defined as $|f| = f^{\mathsf{out}}(s)$, and the maximum flow problem is to find a feasible flow with maximum value. The minimum s-t cut problem is to find an s-t cut $S \subset V$ such that the total capacity $c(S)$ of edges in $\delta^{\mathsf{out}}(S)$ is minimized. In Sec. 5.1, we show how the Ford-Fulkerson algorithm solves both problems, and in the subsequent sections, we illustrate applications of these problems.

But before we begin, we state an important connection between flows and cuts. We leave the proof of the following lemma as exercises.

Lemma 5.1. *For any flow f and s-t cut S, we have $|f| = f^{\mathsf{out}}(S) - f^{\mathsf{in}}(S) \leq c(S)$.*

The lemma formalizes two intuitive ideas about flows and s-t cuts: flow value is conserved across the cut, and this value cannot exceed the capacity of the cut.

5.1 The Ford-Fulkerson algorithm

Problem statement. Let G be a flow network with source s and sink t; our goal is to find a maximum flow from s to t. As we will see, we can also consider the problem of finding an s-t cut of minimum (outgoing) capacity.

Remark. A natural approach is to use BFS (see Sec. 1.1) to find a path from s to t, increase $f(e)$ for every edge e on this path, remove edges that have zero remaining capacity, and repeat until the graph contains no s-t path. This algorithm has the right idea but won't always work; we leave this an exercise.

Algorithm. Start with $f(e) = 0$ for every edge e. For any flow f, we define the *residual network* G^f as follows: G^f and G have the same set of vertices. For each edge $e = (u, v)$ in G, if $f(e) < c_e$, we add e to G^f and set its residual capacity to $c^f_e = c_e - f(e)$; these are *forward* edges. Furthermore, if $f(e) > 0$, we add another edge (v, u) to G^f with capacity $c^f_e = f(e)$; these are *backward* edges. (So G^f has at most twice as many edges as G.)

The algorithm repeatedly finds an s-t path P in G^f (using, e.g., BFS), and uses P to increase the current flow. Such a path is known as an *augmenting path*. The amount by which we modify the current flow value is the minimum residual capacity in P, which is always positive.

Algorithm 12 Ford-Fulkerson

```
1: Initialize f(e) = 0 for all e ∈ E.
2: while there exists an augmenting path P in G^f do
3:     Let Δ_P = min_{e∈P} c^f_e.                ▷ Δ is the "bottleneck" capacity in P
4:     for each edge e ∈ P do
5:         if e is a forward edge then
6:             Increase f(e) in G by Δ_P.
7:         else
8:             Decrease f(e') in G by Δ_P, where e' = (v, u).
9:     Update the residual graph G^f.
10: return f
```

Theorem 5.2. *Algorithm 12 computes a maximum flow in $O(mv)$ time, where m is the number of edges in G and v is the value of the maximum flow.*

Proof. We start by proving that the algorithm terminates in $O(mv)$ time. We do this by proving that $|f|$ increases by $\Delta_P \geq 1$ in every iteration; this suffices because $|f|$ must always be at most v, and each iteration takes $O(m)$ time.[1] Observe that for any augmenting path P, the first edge e of P must leave s, and no edges of G enter s, so e must be a forward edge. This means $f(e)$ increases by Δ_P, so $|f|$ increases by $\Delta_P \geq 1$.

Now we prove that the final flow f is a maximum flow.[2] Let S^* denote the set of vertices reachable from s at the end of the Ford-Fulkerson algorithm; note that $t \notin S^*$ because G^f has no augmenting path. Also, for every edge $e = (u, v) \in \delta^{\mathsf{out}}(S^*)$, we have $f(e) = c_e$. For if not, then e would be a forward edge in G^f, so v would be reachable from s, contradicting the fact that $v \notin S^*$. We can similarly show that for every edge $e' \in \delta^{\mathsf{in}}(S^*)$, we have $f(e') = 0$. Combining this with Lemma 5.1, we get

$$|f| = f^{\mathsf{out}}(S^*) - f^{\mathsf{in}}(S^*) = \sum_{e \in \delta^{\mathsf{out}}(S^*)} c_e - 0 = c(S^*).$$

Recall that Lemma 5.1 also states that the value of any flow is at most the capacity of any cut. Thus, f must have maximum value, because any larger flow f' would have $|f'| > |f| = c(S^*)$, contradicting the lemma. □

[1] From the definition of G^f, we must have $\Delta_P > 0$. Also, since all residual capacities are initially the original (integer) capacities, the first Δ_P is an integer. So after the first iteration, all residual capacities are still integers. Inductively, this implies Δ_P is always an integer; hence $\Delta_P \geq 1$.

[2] Technically, we should first show that f is a feasible flow throughout the algorithm. The proof is fairly straightforward but somewhat tedious, so we leave it as an exercise.

As a bonus, we can also solve the minimum s-t cut problem in $O(mv)$ time. Note that this doesn't necessarily return the minimum cut of the graph, because that cut might not include s. We leave this problem as an exercise.

Theorem 5.3. *We can compute the minimum cut of a flow network G with maximum flow value v in $O(mv)$ time.*

Proof. We first compute a maximum flow f in $O(mv)$ time using the Ford-Fulkerson algorithm. After that, we are left with the residual network G^f that contains no s-t paths. In $O(m)$ time, we find the cut S^* containing vertices reachable from s in G^f. This cut must be a minimum cut, because if there were a cut S' with smaller capacity, then we'd have $c(S') < c(S^*) = |f|$, which violates Lemma 5.1. □

Finally, we can use the proof of Theorem 5.2 to prove the *Max-Flow Min-Cut Theorem* stated below; we leave the details as an exercise.

Theorem 5.4. *In any flow network, the value of a maximum s-t flow is equal to the capacity of a minimum s-t cut.*

5.2 Bipartite Matching

Problem statement. Let $G = (V, E)$ be a bipartite graph where $V = L \cup R$. Recall that a *matching* is a subset of edges such that no two edges share an endpoint. Our goal is to find a matching in G containing the largest number of edges.

Algorithm. Construct a flow network $G' = (V', E')$ as follows: add vertices s, t to V to form V'.[3] The edge set E' contains every edge in G directed from L to R, as well as (s, u) for every $u \in L$ and (v, t) for every $v \in R$. All edges in G' have capacity 1. Use the Ford-Fulkerson algorithm to find a maximum s-t flow f in G', and returns the edges (u, v) (where $u, v \in V$) that satisfy $f(u, v) = 1$.

Theorem 5.5. *The algorithm above returns a maximum matching in G in $O(n^2)$ time.*

Proof. Let M denote the edges returned by the algorithm; we first show that M is a matching. Consider a vertex $u \in L$. There exists at most one $v \in R$ such that $f(u, v) = 1$ because the amount of flow entering u is at most 1, so the amount of flow leaving u is also at most 1. We can similarly show that every vertex in R is incident to at most one edge in M.

Next, we show $|f| = |M|$ by applying Lemma 5.1 to the cut $\{s\} \cup L$. Each edge of M contributes exactly 1 unit of flow leaving this cut, and no edges enter this cut. Thus, $|f|$ is just the amount of flow leaving the cut, which is equal to $|M|$.

Finally, we show that M is a maximum matching. For any matching M', we can construct a flow f' as follows: for every $(u, v) \in M'$, we set $f'(s, u) = f'(u, v) = f'(v, t) = 1$ (and 0 everywhere else). Since f' consists of $|M'|$ edge-disjoint paths, it has value $|M'|$. This value is at most $|f| = |M|$ because f is a maximum flow, so $|M'| \leq |M|$.

[3] We can visualize G' as follows: L and R are two columns of vertices, where L is left of R. Vertex s is left of L, and t is right of R. All edges go from left to right, and no edge "skips" over any column.

The running time of the algorithm is bounded by the running time of computing a maximum flow. If $|L| = n/2$, then the maximum flow value is at most $n/2$, so running the Ford-Fulkerson algorithm would take $O(n^2)$ time. □

5.3 Vertex Cover in Bipartite Graphs

Problem statement. Let $G = (V, E)$ be a bipartite graph where $V = L \cup R$. Recall that $S \subseteq V$ is a *vertex cover* if every edge of G is incident to at least one vertex in S. Our goal is to find a vertex cover of G containing the fewest number of vertices.

Algorithm. We construct a flow network $G' = (V', E')$ very similar to the one from Sec. 5.2. The vertex set V' contains V and two additional vertices s, t. The edge set E' contains every edge in G directed from L to R, (s, u) for every $u \in L$, and (v, t) for every $v \in R$. The original edges have capacity ∞ and the new edges have capacity 1. Use the Ford-Fulkerson algorithm to find a minimum s-t cut S in G', and return $(L \setminus S) \cup (R \cap S)$.

Theorem 5.6. *The algorithm above returns a minimum vertex cover of G in $O(n^2)$ time.*

Proof. Let C denote the output of the algorithm; we first show that C is a vertex cover. Any edge $(u, v) \in E$ not covered by C must satisfy $u \in L \cap S$ and $v \in R \setminus S$. Thus, (u, v) crosses the cut, so $c(S) = \infty$, which contradicts the fact that S is a minimum s-t cut.

Now we show $|C| = c(S)$. There are two types of edges crossing S: those of the form (s, u) where $u \in L \setminus S$, and those of the form (v, t) where $v \in R \cap S$. Each of these edges contributes capacity 1 to $c(S)$, and the number of these edges is $|L \setminus S| + |R \cap S| = |C|$.

Finally, we show that C is a minimum vertex cover. For any vertex cover C', consider the s-t cut $S' = \{s\} \cup (L \setminus C') \cup (R \cap C')$. By the same reasoning as above, we have $c(S') = |C'|$. Since S is a minimum s-t cut, we have $|C| = c(S) \leq c(S') = |C'|$, as desired.

The running time is again (see Theorem 5.5) bounded by the running time of computing a maximum flow. Since the maximum flow value in G' is at most $|L|$, the running time of Ford-Fulkerson is $O(n^2)$. □

5.4 Exercises

1. Let f be a feasible flow from s to t. Prove that $|f| = f^{\mathsf{in}}(t)$.

2. Prove that f remains a feasible flow during any run of the Ford-Fulkerson algorithm.

3. Let S be a cut in a flow network G, and let f be an s-t flow in G. Prove that the value of f is equal to $f^{\mathsf{out}}(S) - f^{\mathsf{in}}(S)$.[4]

4. Use the previous exercise to prove $|f| \leq c(S)$ for every flow f and cut S.

5. Show why the "natural approach" described at the beginning of Sec. 5.1 doesn't always return a maximum flow.

[4]This statement is intuitively true, but you should still write a proof. In particular, at least one $\sum_{u \in S}$ should probably appear somewhere in the proof.

6. Give an $O(mnv)$-time algorithm that finds the *global* minimum cut (i.e. not necessarily an s-t cut) of a flow network.

7. Prove the Max-Flow Min-Cut Theorem (Theorem 5.4).

8. Consider the maximum flow problem, but suppose there are $k \geq 1$ sources $\{s_1, \ldots, s_k\}$ and $\ell \geq 1$ sinks $\{t_1, \ldots, t_\ell\}$. No edges enter any source or leave any sink. A feasible flow is one that satisfies capacity constraints, as well as conservation at all vertices other than sources and sinks. The value of a flow is now defined as the total amount of flow leaving the sources, and we want to find a flow of maximum value. Show how this problem can be solved using the Ford-Fulkerson algorithm.

9. Consider the minimum s-t cut problem, but now each *vertex* (instead of edge) has a capacity; our goal is to remove a minimum weight subset of vertices such that in the resulting graph, there is no path from s to t. Show how this problem can be reduced to the standard minimum s-t cut problem.

10. Consider the same problem as Sec. 5.3, but now each vertex u has weight $w(u) \geq 0$, and we want to find the minimum weight vertex cover. (Sec. 5.3 was the special case where all weights are 1.) Show how to modify the algorithm in that section to solve this problem.

11. Prove that in any bipartite graph, the size of the maximum matching is equal to the size of the minimum vertex cover. (This is known as the König-Egeváry Theorem.)

12. Let $G = (V, E)$ be a bipartite graph where $V = L \cup R$ and $|L| \leq |R|$. For any subset $S \subseteq V$, let $N(S)$ denote the set of neighbors of S. Prove the following: G has a matching of size $|L|$ if and only if $|S| \leq |N(S)|$ for every $S \subseteq L$. (This is known as Hall's Theorem.)

13. Let G be a flow network in which all edges have capacity 1. Prove that there are k edge-disjoint s-t paths if and only if the maximum s-t flow has value at least k.

14. Consider the same setting as the previous exercise. Prove that the maximum number of edge-disjoint s-t paths is equal to the minimum number of edges we'd need to remove to separate s from t. (This is known as Menger's theorem.)

6 Linear Programming

In Chapters 2 (Greedy Algorithms) and 3 (Dynamic Programming), we saw two general approaches to solving optimization problems that are applicable to a variety of settings (e.g., subsequences, knapsacks, graphs). In this chapter, we study another general technique known as *linear programming*. There are entire courses devoted to linear programming; it is a deep topic with countless applications.

This chapter is a bit different from previous chapters. In Sec. 6.1, we introduce the basic idea behind linear programming and give two examples of applications. But we won't actually study any algorithms that solve linear programs; for our purposes, we can assume that we have access to an LP solver. We illustrate this idea by giving an algorithm for the vertex cover problem in Sec. 6.2. Finally, in Sec. 6.3, we show how linear programming allows us to optimally play two-player zero-sum games.

6.1 The Basics of LPs

A *linear program* (LP) is an optimization problem in which the objective is to maximize (or minimize) a linear function subject to constraints in the form of linear inequalities. For example, here is a linear program that has 2 variables and 1 constraint:[1]

$$\begin{aligned} \max\ & 3x_1 + 2x_2 \\ & x_1 + x_2 \leq 5 \\ & x_1, x_2 \geq 0 \end{aligned}$$

A *solution* to the LP is an assignment of each variable to a real number. It is *feasible* if it satisfies all of the constraints, and it is *optimal* if it is feasible and has the highest (or lowest, for minimization problems) objective value among all feasible solutions. (An LP can have multiple optimal solutions.) In the LP above, the solution $(x_1, x_2) = (1, 2)$ is feasible, $(5, 3)$ and $(-1, 4)$ are not, and $(5, 0)$ is optimal.

We often use vectors and matrices to specify linear programs. If an LP has n variables and m constraints, then we let $x \in \mathbb{R}^n$ denote the vector of variables and $c \in \mathbb{R}^n$ denote the coefficients of the objective function. Furthermore, we let $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$ specify the constraints. Putting it all together, an LP in *canonical form*[2] looks like this:

$$\begin{aligned} \max\ & c^\top x \\ & Ax \leq b \\ & x \geq 0 \end{aligned}$$

Note that "$x \geq 0$" means each of the n entries of x is at least 0. Similarly, "$Ax \leq b$" means each of the m entries of $Ax \in \mathbb{R}^m$ is at most the corresponding entry in b. So in the first LP, we had $x = (x_1, x_2), c = (3, 2), A = (1, 1)$, and $b = (5)$ (where x, c, b are column vectors). We now look at two well-known applications of linear programming.

[1] The constraint "$x_1, x_2 \geq 0$" is short for "$x_1 \geq 0$ and $x_2 \geq 0$", and these are not usually counted as constraints because they are so ubiquitous in LPs. However, it is not necessary for them to appear in an LP.

[2] Different sources refer to different things as canonical (or sometimes *standard*) form, but all of the forms you might find can be transformed into each other.

The Diet Problem. Suppose a store sells n food items, where each item j has cost c_j per unit. Also, there are m nutrients, and item j provides A_{ij} units of nutrient i. Our diet mandates that we eat at least b_j units of nutrient j. Our goal is to spend the least amount of money on the n food items while satisfying the m nutritional requirements.

We can model this problem as an LP as follows. Let x be an n-dimensional vector; we need to determine x_j for all $j \in \{1, \ldots, n\}$, which represents the amount of food j we buy. Let $c \in \mathbb{R}^n$ denote the vector containing the costs, let A be an $m \times n$ matrix whose (i, j)-th entry is A_{ij}, and let $b \in \mathbb{R}^m$ denote the vector containing the nutritional lower bounds. Our goal is to minimize $c^\top x$ such that $Ax \geq b$ and $x \geq 0$.

The Manufacturing Problem. Now suppose we're the ones doing the selling. Our factory is capable of producing n products, which are made from m different materials. Each unit of product j requires A_{ij} units of material i and yields c_j units of profit. We have b_i units of material i at our disposal, and our goal is to maximize the total profit.

In this case, x_j represents the amount of product j we produce. The n-dimensional vector c contains the profits, and $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$ encode the material constraints. Our goal is to maximize $c^\top x$ such that $Ax \leq b$ and $x \geq 0$.

Remark. The manufacturing problem LP is in canonical form, but the LP for the diet problem is not. However, if we negate the vector c then the objective becomes a maximization, and if we negative all entries of A and b then the inequality signs flip. In general, the objective can be maximize or minimize, the constraints can be inequalities or equalities, and variables are allowed to be negative.

But no matter what the LP looks like, it can be converted to an equivalent LP in canonical form using variable substitution, negating coefficients, and other small tricks; we leave the details as an exercise. For our purposes, let's assume that we can efficiently find an optimal solution of any linear program, even if it's not in canonical form.[3]

6.2 Vertex Cover

Problem statement. Let $G = (V, E)$ be an undirected graph. A *vertex cover* is a subset of vertices S such that every edge is incident to at least one vertex in S. Our goal is to find a vertex cover of G containing the fewest number of vertices.

Algorithm. Associate each vertex u with a variable $x_u \in \{0, 1\}$. (Intuitively, $x_u = 1$ corresponds to selecting u to be included in the solution, but this is not part of the algorithm.) Then the vertex cover problem is equivalent to the following *integer* LP:

$$\begin{aligned} \min \sum_{u \in V} x_u \\ x_u + x_v \geq 1 \quad \forall \{u, v\} \in E \\ x_u \in \{0, 1\} \quad \forall u \in V. \end{aligned}$$

[3]Again, there are entire courses and books devoted to linear programming and its algorithms; if you are interested in learning about them, I recommend looking at other resources online.

This problem is almost an LP; the only problem is the $x_u \in \{0,1\}$ constraint. Let's replace this constraint with $x_u \geq 0$ and solve the resulting LP to obtain an optimal solution $x^* \in \mathbb{R}^n$. We return $S = \{u : x_u^* \geq 1/2\}$; let $\mathsf{OPT} \subseteq V$ denote an optimal solution.

Theorem 6.1. *The set S is a vertex cover and satisfies $|S| \leq 2 \cdot |\mathsf{OPT}|$.*

Before proving the theorem, we first prove that the optimal value of the linear program is at most the number of vertices in an optimal vertex cover.

Lemma 6.2. *The LP solution x^* and OPT satisfy $\sum_{u \in V} x_u^* \leq |\mathsf{OPT}|$.*

Proof. The integer linear program above (i.e., with the $x_u \in \{0,1\}$ constraints) is equivalent to the vertex cover problem, so its optimal value is equal to $|\mathsf{OPT}|$. By relaxing the constraints $x_u \in \{0,1\}$ to $x_u \geq 0$, the optimal value of the program can only improve, i.e., decrease. Thus, the optimal value of the LP is at most $|\mathsf{OPT}|$, as desired. □

Proof of Theorem 6.1. To show that S is a vertex cover, we must show that every edge $e = \{u, v\}$ is covered. Since x^* is a feasible solution to the linear program, we must have $x_u^* + x_v^* \geq 1$. Thus, at least one the values in $\{x_u^*, x_v^*\}$ must be at least $1/2$, so that corresponding vertex gets included in S, which means it covers the edge e.

Now we bound the size of S. Since every $u \in S$ satisfies $x_u^* \geq 1/2$, we have

$$|S| = \sum_{u \in S} 1 \leq \sum_{u \in S} 2x_u^* \leq 2 \cdot \sum_{u \in V} x_u^* \leq 2 \cdot |\mathsf{OPT}|,$$

where the final inequality follows from Lemma 6.2. □

6.3 Zero-sum Games

Problem statement. Alice and Bob are playing a game; Alice has m options denoted by $[m] = \{1, \ldots, m\}$ while Bob has n options denoted by $[n] = \{1, \ldots, n\}$. If Alice chooses option i and Bob chooses option j, then Alice receives A_{ij} dollars from Bob (or she gives him this amount, if A_{ij} is negative). Of course, Alice wants to maximize her payoff, and Bob wants to minimize it. Our goal is to determine the maximum payoff that Alice can *guarantee* for herself, and her corresponding strategy.[4]

Algorithm. A *pure* strategy is defined by always selecting one option, while a *mixed* strategy is one that specifies a probability distribution over the options. If Alice chooses a pure strategy of option i, then the payoff she can guarantee is $\min_j A_{ij}$, since Bob can pick the strategy j that achieves this minimum value.

Now let's consider Alice's guaranteed payoff if she chooses a mixed strategy, that is, suppose she chooses option i with probability x_i. The key is that Alice can solve for her

[4]This game is known as zero-sum because the sum of Alice's payoff and Bob's payoff is always zero. Alice is happy when her payoff is positive, while Bob is happy when her payoff is negative. There is no result in which both of them are happy, or both of them are sad.

guaranteed payoff value by modeling this problem as a linear program:

$$\max u$$
$$\sum_{i=1}^{n} x_i \cdot A_{ij} \geq u \quad \forall j \in [n]$$
$$\sum_i x_i = 1$$
$$x_i \geq 0 \quad \forall i \in [m]$$

Constraint j says that, if Bob chooses option j, then in expectation (over Alice's options), Alice's payoff must be at least u. Even if Bob uses a mixed strategy, by combining the n inequalities according to his strategy, we can see that Alice still guarantees herself a payoff of u. Although u is technically a variable in the above LP, its value is determined once all of the x_i are determined, and it represents the maximum payoff that Alice can guarantee. The optimal solution of the above LP is known as the *value* of the game.

Remark. Let u^* denote the value of the game, and suppose Alice is forced to announce her strategy before playing. By solving the LP, she can guarantee herself a payoff of at least u^*. Can she do better if she hides her strategy? Surprisingly, the answer is no. Due to a concept known as *duality*, there is a mixed strategy for Bob that guarantees Alice's payoff is at most u^*.[5] Thus, if Bob plays this strategy, then even if Alice hides her strategy, her payoff will still be at most u^*.

6.4 Exercises

1. Describe an LP that has infinitely many feasible solutions, but no optimal solution. Then describe an LP that has infinitely many optimal solutions.

2. In Sec. 6.1, we claimed that any linear program can be converted into canonical form. In this exercise, we consider how this can be done.

   (a) Suppose that we want to minimize $c^\top x$ (rather than maximize).

   (b) Suppose we have a constraint of the form $\alpha^\top x \geq \beta$ or $\alpha^\top x = \beta$, where $\alpha, x \in \mathbb{R}^n$ and $\beta \in \mathbb{R}$.

   (c) Suppose we have a variable x_i that is unrestricted in sign, that is, x_i is allowed to be positive or negative (or 0).

   For each situation above, explain how we can modify the LP such that an optimal solution of the new LP yields an optimal solution to the original LP.

[5] A future version of these notes might contain more about the concept of duality. For now, we'll state that the *dual* of an LP in canonical form is minimizing $b^\top y$ subject to $A^\top y \geq c$ and $y \geq 0$, where $y \in \mathbb{R}^m$ is a new vector of variables. The strong duality theorem states that optimal value of an LP is equal to the optimal value of the dual LP.

3. Explain why it might be impossible to optimally solve a linear program if one of its constraints is a strict inequality (e.g., $x_1 + x_2 < 5$).

4. Let $x^*, y^* \in \mathbb{R}^n$ be two distinct optimal solutions of an LP in canonical form. Prove that $(x^* + y^*)/2 \in \mathbb{R}^n$ is also an optimal solution of the LP. Then show that, in fact, the LP has infinitely many optimal solutions.

5. Recall from Sec. 6.2 that the vertex cover problem can be modeled by an LP. However, the LP didn't specify a matrix A, or vectors b, c, x, as shown in Sec. 6.1. For this problem, determine what A, b, c, and x should be.

6. Again, consider the vertex cover LP from Sec. 6.2. Give an example of a graph such that the optimal LP value is less than the size of the optimal vertex cover.

7. In the weighted vertex cover problem, every vertex u has weight $w_u \geq 0$ and our goal is to find a vertex cover of minimum weight. Show how to modify the algorithm from Sec. 6.2 to obtain a solution whose total weight is at most twice the optimal weight.

8. Consider the maximum flow problem from Ch. 5. Model this problem as an LP, including the exact definitions of the matrix A and vectors b, c, x.

9. Consider the minimum spanning tree problem from Sec. 1.3; we shall model this problem using an *integer* LP (ILP, see Sec. 6.2) in two different ways. Both formulations contain a variable $x_e \in \{0, 1\}$ for every edge $e \in E$. The objective of both is to minimize $\sum_{e \in E} w_e x_e$, but they have different constraints.

   (a) For every $S \subset V$, let $\delta(S)$ denote the set of edges with exactly one endpoint in S. Any tree must contain at least one edge in $\delta(S)$ for every S. Express this idea as a set of constraints in the ILP.

   (b) Any tree contains at most $|S| - 1$ edges within any $S \subseteq V$, and any spanning tree contains $n - 1$ total edges. Express these ideas as constraints in the ILP.

   Suppose we have an optimal solution to the first ILP. Show how we can convert this to into a spanning tree, and prove that the result is in fact an MST. Repeat this for the second ILP. Finally, for only the first ILP, suppose we replace $x_e \in \{0, 1\}$ with $0 \leq x \leq 1$. Show that the optimal value of the resulting LP can be less than the weight of an MST.

10. Suppose Alice and Bob are playing a game. Alice has 2 options, and Bob has 3 options (so there are 6 possible outcomes). If Alice plays option 1, then the 3 possible payoffs are $(2, 1, -3)$. If she plays option 2, then the 3 possible payoffs are $(-1, -2, 1)$. What is the value of this game, and what are the optimal strategies for Alice and Bob?[6]

[6] You may want to use an LP solver; many can be found online.

7 NP-Hard Problems

So far, we have been presented a problem (e.g., minimum spanning tree, maximum flow), and our goal was to solve it by designing and analyzing an algorithm (e.g., Kruskal's, Ford-Fulkerson). In this section, we will do something quite different. Firstly, instead of considering the optimization problem directly, we will consider its decision version. In the decision version, there is (usually) an additional parameter k. Instead of minimizing the cost of a solution, our goal is to simply determine if a solution with value at most k exists. (If our original goal was to find a solution of maximum value, then the question is whether or not a solution with value at least k exists.)

For example, the decision version of the shortest s-t path problem is "Does there exist a path from s to t in G of length at most k?" For the knapsack problem, it's "Does there exist a subset of items with cost at most W and value at least k?" For maximum flow, it's "Does there exist a feasible flow whose value is at least k?" and so on.

Secondly, instead of giving an algorithm to solve a decision problem, we will actually show that *no* polynomial-time algorithm can solve the problem.[1] In other words, instead of showing that a problem can be optimally solved in, say, $O(n^2)$ time, in this chapter, we show that many problems cannot be solved even in $O(n^{100})$ time.

7.1 Basic Complexity Theory

A decision problem is in the complexity class **P** if it can be decided in polynomial time. The problem is in **NP** if a solution can be verified in polynomial time. In other words, if the answer is "yes," then there should exist a "proof" that convinces us of this in polynomial time. We usually think of the "proof" as a "proposed solution."[2]

For example, consider the shortest s-t path problem again. The input is a graph G, vertices s and t, and a number k. This problem is in P because we can find a shortest s-t path using BFS; let's show it's also in NP. A proposed solution is a path from s to t. To verify this solution, we simply need to check that it meets the definition of an s-t path, and its total length is at most k. This verification process takes linear time, which is polynomial, so the shortest s-t path problem is in NP.

In fact, any problem in P is in NP. Roughly speaking, this is because if a problem is in P, then we don't need to verify a proof in order to be convinced that the answer is "yes" — in polynomial time, we can simply solve the problem ourselves from scratch. One intuitive way of interpreting this is that solving a problem is harder than verifying the solution to a problem (for example, think about Sudoku puzzles). So if a problem can be solved in polynomial time, then its solution can be verified in polynomial time.

[1] This statement is true only if we assume P $\neq$ NP, which most computer scientists believe, but remains as perhaps the biggest open question in computer science. For a clear and thorough treatment of this topic, I recommend *Introduction to the Theory of Computation* by Michael Sipser.

[2] What do P and NP stand for? As you can guess, P stands for "polynomial," because these problems can be solved in polynomial time. But NP does *not* stand for "not polynomial," but rather, "nondeterministic polynomial time." We will not go into the details; again, I recommend Sipser's textbook.

Polynomial-time reductions. Let A and B be decision problems. A *polynomial-time reduction* from A to B is a polynomial-time algorithm f that transforms an instance X of A into an instance $f(X)$ of B such that X is a "yes" instance for A if and only if $f(X)$ is a "yes" instance for B. We write $X \in A$ to denote that X is a "yes" instance for A; thus, the reduction f must satisfy the following: $X \in A$ if and only if $f(X) \in B$. If such a reduction exists, we write $A \leq_P B$.

Intuitively, if $A \leq_P B$, then A is at most as hard as B. This is because any polynomial-time algorithm ALG_B for B produces a polynomial-time algorithm for A as follows:

1. On an instance X of A, apply the reduction to convert X into $f(X)$.
2. Run ALG_B on $f(X)$ and return whatever it returns (i.e., "yes" or "no").

(There could be a faster algorithm for A, so A is at most as hard as B.)

The reduction itself only describes how to transform X into $f(X)$. But to prove that the reduction is correct, we must show $X \in A$ if and only if $f(X) \in B$, and this involves two proofs: the "forward" direction is "$X \in A \implies f(X) \in B$," and the "backward" direction is "$f(X) \in B \implies X \in A$." So finding a correct reduction can be tricky because the reduction must simultaneously work for both directions.[3]

Proving NP-completeness. A problem A is **NP-hard** if every problem in NP is reducible to A, and A is **NP-complete** if it is both in NP and NP-hard.[4] We think of NP-complete problems as the "hardest" problems in NP, because a polynomial-time algorithm for any of them would produce (as described above) a polynomial-time algorithm for every problem in NP. It is not obvious that NP-complete (or NP-hard) problems even exist, but all of the remaining problems in this chapter are NP-complete.

The typical problem in complexity asks you to prove that a problem B is NP-complete. Usually, showing that B is in NP is straightforward, so we won't focus on this. To show that B is NP-hard, it suffices to prove that $A \leq_P B$ for some NP-hard problem A (we leave this proof as an exercise).

7.2 3-SAT to Independent Set

In this section, we give a polynomial-time reduction from the 3-Sat problem to the IndSet problem, defined below:

- 3-Sat. Input: A set of clauses over n Boolean variables $x_1, \ldots, x_n \in \{0, 1\}$. Each clause is the disjunction ("OR") of 3 literals, where a literal is a variable or the negation of a

[3] One might think that to show $A \leq_P B$, we can assume that we are given a polynomial-time algorithm for B, and we must use this to design a polynomial-time algorithm for A. In other words, in the algorithm for problem A above, why can we only use ALG_B once, and why are we forced to immediately output whatever it outputs? This is a fair question; in fact, this kind of reduction is known as a Cook reduction, while our definition is known as a Karp reduction. However, for reasons beyond the scope of these notes, we must stick with Karp reductions: we can use ALG_B once, and we must output whatever it outputs.

[4] There are NP-hard problems outside NP, such as the Halting Problem. The other direction is addressed by a result known as Ladner's theorem, which states that if P $\neq$ NP, then there are problems in NP that are not in P but also not NP-complete. But again, nobody has proven P $\neq$ NP.

variable. (For example, $x_1 \vee \overline{x_2} \vee x_5$ is one possible clause.) A clause is *satisfied* if at least one of its literals is set to 1. Question: Does there exist an assignment of the x_i variables to 0/1 (i.e., False/True) such that all clauses are satisfied?

- IndSet. Input: An undirected graph G and value k. An *independent set* is a subset S of vertices such that no two vertices in S share an edge. Question: Does G contain an independent set of size at least k?

Recall that we need to do three things: transform an instance of 3-Sat (i.e., a set of clauses) into an instance of IndSet (i.e., a graph G and a value of k), prove the "forward direction" (if there is a satisfying assignment for the 3-Sat instance, then G has an independent set of size at least k), and prove the "backward direction" (the converse).

Theorem 7.1. *The* 3-Sat *problem is reducible to* IndSet.

Proof. Suppose the 3-Sat instance has ℓ clauses $C_1, \ldots, C_\ell$. For each C_i, we add a triangle of vertices v_i^1, v_i^2, v_i^3 to G for a total of 3ℓ vertices and 3ℓ edges so far. So vertex v_i^j in G represents the j-th literal in clause C_i. For any two vertices in G, if their correspond literals are negations of each other, then we add an edge between these two vertices. This completes the construction of G; for k, we simply set its value equal to ℓ.

To prove the forward direction, suppose there is an assignment of each x_i to 0/1 such that all k clauses are satisfied. This means that in each clause, one literal is set to 1. (If multiple literals are 1, we pick one arbitrarily.) Let S be the corresponding set of vertices, so $|S| = \ell$; we claim that S is independent. For contradiction, suppose $u, v \in S$ and (u, v) is an edge of G. This edge is not a "triangle" edge, so it must be a "negation" edge, which means u represents a variable x_i and v represents its negation $\overline{x_i}$ (or vice versa). By definition of S, the assignment sets both x_i and $\overline{x_i}$ to 1, contradicting the validity of the assignment.

Now we prove the backward direction. Let S be an independent set of size ℓ; we must assign each variable x_i to 0/1. Recall that there is a one-to-one relation between vertices of G and literals in clauses. If neither x_i nor $\overline{x_i}$ is in S (as a vertex), then we can set x_i to either 0 or 1. If $x_i \in S$, then $\overline{x_i} \notin S$ because S is independent, so in this case, we can set $x_i = 1$. Similarly, if $\overline{x_i} \in S$, then we set $x_i = 0$ (so $\overline{x_i} = 1$). We claim that this assignment satisfies every clause. Recall that $|S| = \ell$, so S contains one vertex from each of the ℓ triangles in G. In each triangle, the literal corresponding to the vertex in S is set to 1, so the clause corresponding to that triangle is satisfied. □

7.3 Vertex Cover to Dominating Set

Our next reduction is from the Vertex Cover problem to the Dominating Set problem. On the surface, the two problems look similar (both seek to find a subset of vertices that "cover" elements of the graph); our reduction will follow this intuition.

- VertexCover. Input: An undirected graph G and value k. A *vertex cover* is a subset S of vertices such that every edge of G is incident to at least one vertex in S. Question: Does G contain a vertex cover of size at most k?

- DomSet. Input: An undirected graph G and value k. A *dominating set* is a subset S of vertices such that every vertex of G is either in S or is adjacent to at least one vertex in S. Question: Does G contain a dominating set of size at most k?

Theorem 7.2. *The* VertexCover *problem is reducible to* DomSet.

Proof. Let (G, k) denote an instance of VertexCover; we shall construct an instance (G', k') of DomSet. Initialize G' as G. Additionally, for each edge $e = (u, v)$ of G, we add a vertex x_e to G, as well as the edges (u, x_e) and (v, x_e). So we can imagine G' as being G with an additional copy of each edge, and there is a new vertex placed "on" each copy. We'll call these new vertices "edge" vertices. Finally, we set $k' = k$.

Now we prove the forward direction. Let S be a vertex cover of G such that $|S| = k = k'$; we claim that S is a dominating set in G'. Consider any vertex u in G'. If $u \notin S$, then consider any edge (u, v) incident to u. Since S is a vertex cover that doesn't contain u, it must contain v. Thus, u is adjacent to $v \in S$, so S is a dominating set.

Now we prove the backward direction. Let S' be a dominating set of G' such that $|S'| = k' = k$. Notice that the vertices in S' are not necessarily in G because G' contains the "edge" vertices. However, consider any "edge" vertex $x_e \in S'$, where $e = (u, v)$. Let's remove x_e from S' and replace it with u (or v). (If both u and v were already in S', then we can remove x_e from S' and S' remains a dominating set.) Then S' is still dominating because the three vertices dominated by x_e (x_e, u, v) are also dominated by u.

Thus, we can replace all "edge" vertices in S' with either of their endpoints and S' remains a dominating set. Now we can claim that S' is a vertex cover of G. Consider any edge $e = (u, v)$ in G. Since the vertex x_e in G' is dominated by S', at least one of its neighbors $\{u, v\}$ must be in S', so e is incident to at least one vertex in S'. □

7.4 Subset Sum to Scheduling

Our final reduction is from a knapsack-like problem to a scheduling problem. Although the latter sounds complicated, in the reduction, the instance we construct is fairly simple.

- SubsetSum. Input: Positive integers $x_1, \ldots, x_n$ and a target value T. Question: Is there a subset of these integers that sums to exactly T?

- Schedule. Input: A set of jobs labeled $\{1, \ldots, n\}$, where job i has release time $r_i \geq 0$, deadline $d_i > r_i$, and length $\ell_i \geq 1$. (All values are integers.) There is 1 machine; we must allocate every job i to a contiguous block of length ℓ_i in the interval $[r_i, d_i]$. Question: Is there an allocation that finishes all jobs by their deadline?

Theorem 7.3. *The* SubsetSum *problem is reducible to* Schedule.

Proof. Given an instance of SubsetSum $(\{x_1, \ldots, x_n\}, T)$, we must construct an instance of Schedule. Let $X = \sum_{i=1}^{n} x_i$ denote the sum of the input integers. For each x_i, we create a job i with $[r_i, d_i] = [0, X + 1]$ and $\ell_i = x_i$. We also create a special $(n + 1)$-th job, with $[r_{n+1}, d_{n+1}] = [T, T + 1]$ and $\ell_{n+1} = 1$. This completes the reduction.

For the forward direction, suppose there exists $S \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in S} x_i = T$. In any feasible schedule, we must assign job $n + 1$ to the interval $[T, T + 1]$. The jobs in S have

total length T, so we can assign them (in any order) to finish exactly at time T. Finally, the amount of time between $T+1$ and $X+1$ is $X-T$, which is exactly the amount of time the remaining jobs require. Thus, we can order them arbitrarily in $[T+1, X+1]$.

Now we prove the backward direction. Again, any feasible schedule must assign job $n+1$ to $[T, T+1]$. This leaves X units of time to schedule the first n jobs, whose total length is X. So a feasible schedule cannot leave any idle time on the machine. In particular, the interval $[0, T]$ must be fully packed with jobs, so these jobs have total length T. Thus, the integers corresponding to these jobs must sum to exactly T. □

7.5 More Hard Problems

The following problems are all NP-complete, and many of the exercises will refer to them. Unless otherwise stated, all of the input graphs are connected and undirected. Note that the nomenclature for these problems is not consistent across all sources, so for example, you may find a book that uses "HamPath" to refer to a small variant of the problem we describe below. But in these cases, the problems are reducible to one another, so the names of the problem don't really matter.

- Clique. Input: A graph G and value k. A *clique* is a subset S of vertices such that there is an edge between every pair of vertices in S. Question: Does G contain a clique of size at least k?

- LongestPath. Input: A graph G, where each edge has some nonnegative length, two vertices s, t in G, and some value k. Question: Does there exist a path (i.e., no repeated vertices) from s to t of length at least k?

- HamPath. Input: A graph G and two vertices s, t in G. A *Hamiltonian path* is a path that visits every vertex exactly once. Question: Does G contain a Hamiltonian path from s to t?

- HamCycle. Input: A graph G. A *Hamiltonian cycle* is a cycle that visits every vertex exactly once. Question: Does G contain a Hamiltonian cycle?

- TSP (Traveling Salesman Problem). Input: A complete graph G where every edge has a nonnegative length, and a value k. Question: Does G contain a Hamiltonian cycle of total length at most k?

- 3D-Matching. Input: Three disjoint sets U, V, W, all of size n, and a set of m triples $E \subseteq U \times V \times W$. Question: Does there exist a subset S of n triples such that each element of $U \cup V \cup W$ is contained in exactly one triple in S?

- SetCover. Input: A set of n elements U, sets $S_1, \ldots, S_m \subseteq U$, and a value k. A *cover* is a collection of subsets whose union is U. Question: Does there exist a cover of size at most k?

- Coloring. Input: A graph G and value k. A *k-coloring* of G is a function $f : V \to \{1, \ldots, k\}$; it is *proper* if, for every edge (u, v), $f(u) \neq f(v)$. Question: Does G contain a proper k-coloring?

7.6 Exercises

1. Let A, B, C be decision problems.
   (a) Prove that if $A \leq_P B$ and $B \leq_P C$, then $A \leq_P C$.
   (b) Suppose A is NP-hard and $A \leq_P B$. Prove that B is NP-hard.
2. Suppose P = NP. Prove that every problem in P is NP-complete.
3. Prove IndSet $\leq_P$ VertexCover and vice versa.
4. Prove VertexCover $\leq_P$ SetCover.
5. Consider the reduction from VertexCover to DomSet in Sec. 7.3. Explain what goes wrong if we set $G' = G$ and $k' = k$.
6. Prove HamPath $\leq_P$ HamCycle and vice versa.
7. Consider the HamPath problem, but suppose s and t are not specified. Show that the HamPath problem reduces to this version, and vice versa.
8. Let G be an undirected graph. Question: Does G contain a spanning tree in which the degree of every vertex (in the tree) is at most 3? Prove that this problem is NP-hard. (Hint: You may want to use a result from the previous exercise.)
9. Let $x_1, \ldots, x_n$ be positive integers. Question: Does there exist a subset S of these integers such that the sum of integers in S is equal to the sum of integers not in S? Prove that this problem is NP-hard.
10. Let G be an undirected graph and k be a value. A *strongly independent set* (SIS) S is an independent set such that the distance between any two vertices in S is at least 3. Question: Does G contain an SIS of size at least k? Prove that this problem is NP-hard.
11. Consider the 3-Sat problem, but now the question is this: Are there *at least two* assignments of the x_i variables to 0/1 such that all clauses are satisfied? Prove that this problem is NP-hard.
12. The Coloring problem is NP-hard even for $k = 3$. Consider the following variant: suppose the number of vertices in the input graph is a multiple of 3, and our goal is to find a proper 3-coloring that uses each color the same number of times. Prove that this problem is also NP-hard.
13. Suppose we are given an oracle that solves the HamPath (decision) problem in polynomial time. Use this oracle to design a polynomial-time algorithm that returns an s-t Hamiltonian Path (if one exists).
14. Suppose we are given an oracle that solves the Clique (decision) problem in polynomial time. Use this oracle to design a polynomial-time algorithm that finds a clique of maximum size. Do the same for VertexCover (i.e., find a minimum vertex cover).

8 Approximation Algorithms

In Ch. 7, 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 use 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, we consider $c > 1$, so ALG satisfies

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

For maximization problems, we flip all three inequality symbols above (so $c < 1$ 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 6.2, we saw a 2-approximation algorithm for the vertex cover problem. In this chapter, we'll see a few more examples, starting with another 2-approximation algorithm for the same problem.

8.1 Vertex Cover

Problem statement. Hopefully, if you've read everything so far, 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 in E 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 8.1. *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. □

8.2 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 quite simple, and greedy: iterate through the set of jobs in any order, assigning each one to the machine with the smallest load so far.[2]

Theorem 8.2. *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 j denote the last job assigned to i, and consider the moment in the algorithm immediately before it assigned j to i. At this point, the load on i was $T - \ell_j$, and because we chose to assign j greedily, this was the smallest load among the m machines. Thus, the total load is at least $m \cdot (T - \ell_j)$. Combining this with our bounds on T^*, we have

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

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

8.3 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 8.3. *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]Intuitively, we can think of 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. □

8.4 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 8.4. *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.

8.5 Exercises

1. Consider the following greedy algorithm for the vertex cover problem (Sec. 8.1): 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.

2. Consider the greedy algorithm for the load balancing problem from Sec. 8.2.

   (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.

3. For the load balancing problem (Sec. 8.2), give a 3/2-approximation algorithm that runs in $O(n^2)$ time. (Hint: The algorithm is similar to the greedy algorithm.)

4. Show how the algorithm in Sec. 8.3 can be implemented in $O(nk)$ time.

5. In this exercise, we give another proof of Theorem 8.3.

   (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.

6. Suppose there exists a polynomial-time α-approximation algorithm for the metric k-center problem (Sec. 8.3), where $\alpha < 2$. Show how we can use this algorithm to solve the DomSet problem from Sec. 7.3 in polynomial time.

7. Prove that the algorithm in Sec. 8.4 always terminates. Then prove that the algorithm terminates in $O(n^4)$ time if every edge has weight 1.

8. 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. 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. (Hint: This is similar to the previous exercise.)

10. Recall the minimum cut problem from Ch. 5. 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 all k terminals into different connected components. Give a $(2 - 1/k)$-approximation algorithm for this problem. (Hint: Start with the goal of a 2-approximation, and use a minimum s-t cut algorithm multiple times.)