Notes on Algorithms

Kevin Sun

Preface

These notes cover topics in algorithms at a standard undergraduate level. They assume familiarity with fundamental programming concepts (e.g., arrays, loops), discrete math (e.g., basic set theory, graphs), and asymptotic notation. I recommend the textbooks below for further reading; these notes are primarily based on them:

- *Introduction to Algorithms* by Cormen, Leiserson, Rivest, and Stein
- *Algorithms* by Dasgupta, Papadimitriou, and Vazirani
- *Algorithms* by Erickson
- *Algorithm Design* by Kleinberg and Tardos
- *The Algorithm Design Manual* by Skiena

Of course, there are many other excellent textbooks and resources not listed above, and I encourage you to seek them out.

— Kevin Sun
Initially written: June 2021
Last updated: June 2024

Preliminaries

Unless specified otherwise, we assume the following:

- The set of integers is $\mathbb{Z}$, and the set of positive integers is $\mathbb{Z}^+$.
- For all $n \in \mathbb{Z}^+$, $[n] = \{1, 2, \ldots, n\}$, and for all $a, b \in \mathbb{Z}$, $[a, b] = \{a, a+1, \ldots, b\}$.
- For every array A, the length of A is n, the indices of A range from 1 through n, and $A[i : j]$ is the subarray $[A[i], A[i+1], \ldots, A[j]]$.
- For every graph G, the set of vertices is $V(G) = [n]$ (or just V if G is clear from the context). The set (not multiset) of edges is $E(G)$ (or just E). Every edge has two distinct endpoints; if G is directed, (u, v) and (v, u) can both be edges.
- Every graph G is represented in adjacency list format. More specifically, G is an array of length n, and for all $u \in V$, $G[u]$ is a list of the (out-)neighbors of u.

Correctness proofs

The purpose of a correctness proof is to convince the reader that an algorithm is correct. In these notes, we prioritize communicating the main ideas behind each algorithm in a clear and concise manner, not expanding on every mathematical detail.

Running time

The "running time" of an algorithm is the maximum number (as a function of the input length) of primitive operations that it executes. If the input has length n, then a $T(n)$-time algorithm executes at most $T(n)$ primitive operations on that input. Each of the following is a primitive operation:

- Assigning a value to a variable (e.g., $x \leftarrow 0$)
- Performing a comparison (e.g., $x > y$)
- Performing an arithmetic operation (e.g., $x + y$)
- Indexing into an array (e.g., accessing $A[3]$)
- Calling or returning from a method

Calculating the exact number of primitive operations executed by an algorithm can be tricky. For example, since we won't work with a totally precise model of computation, we can't even count exact number of primitive operations in a simple for-loop. But throughout these notes, we'll use asymptotic notation, which allows us to bypass the need to meticulously count primitive operations. It is also ubiquitous in theoretical computer science, so for further reading, it is worthwhile to become familiar with it.

1 Array Algorithms

Every chapter begins with an overview. Recall that unless stated otherwise, every array has length n, and the indices range from 1 through n.

Overview

Section	Summary	Time
1.1: Max in Array	Scan A while updating the maximum seen so far.	$O(n)$
1.2: Two Sum	Set $(i, j) = (1, n)$; if $A[i] + A[j] \neq t$, increment i or decrement j as necessary.	$\uparrow$
1.3: Binary Search	Check the middle element; if it's not t, recurse on either the left or right half of the A.	$O(\log n)$
1.4: Selection Sort	For $i \in [n]$, swap the smallest element in $A[i:n]$ with $A[i]$.	$O(n^2)$
1.5: Merge Sort	Split A in two halves, recursively sort each half, and merge using two pointers.	$O(n \log n)$

1.1 Max in Array

Problem Statement

The input is an array A of n distinct, positive integers. Our goal is to return the largest integer in A. Describe an $O(n)$-time algorithm for this problem, prove that it's correct, and analyze its running time.

```
Max-in-Array($A$):
1 $m = A[1]$
2 for $i = 2, \ldots, n$:
3     if $A[i] > m$:
4         $m = A[i]$
5 return $m$
```

Algorithm: Set $m = A[1]$. Scan A and update m by setting $m = A[i]$ whenever $A[i] > m$. (Alternatively, we could set $m = \max(m, A[i])$ in every iteration.) Return m.

Correctness: At any point in the algorithm, $m = A[i]$ for some $i \in [n]$, so if the largest integer in A is $A[i^*]$, at the beginning of the iteration where $i = i^*$, we must have $A[i^*] > m$. So in this iteration, the algorithm sets $m = A[i^*]$. In the subsequent iterations, m does not change because $A[i^*]$ is the largest integer in A. Thus, the algorithm returns $A[i^*]$, as desired.

Running time: The algorithm makes $n - 1$ iterations, and each iteration takes $O(1)$ time, so the total running time is $O(n)$.

1.2 Two Sum

For the Two Sum problem, the "brute-force" algorithm takes $O(n^2)$ time, but it does not take advantage of the fact that the input array A is sorted, which allows us to obtain an $O(n)$-time algorithm.

Problem Statement

The input is (A, t), where A is an array of n distinct integers sorted in increasing order and $t \in \mathbb{Z}$. Our goal is to return indices (i, j) such that $i < j$ and $A[i] + A[j] = t$ (or nothing if no such (i, j) exists). Describe an $O(n)$-time algorithm for this problem, prove that it's correct, and analyze its running time.

```
Two-Sum($A, t$):
$i, j = 1, n$
while $i < j$:
    if $A[i] + A[j] = t$:
        return $(i, j)$
    else if $A[i] + A[j] < t$:
        $i \mathrel{+}= 1$
    else:
        $j \mathrel{-}= 1$
```

Algorithm: We use a "two-pointer" technique: Starting with $(i, j) = (1, n)$ as our first "candidate" solution, we check if $A[i] + A[j] = t$. If so, we're done. If not, we either increment i (if $A[i]+A[j]$ is too small) or decrement j (if $A[i] + A[j]$ is too big). Either way, we keep checking and adjusting until we return a solution or $i = j$.

Correctness: If the algorithm returns some (i, j), then it is a valid solution. Conversely, suppose the input (A, t) has at least one solution, and let (i^*, j^*) denote the one with the smallest value of i^*. If $(i^*, j^*) = (1, n)$, then the algorithm immediately returns $(1, n)$. If not, then i and j approach each other until $j = j^*$ or $i = i^*$. If $j = j^*$ occurs first, then at this point, $i < i^*$, so $A[i] + A[j] < A[i^*] + A[j^*] = t$ since the integers in A are sorted and distinct. Thus, j "waits" at j^* while the algorithm increases i until $i = i^*$. If $i = i^*$, then i waits at i^* instead. Either way, the algorithm finds (i^*, j^*) and returns it.

Running time: Notice that $j - i$ is initially $n - 1$, decreases by 1 in each iteration, and cannot decrease below 0. Thus, the algorithm makes $O(n)$ iterations. Each iteration takes $O(1)$ time, so the total running time is $O(n)$.

1.3 Binary Search

Here is another example of speeding up an algorithm by taking advantage of the fact that the input array is sorted.

Problem Statement

The input is (A, t), where A be an array of n distinct integers sorted in increasing order and $t \in \mathbb{Z}$. Our goal is to return an index k such that $A[k] = t$ (or nothing if no such k exists). Describe an $O(\log n)$-time algorithm for this problem, prove that it's correct, and analyze its running time.

```
Binary-Search(A, t):
1 $i, j = 1, n$
2 while $i \le j$:
3     $m = \lfloor (i+j)/2 \rfloor$
4     if $A[m] = t$:
5         return $m$
6     else if $A[m] < t$:
7         $i = m + 1$
8     else:
9         $j = m - 1$
```

Algorithm: Check if $A[m] = t$, where m is the middle index of A. If $A[m] < t$, recurse on the right half of A (i.e., $A[m+1:n]$ becomes the new input array); otherwise, recurse on the left half.

Correctness: If the algorithm returns some m, then it must satisfy $A[m] = t$. Conversely, assume $t \in A$; we must prove that the algorithm returns m such that $A[m] = t$.

The algorithm always "focuses" on a subarray $A[i:j]$. Notice that $A[i:j]$ always contains t, and $A[i:j]$ shrinks in each iteration. So by the iteration in which $A[i:j]$ only contains t, the algorithm returns t.

Running time: We claim that if the subarray $A[i:j]$ has length ℓ at the beginning of some iteration and length ℓ' at the end of the same iteration, then $\ell' \le \ell/2$. Assuming that the algorithm sets $i = m+1$ in this iteration,

$$\ell' = j - (m+1) + 1 = j - \left\lfloor \frac{i+j}{2} \right\rfloor \le j - \frac{i+j}{2} + \frac{1}{2} = \frac{j-i+1}{2} = \frac{\ell}{2}.$$

We can similarly show $\ell' \le \ell/2$ if the algorithm sets $j = m-1$ in this iteration. So either way, since $A[i:j]$ initially has length n, after k iterations, $A[i:j]$ has length at most $n/2^k$. Thus, the algorithm terminates within $O(\log n)$ iterations, and each iteration takes $O(1)$ time, so the total running time is $O(\log n)$.

1.4 Selection Sort

Selection Sort is one of many $O(n^2)$-time sorting algorithms.

Problem Statement

The input is an array A of n distinct integers. Our goal is to sort A by increasing value (i.e., rearrange A such that $A[1] < A[2] < \cdots < A[n]$). Describe an $O(n^2)$-time algorithm for this problem, prove that it's correct, and analyze its running time.

```
Selection-Sort(A):
1 for $i = 1, \ldots, n$:
2     $m = i$
3     for $j = i+1, \ldots, n$:
4         if $A[j] < A[m]$:
5             $m = j$
6     swap $A[i], A[m]$
```

Algorithm: The algorithm executes n rounds, one for each $i \in [n]$. In round i, it finds the smallest element $A[m]$ in $A[i:n]$ and swaps it with $A[i]$.

Correctness: We claim that after round i, $A[1:i]$ contains the i smallest elements of A in sorted order. The correctness of the algorithm follows from this claim when $i = n$. The base case is round $i = 1$, in which the algorithm sets $A[1]$ to be the smallest element of A. In Lines 2–5 of round $i+1$, the algorithm finds the smallest element in $A[i+1:n]$, which is the $(i+1)$-th smallest element of A (since $A[1:i]$ contains i smaller elements). In Line 6, it places this element at index i. The result is that at the end of round $i+1$, $A[1:i+1]$ contains the $i+1$ smallest elements of A in sorted order, as desired.

Running time: In round i, the algorithm executes at most $c(n-i)$ operations for some $c \in \mathbb{Z}^+$. Summing over all $i \in [n]$, we see that the total number of operations is at most

$$c\sum_{i=1}^{n}(n-i) = c \cdot ((n-1)+(n-2)+\cdots+2+1) = O(n^2).$$

1.5 Merge Sort

Problem Statement

The input is an array A of n integers. Our goal is to sort A by non-decreasing value (i.e., rearrange A such that $A[1] \leq A[2] \leq \cdots \leq A[n]$). Describe an $O(n \log n)$-time algorithm for this problem, prove that it's correct, and analyze its running time.

Algorithm: The algorithm is recursive: if $n = 1$, simply return A. Otherwise, split A into its left and right halves and recursively sort each half. Then *merge* the two halves using a two-pointer approach.

```
Merge-Sort($A$):
1  if $n = 1$:
2      return $A$
3  $k = \lfloor n/2 \rfloor$
4  $A_L$ = Merge-Sort($A[1:k]$)
5  $A_R$ = Merge-Sort($A[k+1:n]$)
6  $i, j, B = 1, 1,$ [empty list]
7  while $i \leq k$ and $j \leq n-k$:
8      if $A_L[i] \leq A_R[j]$:
9          append $A_L[i]$ to $B$; $i$ += 1
10     else:
11         append $A_R[j]$ to $B$; $j$ += 1
12 if $i > k$:
13     append each element of $A_R[j:n-k]$ to $B$
14 else:
15     append each element of $A_L[i:k]$ to $B$
16 $A = B$ (as an array)
17 return $A$
```

Correctness: Assuming (by induction) that A_L and A_R are each sorted, notice that the merge process combines them into a sorted list B. More specifically, if $A_L[i] \leq A_R[j]$, then $A_L[i]$ is the smallest integer in the two subarrays, so it should be the next element appended to B (Line 9). The same reasoning holds if $A_R[j] < A_L[i]$. This continues until one of the subarrays becomes empty. If

$i > k$, $A_L[i:k]$ is empty, so $A_R[j:n-k]$ contains all of the remaining elements of A in sorted order, so we should append them to B (Line 13). The same reasoning holds if $j > n-k$.

Remark

Merge Sort is one of the most famous "divide-and-conquer" algorithms. A typical divide-and-conquer algorithm splits the input into disjoint parts, recursively solves the problem on each part, and combines the solutions to solve the original problem.

Running time: Let $T(n)$ denote the running time of the algorithm on an input of size n. In Lines 4 and 5, we call the algorithm itself on twice, where each input has size $n/2$.[1] These two calls together contribute $2 \cdot T(n/2)$ time to the total running time. After the recursive calls, in each iteration of the while-loop, $i+j$ increases by 1. Since $i+j \leq n$, Lines 7–11 take $O(n)$ time. Appending the remaining elements of A_R or A_L takes $O(n)$ time. Putting this all together, we have

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn$$

for some constant c; assuming $T(1) \leq k$ for some constant $k \geq c$, we can prove $T(n) \leq kn(\log n + 1)$ by induction:

$$\begin{aligned} T(n) &\leq 2 \cdot T\left(\frac{n}{2}\right) + cn \\ &\leq 2 \cdot \frac{kn}{2}\left(\log \frac{n}{2} + 1\right) + cn \\ &\leq kn \log n + kn \\ &= kn(\log n + 1). \end{aligned}$$

Thus, $T(n) \leq kn \cdot (2 \log n) = O(n \log n)$, as desired.

Remark

Selection Sort and Merge Sort are both comparison-based sorting algorithms (i.e., they only obtain information about elements in A by comparing them to each other). Every comparison-based sorting algorithm has running time $\Omega(n \log n)$. By making additional assumptions about the input, we can obtain faster algorithms. For example, if every element in A is in $\{1, 2, 3\}$, we can sort A in $O(n)$ time: partition A into 3 lists and combine them.

[1] For simplicity, we assume $n/2$ is always an integer; the full analysis uses similar reasoning.

2 Essential Graph Algorithms

The algorithms in this section are essential building blocks in the study of graph algorithms. Note that the adjacency list representation of any graph has size $\Theta(m+n)$, so every algorithm in this chapter runs in linear time.

Overview

Section	Summary	Time
2.1: BFS	Process vertices in "layers" via a queue: start with s, then out-neighbors of s, etc.	$O(m+n)$
2.2: DFS	Set $\mathsf{pre}[u]$ when we start exploring u, set $\mathsf{post}[u]$ when we stop, backtrack when stuck.	$\uparrow$
2.3: Cycle Finding	Run DFS, find a back edge (u,v) and the v-u path P in the DFS tree, return $P+(u,v)$.	$\uparrow$
2.4: Topological Ordering	Order V by decreasing post-value.	$\uparrow$
2.5: SCCs	Run DFS(G^R), run BFS(G,u) for each u in decreasing order of $\mathsf{post}[u]$.	$\uparrow$

2.1 Breadth-First Search

Finding shortest paths is a classic problem. We start with a relatively simpler version; we'll consider more complicated situations in Chapter 5. Note that the distance from s to v is the length of the shortest path from s to v.

Problem Statement

The input is (G,s), where G is a directed graph and $s \in V$. Our goal is to return an array d such that for all $v \in V$, $d[v]$ is the distance from s to v. Describe an $O(m+n)$-time algorithm for this problem, prove that it's correct, and analyze its running time.

```
BFS(G, s):
d = [∞] * n; d[s] = 0
Q = queue(s)
while |Q| ≥ 1:
    u = dequeue from Q
    for v in G[u]:
        if d[v] = ∞:
            d[v] = d[u] + 1; add v to Q
return d
```

Algorithm: Intuitively, we start at s and work our way through the graph in "layers" by visiting the out-neighbors of s, the out-neighbors of those vertices, etc.

More formally, create a queue Q with just s in it and set $d[s] = 0$. While Q is not empty: dequeue a vertex u from Q and process it. "Processing u" means looping through each out-neighbor v of u and adding v to Q if v has not been encountered before; if v gets added to Q, set $d[v] = d[u] + 1$.

Correctness: Suppose the distance from s to some vertex v is k; we want to show that at the end of BFS, $d[v] = k$. If $k = 0$, then $s = v$ and $d[v] = 0$, as desired. If $k > 0$, then by induction on k, there exists an in-neighbor u of v such that BFS set $d[u] = k - 1$ and added u to Q. When BFS later removed u from Q, it set $d[v] = d[u] + 1 = k$.

Running time: Initializing d and Q takes $O(n) + O(1)$ time. Every vertex gets added to Q at most once, so the while-loop makes at most n iterations. It seems like processing u takes $\Omega(n)$ time (since u could have $\Omega(n)$ out-neighbors), which results in a total running time is $O(n^2)$. This is correct, but the analysis can be improved: processing u only requires $O(\text{out-deg}(u))$ time, and $\sum_{u \in V} \text{out-deg}(u) = m$, so the overall running time is $O(m + n)$.

Remark

To find the shortest *path* (not just distance) from s to any vertex, we can maintain an array p of parent pointers as we run BFS. In particular, whenever we set $d[v] = d[u] + 1$, we also set $p[v] = u$. This indicates that in a shortest s-v path, u appears immediately before v. By following p from v to s, we obtain a shortest path from s to v (in reverse order). Tracing parent pointers will be useful for many other problems.

2.2 Depth-First Search

BFS is like water spreading across the surface of a table. In contrast, Depth-First Search (DFS) is like running in a maze and leaving behind a trail of breadcrumbs. Whenever we reach a fork in the road, we pick a direction and continue until we get stuck, at which point we backtrack along the breadcrums and try another direction. In this section, we'll describe DFS without any particular problem in mind, and in the next two sections, we'll give applications.

```
Explore($G, u$):
1 pre$[u] = t$; $t$ += 1
2 for $v$ in $G[u]$:
3     if pre$[v] = \infty$:
4         Explore($G, v$)
5 post$[u] = t$; $t$ += 1
```

Algorithm: Depth-First Search uses a recursive "Explore" procedure, which popoulates two arrays, pre and post. If we start exploring a vertex u at time t, we set pre$[u] = t$, and if we stop exploring at time t', we set post$[u] = t'$. DFS itself is simply a wrapper around Explore. For each vertex u, if u has not been explored, DFS calls Explore(G, u). While exploring u, we also explore all vertices reachable from u (if they have not already been explored).

```
DFS($G$):
1 pre, post, $t$ = $[\infty] * n, [\infty] * n, 1$
2 for $u$ in $V$:
3     if pre$[u] = \infty$:
4         Explore($G, u$)
5 return pre, post
```

Classifying edges: After we run DFS, we can classify every edge $(u, v) \in E$ into one of four types: tree, forward, back, cross. Tree edges were directly traversed by DFS, and their union is called the *DFS tree*. (Note that a "DFS tree" could actually be a forest containing multiple trees rooted at different vertices.) The other three types of edges are all determined with respect to the DFS tree T: (u, v) is a *forward* edge if there is a path from u to v in T, (u, v) is a *back* edge if there is a path from v to u in T, and (u, v) is a *cross* edge if it is not a tree, forward, or back edge.

Remark

Observe that (u, v) is a back edge if and only if $\mathsf{pre}[v] < \mathsf{pre}[u] < \mathsf{post}[u] < \mathsf{post}[v]$. Similarly, (u, v) is a cross edge if and only if $\mathsf{pre}[v] < \mathsf{post}[v] < \mathsf{pre}[u] < \mathsf{post}[u]$. If $\mathsf{pre}[u] < \mathsf{pre}[v] < \mathsf{post}[v] < \mathsf{post}[u]$, then (u, v) is either a tree edge or a forward edge. No other ordering of these four values can exist.

2.3 Cycle Finding

Problem Statement

The input is a directed graph G. Our goal is to return a directed cycle in G (or nothing if none exists). Describe an $O(m + n)$-time algorithm for this problem, prove that it's correct, and analyze its running time.

```
Find-Cycle($G$):
run DFS($G$); $T$ = DFS tree
for $u$ in $V$:
    for $v$ in $G[u]$:
        if $(u, v)$ is a back edge:
            $P$ = $v$-$u$ path in $T$
            return $P + (u, v)$
```

Algorithm: Run DFS to obtain pre- and post-values and a DFS tree T. If there is a back edge (u, v), then there is a path P from v to u in T, so we can return $P + (u, v)$ as a cycle in G.

Correctness: If (u, v) is a back edge, then (by definition) there exists a path P from v to u in the DFS tree T, so $P + (u, v)$ is a cycle in G. Conversely, suppose the graph has a cycle C. Let v denote the first vertex in C explored by DFS, and let u denote the vertex preceding v in C. Then during Explore(v), we call Explore(u), resulting in (u, v) being a back edge.

Running time: Running DFS and constructing T take $O(m + n)$ time. Then we look for a back edge, which takes $O(1)$ time per edge. If we find one, we compute P in $O(n)$ time using BFS (or DFS) in T. Since T only has $n - 1$ edges, BFS(T) takes $O(n)$ time, so the total running time is $O(m + n) + m \cdot O(1) + O(n) = O(m + n)$.

2.4 Topological Ordering

A *topological ordering* of a directed graph $G = (V, E)$ is an ordering R of V such that for all $(u, v) \in E$, u appears before v in R. We can visualize R as an arrangement of V in a horizontal line such that every edge points from left to right. If G has a cycle (which we can detect in $O(m + n)$ time using the algorithm described above), then G does not have a topological ordering. So for the sake of this problem, we can assume that the input is a directed acyclic graph (DAG).

Problem Statement

The input is a DAG G. Our goal is to return a topological ordering of G. Describe an $O(m + n)$-time algorithm for this problem, prove that it's correct, and analyze its running time.

Algorithm: In short, we return V in order of decreasing post value. More specifically, we run a slight modification of DFS on G: Whenever we set $\mathsf{post}[u]$, we add u to the front of a list R (initially empty). At the end, we return R.

```
Topological-Sort($G$):
1 $R$ = [empty list]
2 DFS($G$)
  /* After post[$u$] = $t$, append
     $u$ to front of $R$        */
3 return $R$
```

Correctness: Consider any edge (u, v); it suffices to show that $\mathsf{post}[u] > \mathsf{post}[v]$. Note that there are two possible cases: we call Explore(u) before calling Explore(v), or vice versa. In the first case, while exploring u, we will explore v because v is an out-neighbor of u. Thus, we do not set $\mathsf{post}[u]$ until after we set $\mathsf{post}[v]$, so $\mathsf{post}[u] > \mathsf{post}[v]$. In the second case, since G is a DAG and $(u, v) \in E$, u is not reachable from v. So we do not explore u until after we finish exploring v, which also implies $\mathsf{post}[u] > \mathsf{post}[v]$.

Running time: Appending to a list takes $O(1)$ time, so this algorithm has the same asymptotic running time as DFS, which is $O(m + n)$.

Remark

Any algorithm can be implemented in multiple ways. For example, Topological-Sort can also be implemented as follows: run DFS to compute all post-values, then sort V by decreasing post-values. This implementation is arguably easier to describe, but its running time depends on that of the sorting algorithm.

2.5 Strongly Connected Components

A directed graph $G = (V, E)$ is *strongly connected* if, for every $u, v \in V$, G contains a u-v path and a v-u path. Notice that the strongly connected components of G partition V. Thus, a strongly connected component (SCC) in a directed graph is analogous to a connected component in an undirected graph. (Notice that G is a DAG if and only if every SCC is just a single vertex.)

Problem Statement

The input is a directed graph G. Our goal is to return an array c such that for all $u, v \in V$, u and v are in the same SCC if and only if $c[u] = c[v]$. Describe an $O(m + n)$-time algorithm for this problem, prove that it's correct, and analyze its running time.

Algorithm: Construct the reverse graph G^R of G by reversing every edge (i.e., $(u, v) \in E(G)$ becomes $(v, u) \in E^R(G)$). Run DFS on G^R to obtain $\mathsf{post}[u]$ for every $u \in V$. Then, for each $u \in V$ in order of decreasing $\mathsf{post}[u]$, run BFS to find the vertices reachable from u in G, and label these as a strongly connected component.

Correctness: Consider the graph G_{SCC}, where each vertex is a connected component C_i of G, and (C_i, C_j) is an edge if there exists $(u, v) \in E(G)$ such that $u \in C_i$ and $v \in C_j$. Notice that G_{SCC} is a DAG.

```
**SCC**($G$):
1 $G^R$ = reverse of $G$
2 pre, post = DFS($G^R$)
3 $c, k = [\infty] * n, 1$
4 for $u \in V$ in decreasing order of post[$u$]:
5     if $c[u] = \infty$:
6         BFS($G, u$);  $k$ += 1
          /* Set $c[v] = k$ for all $v$
             reached from $u$          */
7 return $c$
```

We call C a *source* SCC if no edge in G_{SCC} enters C, and we call C a *sink* SCC if no edge in G_{SCC} leaves C. Ideally, we would call BFS(G, u) where u is in a sink SCC and label all vertices reached from u as an SCC. Unfortunately, there is no simple way to find a vertex in a sink SCC.

However, observe the following: suppose H is a directed graph, we obtain post-values by calling DFS(H), and we set $\mathsf{post}[C] = \max_{u \in C} \mathsf{post}[u]$ for every SCC C of H. Then, if we order the SCCs by decreasing post-values, the result is a topological ordering of H_{SCC}. (This is a generalization of the result in Section 2.4.)

By the observation above, the algorithm processes G^R_{SCC} in topological order, which is equivalent to processing G_{SCC} reverse topological order. Thus, whenever the algorithm calls BFS, it labels (ignoring vertices that have already been labeled) the vertices of some sink SCC of G.

Running time: Constructing G^R and running DFS takes $O(m+n)$ time. We then run BFS from one vertex u per SCC; notice that this run of BFS only processes vertices that are reachable from u and haven't been processed before. Thus, the total running time is $O(m+n)$.

3 Greedy Algorithms

An algorithm is greedy if it iteratively constructs a solution by choosing the option that *appears* optimal without considering how current decisions affect future options.

Overview

Section	Summary	Time
3.1: MST	**Prim's:** Starting with $S = \{1\}$, expand S by adding the lightest edge crossing S. **Kruskal's:** Keep adding the lightest edge without creating a cycle. **Reverse-Delete:** Keep deleting the heaviest edge without disconnecting G.	$O(m^2)$
3.2: Selecting Compatible Intervals	Sort by non-decreasing end times, pick compatible intervals in this order.	$O(n \log n)$
3.3: Fractional Knapsack	Sort items by non-decreasing $v[i]/w[i]$, pick as much of each item as possible in this order.	$\uparrow$

3.1 Minimum Spanning Tree

A *spanning tree* of an undirected graph $G = (V, E)$ is a graph $T = (V, F)$ such that $F \subseteq E$ and T is connected; T is a *minimum spanning tree* (MST) if $\sum_{e \in F} w(e)$ is as small as possible.

Problem Statement

The input is a connected, undirected graph $G = (V, E)$ where each edge e has a distinct weight $w(e) \in \mathbb{Z}$. Our goal is to return a minimum spanning tree of G. Describe an $O(m^2)$-time algorithm for this problem, prove that it's correct, and analyze its running time.

We can incorporate weights (and other edge attributes) into the adjacency list representation of G as follows: for every $u \in V$ and neighbor v of u, $G[u]$ contains a pair (v, w) (rather than just v), where w is the weight of the edge $\{u, v\}$.

Remark

Assuming that the edge weights in G are distinct allows us to prove (by an exchange argument similar to the ones in this section) that G has exactly one MST. Thus, we can refer to *the* (rather than *an*) MST of G. Even if the edge weights are not distinct, we can pretend that they are by using a consistent tiebreaking function.

Instead of explicitly returning a tree T in adjacency list format, for convenience, we sometimes simply return a list F of edges. In a similar vein, we sometimes say things like "F is a subset of T" even if neither F nor T is technically a set. Converting F to T is a relatively straightforward task that can be completed in linear time.

We will describe three algorithms that all run in $O(m^2)$ time (or faster): Prim's algorithm, Kruskal's algorithm, and Reverse-Delete. But first, we prove a property about MSTs known as the Cut Property. A *cut* is a subset of vertices, and an edge *crosses* a cut if has exactly one endpoint in the cut.

Cut Property

For any cut S in G, the lightest edge crossing S is in the MST of G.

Proof: For contradiction, assume there exists a cut S such that the lightest edge e crossing S is not in the MST T of G. Notice that $T \cup \{e\}$ contains 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, $T' = T + e - f$ is a spanning tree and T' is lighter than T; this contradicts our assumption that T is the MST of G.

Using the Cut Property, we can prove the correctness of Prim's and Kruskal's algorithms.

Prim's Algorithm

Algorithm: Let $S = \{1\}$ (recall that $V = [n]$). Repeat the following $n-1$ times: find the lightest edge e crossing S, add the endpoint of e not in S to S, and add e to F. Return F.

```
Prim(G):
1 S = {1}; F = [empty list]
2 for i = 1, ..., n − 1:
3     e = lightest edge crossing S
4     v = endpoint of e not in S
5     add v to S; add e to F
6 return F
```

Although S is a set, implementations of sets typically require hashing. We can avoid the complexities of hashing by implementing S as a binary array, where $S[u] = 1$ if and only if $u \in S$. In other words, $S = \{1\}$ is equivalent to "$S = [0] * n$; $S[1] = 1$" and "add v to S" is equivalent to "$S[v] = 1$".

Correctness: In each iteration of Prim's algorithm, we add the lightest edge crossing S to F. By the Cut Property, this edge is in the MST T, so F is always a subset of T. Since the algorithm terminates when F has $n-1$ edges, and any spanning tree has exactly $n-1$ edges, it returns the MST of G. (As noted earlier, converting F to adjacency list format is relatively straightforward, linear-time task.)

Running time: The algorithm makes $n-1$ iterations, and each iteration involves finding the lightest edge, which takes $O(m)$ time. Thus, the total running time is $O(mn) = O(m^2)$.

Kruskal's Algorithm

Algorithm: Sort E by increasing weight. For each edge e in this order, add e to a list F (initially empty) if $F + e$ is acyclic. Return F.

Correctness: Consider any edge $e = \{u, v\}$ added to F, and let S denote the set of vertices connected to u via F immediately before e was added to F. At this point, e is the lightest edge that could be added to F without creating a cycle, so e is the lightest edge that crosses S. So by the Cut Property, F is always a subset of the MST.

```
Kruskal(G):
sort $E$ by increasing weight
$F$ = [empty list]
for $e$ in $E$:
    if $F + e$ is acyclic:
        add $e$ to $F$
return $F$
```

To show that F is spanning, we assume there exists a cut S' such that no edge crossing S' is in F. This contradicts the behavior of the algorithm, since $F + e$ is acyclic for any edge e that crosses S'.

Running time: Sorting E takes $O(m \log m)$ time. The algorithm then makes m iterations, and in each iteration, we need to check if $F + e$ is acyclic. We can do this in $O(n)$ time as follows: run BFS to check if the endpoints of e are connected in the graph (V, F). Normally, BFS takes $O(m + n)$ time, but (V, F) has at most $n - 1$ edges, so it only takes $O(n)$ time in this case. Thus, the total running time is $O(m \log m) + m \cdot O(n) = O(mn) = O(m^2)$.

Remark

There is often a tradeoff between simplicity and speed among the different implementations of the same algorithm. For example, Prim's algorithm can be implemented in $O(m \log n)$ time using a heap-based priority queue, and Kruskal's algorithm can be implemented in $O(m \log n)$ time using an array-based Union-Find data structure.

Reverse-Delete

We now introduce another property about MSTs and use it to prove the correctness of the Reverse-Delete algorithm.

Cycle Property

For any cycle C in G, the heaviest edge in C is not in the MST of G.

Proof: This proof, like that of the Cut Property (and many correctness proofs of greedy algorithms), uses an exchange argument. For contradiction, assume there exists a cycle C whose heaviest edge f is in the MST T of G. Notice that removing f from T creates a cut S. Some other edge e in C must cross S, and since f is the heaviest edge in C, we have $w(e) < w(f)$. Thus, $T' = T + e - f$ is a spanning tree and T' is lighter than T; this contradicts our assumption that T is the MST.

Algorithm: (This algorithm is a backward version of Kruskal's algorithm.) Sort E by decreasing weight. For each edge e in this order, remove e from G if $G - e$ is connected. Return G.

Correctness: When the algorithm removes an edge e from G, G remains connected, so e must be the heaviest edge in some cycle in G. Thus, by the cycle property, e is not in the MST, which means that the MST T is contained in G throughout the algorithm. We want to show that at the end of the algorithm, G is acyclic.

```
Reverse-Delete($G$):
1 sort $E$ by decreasing weight
2 for $e$ in $E$:
3     if $G - e$ is connected:
4         remove $e$ from $G$
5 return $G$
```

For contradiction, suppose G contains a cycle C at the end of the algorithm, and let f denote the heaviest edge in C. This contradicts the behavior of the algorithm, since f could have been removed from G without disconnecting it.

Running time: (This analysis is also similar to that of Kruskal's algorithm.) Sorting E takes $O(m \log m)$ time. We can check if $G - e$ is connected by running BFS, which takes $O(m+n) = O(m)$ time. Removing e from G takes $O(n) = O(m)$ time, so each of the m iterations takes $O(m)$ time. Thus, the total running time is $O(m \log m) + m \cdot O(m) = O(m^2)$.

3.2 Selecting Compatible Intervals

An *interval* is an array of two positive integers $[s, t]$ such that $s < t$. We think of an interval as an event, where s is the "start time" and t is the "end time." In the following problem, we're given an array of events, and we want to attend as many as possible.

Problem Statement

The input is an array A of n intervals. A list of intervals is *compatible* if no two intervals in the list conflict at any point in time. Our goal is to return a list of compatible intervals that contains as many intervals as possible. Describe an $O(n \log n)$-time algorithm for this problem, prove that it's correct, and analyze its running time.

Here is a generic greedy algorithm for this problem: "Among all intervals compatible with S, keeping adding the interval e according to some criterion C." And here are four possible choices for the criterion C: e starts the earliest, e is the shortest, e overlaps with the fewest number of remaining intervals, e ends the earliest. All four criteria might sound plausible, but only the fourth leads to an optimal algorithm.

Remark

Sometimes, it is relatively easy to find a counterexample showing that an algorithm is not optimal. Other times, it is quite challenging! In those cases, we can alternate between finding a counterexample and proving that the algorithm is actually optimal.

Algorithm: Sort A by non-decreasing end time and create an empty list S. For each interval e in this order, if e does not conflict with the last interval in S, add e to the end of S. (Notice that if e conflicts with any interval in S, e must conflict with the last interval in S.) Return S.

Correctness: Let ALG and OPT denote the the algorithm's solution and the optimal solution, respectively, each sorted by non-decreasing end time; also, let $m = |\mathsf{ALG}|$. If $\mathsf{ALG}[i] = \mathsf{OPT}[i]$ for all $i \in [m]$, then the algorithm is optimal. Otherwise, let $i \in [m]$ denote the smallest index such that $\mathsf{ALG}[i] \neq \mathsf{OPT}[i]$. We will construct another optimal solution OPT' such that $\mathsf{ALG}[j] = \mathsf{OPT}[j]$ for all $j \in [i]$. Notice that this suffices because, by repeating the exchange procedure, we can eventually arrive at an optimal solution that agrees with ALG for all $i \in [m]$.

The key is that $\mathsf{OPT}[i]$ cannot end before $\mathsf{ALG}[i]$ because the algorithm is greedy: the algorithm chose $\mathsf{ALG}[i]$ because it was compatible with the $i-1$ previously chosen intervals and it had the smallest end time. So we construct OPT' as follows: start with OPT, remove $\mathsf{OPT}[i]$, and add $\mathsf{ALG}[i]$. Notice that OPT' is feasible, $|\mathsf{OPT}'| = |\mathsf{OPT}|$, and $\mathsf{ALG}[j] = \mathsf{OPT}[j]$ for all $j \in [i]$. Thus, as mentioned above, we can repeat this exchange argument until we arrive at an optimal solution that agrees with ALG for all $i \in [m]$.

```
Select-Intervals($A$):
1 sort $A$ by non-decreasing end time
2 $S$ = [empty list]
3 for $e$ in $A$:
4     if $e$ does not conflict with the last interval in $S$:
5         add $e$ to the end of $S$
6 return $S$
```

Running time: Sorting A takes $O(n \log n)$ time, and each of the n iterations takes $O(1)$ time, so the total running time is $O(n \log n)$.

3.3 Fractional Knapsack

Suppose we're selecting items to pack in our knapsack (i.e., backpack). Each item i has a value $v[i]$ (e.g., 5 dollars) and weight $w[i]$ (2 pounds), and our knapsack has a capacity B (8 pounds). We want to maximize the total value in our knapsack, but the total weight must be at most B. In the Fractional Knapsack problem, we're allowed to split items (e.g., if we take 20% of item 3, our total value increases by $v[3]/5$ and our total weight increases by $w[3]/5$). (In Section 4.4, we'll solve the 0/1 Knapsack problem, in which we cannot split any item.)

Problem Statement

The input is (v, w, B), where v and w are arrays of n positive integers each and $B \in \mathbb{Z}^+$. Our goal is to return an array x of n real numbers such that $0 \le x[i] \le 1$ for all $i \in [n]$, $\sum_{i=1}^{n} x[i] \cdot w[i] \le B$, and $\sum_{i=1}^{n} x[i] \cdot v[i]$ is maximized.

```
Fractional-Knapsack($v, w, B$):
1 sort items by non-increasing $v[i]/w[i]$
2 $x = [0] * n$
3 for each item $i$:
4     increase $x[i]$ by as much as possible
5 return $x$
```

Algorithm: Sort the items by non-increasing $v[i]/w[i]$. In this order, pick as much of each item as possible without the total weight exceeding B. (Notice that the algorithm returns at most one $x[i] \notin \{0, 1\}$.)

Correctness: Let ALG denote the algorithm's solution, and let OPT denote an optimal solution. For any solution y, let $v(y)$ denote the value of y. For contradiction, assume $v(\mathsf{OPT}) > v(\mathsf{ALG})$; let i denote the smallest index such that $\mathsf{OPT}[i] \neq \mathsf{ALG}[i]$. By design of the algorithm, we must have $\mathsf{OPT}[i] < \mathsf{ALG}[i]$. Furthermore, since $v(\mathsf{OPT}) > v(\mathsf{ALG})$, there must exist some $j > i$ such that $\mathsf{OPT}[j] > 0$.

We will construct another optimal solution OPT'. More specifically, we obtain OPT' by modifying OPT as follows: increase $\mathsf{OPT}[i]$ by some $\epsilon > 0$ and decrease $\mathsf{OPT}[j]$ by $\epsilon w[i]/w[j]$. Note that if ϵ is small enough, then OPT' is feasible; in fact, it has the same weight as OPT. Now consider the value of OPT':

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

where the inequality follows from $\epsilon > 0$ and $v_i/w_i \geq v_j/w_j$. Thus, OPT' is also an optimal solution but it takes more of item i than OPT does. By repeating this process, we can arrive at an optimal solution identical to ALG.

Remark

For every greedy algorithm ALG we've seen, the correctness proof uses an exchange argument: assume that OPT forgoes a greedy choice, construct another optimal solution OPT' that makes the greedy choice instead, and show that OPT' is either "more optimal" than OPT or "more similar" to ALG. Either way, we can conclude that ALG is optimal.

Running time: Sorting takes $O(n \log n)$ time, and each of the n iterations takes $O(1)$ time, so the total running time is $O(n \log n)$.

4 Dynamic Programming

Dynamic programming can be summarized as "recursion with a table." The idea is to solve a sequence of increasingly larger subproblems by using solutions to smaller subproblems.

Overview

Section	Summary	Time
4.1: Max in Array	$\mathsf{OPT}[i] = \max(A[1:i]) = \max(\mathsf{OPT}[i-1], A[i])$.	$O(n)$
4.2: LIS	$\mathsf{OPT}[i] =$ length of LIS in $A[1:i]$ ending at $A[i]$ $= 1 + \max_{j<i, A[j]<A[i]} \mathsf{OPT}[j]$.	$O(n^2)$
4.3: LPS	$\mathsf{OPT}[i][j] =$ length of LPS in $A[i:j] = 2 + \mathsf{OPT}[i+1][j-1]$ if $A[i] = A[j]$, else $\max(\mathsf{OPT}[i+1][j], \mathsf{OPT}[i][j-1])$	$\uparrow$
4.4: 0/1 Knapsack	$\mathsf{OPT}[i][j] =$ maximum value with items $[i]$ available and capacity $j = \max(\mathsf{OPT}[i-1][j], v[i] + \mathsf{OPT}[i-1][j-w[i]])$.	$O(nB)$
4.5: Edit Distance	$\mathsf{OPT}[i][j] = \text{Edit-Distance}(A[1:i], B[1:j]) = \min(\mathsf{OPT}[i][j-1]+1, \mathsf{OPT}[i-1][j]+1, \mathsf{OPT}[i-1][j-1]+\delta_{ij})$.	$O(mn)$
4.6: MIS in Trees	$\mathsf{OPT}_{\mathsf{in}}[u]/\mathsf{OPT}_{\mathsf{out}}[u] =$ value of MIS in T_u that includes/excludes u, $\mathsf{OPT}_{\mathsf{in}}[u] = w(u) + \sum_{v \in T[u]} \mathsf{OPT}_{\mathsf{out}}[v]$, $\mathsf{OPT}_{\mathsf{out}}[u] = \sum_{v \in T[u]} \max(\mathsf{OPT}_{\mathsf{in}}[v], \mathsf{OPT}_{\mathsf{out}}[v])$.	$O(n)$

When presenting a dynamic programming (DP) algorithm, the format described below is clearer than the usual (i.e., "algorithm, correctness, running time").

Format for DP Algorithms

1. **Subproblems:** Which subproblems will we solve? What will we return?
2. **Recurrence:** How can we solve each subproblem using solutions to smaller subproblems? What are the base cases? Why does the recurrence hold?
3. **Algorithm:** How do we use the recurrence to populate a DP table (i.e., array d)?
4. **Running time:** What is the running time of the algorithm? (This is typically (number of problems) $\times$ (time per subproblem).)

When explaining the logic behind an algorithm, we generally refer to "the" optimal solution of a problem even though multiple optimal solutions could exist. But all of them are equally valid, so it's fine for us to focus on just one of them. (This applies throughout these notes.)

4.1 Max in Array

Let's use dynamic programming to solve a familiar problem.

Problem Statement

The input is an array A of n distinct, positive integers. Our goal is to return the largest integer in A. Describe an $O(n)$-time algorithm for this problem, prove that it's correct, and analyze its running time.

Subproblems: For all $i \in [n]$, let $\mathsf{OPT}[i] = \max(A[1:i])$; we will return $\mathsf{OPT}[n]$.

Recurrence: The base case is $\mathsf{OPT}[1] = A[1]$. For all $i \geq 2$, $\mathsf{OPT}[i] = \max(\mathsf{OPT}[i-1], A[i])$. The recurrence holds because $\max(A[1:i])$ is either the largest integer in $A[1:i-1]$ (i.e., $\mathsf{OPT}[i-1]$), or it is $A[i]$; $\mathsf{OPT}[i]$ "picks" the larger option.

```
Max-in-Array-DP($A$):
1 $d = [A[1]] * n$
2 for $i = 2, \ldots, n$:
3     $d[i] = \max(d[i-1], A[i])$
4 return $d[n]$
```

Algorithm: Initialize $d = [A[1]] * n$. For $i \geq 2$, set $d[i]$ according to the recurrence for $\mathsf{OPT}[i]$ above. That is, set $d[i] = \max(d[i-1], A[i])$. At the end, return $d[n]$.

Running time: The algorithm makes $n-1$ iterations, and each iteration takes $O(1)$ time, so the total running time is $O(n)$.

Remark

When describing a DP algorithm, we use "d" instead of "OPT" to indicate a distinction between the reasoning behind the algorithm itself (i.e., arrays, loops, etc.) and the reasoning behind the algorithm (i.e., subproblems, recurrence, etc.).

4.2 Longest Increasing Subsequence

A *subsequence* of an array A is a subarray of A, but with some elements possibly skipped. For example, some subsequences of $A = [3, 5, 5, 1, 3, 2]$ are $[3, 5, 3]$, $[5]$, and $[5, 2]$. Note that $[1, 5]$ and $[1, 1]$ are not subsequences of A.

Problem Statement

The input is an array A of n integers. Our goal is to return the length of the longest increasing subsequence (LIS) of A. Describe an $O(n^2)$-time algorithm for this problem, prove that it's correct, and analyze its running time.

Subproblems: For all $i \in [n]$, let $\mathsf{OPT}[i]$ denote the length of the LIS of A that must end on $A[i]$. Since the LIS of A must end somewhere, we will return $\max_i \mathsf{OPT}[i]$.

Remark

It might seem more natural to let $\mathsf{OPT}[i]$ denote the length of the LIS of $A[1:i]$. This is a reasonable idea, but it's not clear how we can solve these subproblems by following a recurrence. In general, in any DP algorithm, it is not necessarily the case that each subproblem is strictly a smaller version of the original problem.

Recurrence: The base case is $\mathsf{OPT}[1] = 1$. Intuitively, we calculate $\mathsf{OPT}[i]$ for all $i \geq 2$ by finding the "best" entry $A[j]$ to "come from" before "jumping" to $A[i]$. Note that $A[j]$ is an option if and only if $j < i$ and $A[j] < A[i]$. More formally, for any $i \geq 2$, $\mathsf{OPT}[i]$ satisfies the following recurrence:

$$\mathsf{OPT}[i] = 1 + \max_{j \in C} \mathsf{OPT}[j],$$

where $C = \{j \mid j < i \text{ and } A[j] < A[i]\}$ denotes the set of "candidates" we consider when calculating $\mathsf{OPT}[i]$. (If $C = \emptyset$, $\mathsf{OPT}[i] = 1$.)

```
LIS(A):
d = [1] * n
for i = 2, ..., n:
    for j = 1, ..., i − 1:
        if A[j] < A[i] and d[i] < 1 + d[j]:
            d[i] = 1 + d[j]
return max(d)
```

In other words, the LIS S that ends on $A[i]$ must be of the form $S = T + A[i]$, where T is an LIS that ends on $A[j]$ for some $j \in C$. By finding the maximum $\mathsf{OPT}[j]$ over all all $j \in C$, we correctly calculate $\mathsf{OPT}[i]$.

Algorithm: Calculate $d[i]$ according to the recurrence for $\mathsf{OPT}[i]$ above and return $\max(d)$.

Running time: We calculate each $d[i]$ in $O(i)$ time, so the total running time is $O(n^2)$.

Remark

To find the LIS itself (as opposed to just its length), we can maintain an array of parent pointers as we did in BFS (Section 2.1). More specifically, whenever we update $d[i] = 1 + d[j]$, we also set $p[i] = j$ to indicate that $\mathsf{OPT}[i]$ is obtained from appending $A[j]$ to $\mathsf{OPT}[j]$. Given d and p, we can trace p to recover the LIS itself in $O(n)$ time.

4.3 Longest Palindromic Subsequence

A subsequence S is *palindromic* if S is equal to the reverse of itself. For example, if $A = \mathsf{abracadabra}$, then aca and abba are palindromic subsequences of A.

Problem Statement

The input is an string A of length n. Our goal is to return the length of a longest palindromic subsequence (LPS) of A. Describe an $O(n^2)$-time algorithm for this problem, prove that it's correct, and analyze its running time.

Subproblems: For all $i \in [n-1]$ and $j \in [i, n]$, let $\mathsf{OPT}[i][j]$ denote the length of an LPS in $A[i:j]$. We will return $\mathsf{OPT}[1][n]$. We can visualize the DP table as an $n \times n$ square, but we are only concerned with the values in the upper-right triangle.

Recurrence: If $i = j$, then $A[i:j]$ is just a single character, so $\mathsf{OPT}[i][i] = 1$. If $i < j$ and $A[i] = A[j]$, then the LPS of $A[i:j]$ starts with $A[i]$, ends with $A[j]$, and contains an LPS of $A[i+1:j-1]$ in between. Otherwise, if $A[i] \neq A[j]$, then the LPS of $A[i:j]$ must exclude $A[i]$ or $A[j]$. In summary,

$$\mathsf{OPT}[i][j] = \begin{cases} \mathsf{OPT}[i+1][j-1] + 2 & \text{if } A[i] = A[j] \\ \max(\mathsf{OPT}[i+1][j], \mathsf{OPT}[i][j-1]) & \text{otherwise.} \end{cases}$$

```
LPS($A$):
d = [0] * (n × n)
for $i = n, \dots, 1$:
    $d[i][i] = 1$                    // base case
    for $j = i+1, \dots, n$:
        if $A[i] = A[j]$:
            $d[i][j] = d[i+1][j-1] + 2$
        else:
            $d[i][j] = \max(d[i+1][j], d[i][j-1])$
return $d[1][n]$
```

Algorithm: Each $\mathsf{OPT}[i][j]$ depends on entries below and to the left of it, so one way to fill the DP table is row by row, starting with the last row. In each row, we set $d[i][j]$ according to the recurrence above. At the end, we return $d[1][n]$.

Running time: The DP table has $O(n^2)$ entries, and each entry takes $O(1)$ time to compute, so the total running time is $O(n^2)$.

4.4 0/1 Knapsack

Now we'll consider a problem with multiple parts and see how we can shrink them simultaneously. Recall that in Knapsack problem (Section 3.3), v is an array of item values, w is an array of item weights, and B is the capacity of the knapsack.

Problem Statement

The input is (v, w, B), where v and w are arrays of n positive integers each and $B \in \mathbb{Z}^+$. For any $S \subseteq [n]$, S is feasible if $\sum_{i \in S} w[i] \leq B$, and the value of S is $\sum_{i \in S} v[i]$. Our goal is to return the maximum value over all feasible solutions. Describe an $O(nB)$-time algorithm for this problem, prove that it's correct, and analyze its running time.

Subproblems: For all $i \in [n]$ and $j \in \{0, 1, \dots, B\}$, let $\mathsf{OPT}[i][j]$ denote the optimal value if we could only select items from $[i]$ and the knapsack only had capacity j. We will return $\mathsf{OPT}[n][B]$.

Recurrence: The base cases are $\mathsf{OPT}[1][j] = 0$ for all $j < w[1]$ and $\mathsf{OPT}[1][j] = v[1]$ for all $j \geq w[1]$. For $i \geq 2$, let $S_{ij} \subseteq [i]$ denote the optimal solution for $\mathsf{OPT}[i][j]$. Notice that S_{ij} either contains item i or it doesn't. If $w[i] > j$, then S_{ij} cannot contain i, so $\mathsf{OPT}[i][j] = \mathsf{OPT}[i-1][j]$. Otherwise, S_{ij} either ignores i, or includes it to obtain value $v[i]$ and fills the rest of its knapsack with value $\mathsf{OPT}[i-1][j-w[i]]$. In summary,

$$\mathsf{OPT}[i][j] = \begin{cases} \mathsf{OPT}[i-1][j] & \text{if } w[i] > j \\ \max(\mathsf{OPT}[i-1][j], v[i] + \mathsf{OPT}[i-1][j-w[i]]) & \text{otherwise.} \end{cases}$$

```
Knapsack-DP(v, w, B):
d = [0] * (n × (B + 1))
for j = 1, ..., B:
    if j ≥ w[1]:
        d[1][j] = v[1]                                  // base case
for i = 2, ..., n:
    for j = 1, ..., B:
        if j < w[i]:
            d[i][j] = d[i − 1][j]          // we must ignore item i
        else:
            // we can ignore or include item i
            d[i][j] = max(d[i − 1][j], v[i] + d[i − 1][j − w[i]])
return max(d)
```

Algorithm: We populate an array d of size $n \times (B+1)$ according to the recurrence above. (The row indices are $\{1, \ldots, n\}$, and the column indices are $\{0, 1, \ldots, B\}$.) Again, we can populate d in multiple ways; the code above goes row by row. Any order would be correct, as long as we calculate $\mathsf{OPT}[i][j]$ after we've calculated $\mathsf{OPT}[i-1][j]$ and $\mathsf{OPT}[i-1][j-w[i]]$.

Running time: The DP table has $n(B+1)$ entries, and each entry takes $O(1)$ time to compute, so the total running time is $O(nB)$.

4.5 Edit Distance

If A and B are strings, the *edit distance* from A to B is a non-negative integer that represents the "distance" from A to B. More specifically, it is the minimum number of "moves" we need to make to turn A into B. There are three valid types of moves: insert a character (anywhere in A), delete a character (anywhere from A), and replace one character with another. (A replacement is equivalent to a deletion followed by an insertion, but it counts as one move rather than two.)

Problem Statement

The input is (A, B), where A and B are strings of length m and n, respectively. Our goal is to return the edit distance from A to B. Describe an $O(mn)$-time algorithm for this problem, prove that it's correct, and analyze its running time.

Subproblems: For all $i \in \{0, 1, \ldots, m\}$ and $j \in \{0, 1, \ldots, n\}$, let $\mathsf{OPT}[i][j]$ denote the edit distance from $A[1:i]$ to $B[1:j]$. We will return $\mathsf{OPT}[m][n]$.

Recurrence: The base cases are $\mathsf{OPT}[0][j] = j$ and $\mathsf{OPT}[i][0] = i$. (Editing the empty string to a string of length j takes j insertions, and editing a string of length i into the empty string takes i deletions.) Now suppose $i \geq 1$ and $j \geq 1$. Since $\mathsf{OPT}[i][j]$ edits $A[1:i]$ such that its last character is $B[j]$, it must make one of the following three moves:

1. Edit $A[1:i]$ into $B[1:j-1]$ and insert $B[j]$.
2. Edit $A[1:i-1]$ into $B[1:j]$ and delete $A[i]$.
3. Edit $A[1:i-1]$ into $B[1:j-1]$ and replace $A[i]$ with $B[j]$. Note that this "replacement" costs 0 if $A[i] = B[j]$ and 1 otherwise.

This implies that $\mathsf{OPT}[i][j]$ satisfies the following recurrence:

$$\mathsf{OPT}[i][j] = \min \begin{cases} \mathsf{OPT}[i][j-1] + 1 \\ \mathsf{OPT}[i-1][j] + 1 \\ \mathsf{OPT}[i-1][j-1] + \delta_{ij} \end{cases}$$

where $\delta_{ij} = 0$ if $A[i] = B[j]$ and 1 otherwise.

```
Edit-Distance(A, B):
d = [0] * ((m+1) × (n+1))
for j = 1, ..., n:
    d[0][j] = j                                  // fill row 0
for i = 1, ..., m:
    d[i][0] = i                                  // fill column 0
for i = 1, ..., m:
    for j = 1, ..., n:
        d[i][j] = min(d[i][j-1] + 1, d[i-1][j] + 1)
        if A[i] = B[j]:
            d[i][j] = min(d[i][j], d[i-1][j-1])
        else:
            d[i][j] = min(d[i][j], d[i-1][j-1] + 1)
return d[m][n]
```

Algorithm: As usual, we populate an array d according to the recurrence above. (The row indices are $\{0, 1, \ldots, m\}$ and the column indices are $\{0, 1, \ldots, n\}$.) There are multiple ways to populate d; again, we go one row at a time, and within each row, we go from left to right.

Remark

It can be proven that if $A[i] = B[j]$, then $\mathsf{OPT}[i][j] = \mathsf{OPT}[i-1][j-1]$. (This is similar to the $A[i] = A[j]$ case in LPS.) Using this fact alone, we cannot improve the asymptotic running time of the algorithm, but it does speed up the process of running the algorithm by hand.

Running time: The DP table has $(m+1)(n+1) = O(mn)$ entries, and each entry takes $O(1)$ time to compute, so the total running time is $O(mn)$.

4.6 Independent Set in Trees

As we've seen, dynamic programming works by turning the original problem into subproblems. If the input is an array A, each subproblem often corresponds to a prefix of A. In this section, we'll apply the same idea to trees: if the input is a tree T, each subproblem corresponds to a subtree of T. We will simultaneously see an additional technique: for each subtree of T, we will define *two* subproblems tied together by their recurrences.

For any undirected graph G, a subset S of vertices is an *independent set* if no edge in G has both endpoints in S. If every vertex u has a weight $w(u)$, the weight of S is $\sum_{u \in S} w(u)$. (If weights are not given, we assume every vertex has weight 1.) A maximum independent set (MIS) is an independent set whose weight is as large as possible.

Problem Statement

The input is (T, w), where $T = (V, E)$ is a tree rooted at vertex 1 and $w[u] \in \mathbb{Z}^+$ is the weight of vertex $u \in V$. (For every vertex u, $T[u]$ is a list of u's children.) Our goal is to return the weight of a maximum independent set in T. Describe an $O(n)$-time algorithm for this problem, prove that it's correct, and analyze its running time.

Subproblems: For all $u \in V$, let T_u denote the subtree of T rooted at u. We define two subproblems per T_u: let $\mathsf{OPT}_{\mathsf{in}}[u]$ denote the weight of the MIS in T_u that includes u, and let $\mathsf{OPT}_{\mathsf{out}}[u]$ denote the weight of the MIS in T_u that excludes u. Since the original tree is T_1 and the MIS in T_1 either includes or excludes vertex 1, we will return $\max(\mathsf{OPT}_{\mathsf{in}}[1], \mathsf{OPT}_{\mathsf{out}}[1])$.

Remark

The definition of $\mathsf{OPT}_{\mathsf{in}}[u]$ is similar to that of $\mathsf{OPT}[i]$ for the LIS problem (Section 4.2): in both subproblems, we are required to include the index of the subproblem as part of the solution. In contrast, for 0/1 Knapsack, it is not required that the solution for $\mathsf{OPT}[i][j]$ must include item i. However, the logic below is similar to the logic behind 0/1 Knapsack.

Recurrence: If u is a leaf, then T_u is just u, so $\mathsf{OPT}_{\mathsf{in}}[u] = w(u)$ and $\mathsf{OPT}_{\mathsf{out}}[u] = 0$. Now suppose u is not a leaf. The solution S to $\mathsf{OPT}_{\mathsf{in}}[u]$ includes u, so for each $v \in T[u]$, S contains an MIS in T_v that excludes v. This implies

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

On the other hand, the solution S' to $\mathsf{OPT}_{\mathsf{out}}[u]$ excludes u, which means for each $v \in T[u]$, S' is free to either include or exclude v. This implies

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

Algorithm: We populate two arrays, d_{in} and d_{out}, according to the recurrences above. For any $u \in V$, when computing $d_{\mathsf{in}}[u]$ and $d_{\mathsf{out}}[u]$, we need to make sure that $d_{\mathsf{out}}[v]$ and $d_{\mathsf{out}}[v]$ have both already been computed for every child $v \in T[u]$. Thus, if we visualize the root as the top of the tree, we need to work our way up the tree.

```
MIS-Tree($T, w$):
1 $d_{\mathsf{in}}, d_{\mathsf{out}} = [0] * n, [0] * n$
2 sort $V$ in reverse topological order
3 for $u \in V$:
4     $d_{\mathsf{in}}[u] = w[u]$
5     for $v \in T[u]$:
6         $d_{\mathsf{in}}[u] \mathrel{+}= d_{\mathsf{out}}[v]$
7         $d_{\mathsf{out}}[u] \mathrel{+}= \max(d_{\mathsf{in}}[v], d_{\mathsf{out}}[v])$
8 return $\max(d_{\mathsf{in}}[1], d_{\mathsf{out}}[1])$
```

One way to do this is the following: if $v \in T[u]$, treat (u, v) as a directed edge, process V in *reverse* topological ordering (Section 2.4). This ensures that when we're computing $d_{\mathsf{in}}[u]$ and $d_{\mathsf{out}}[u]$, we've already computed $d_{\mathsf{in}}[v]$ and $d_{\mathsf{out}}[v]$ for every $v \in T[u]$, as desired. At the end, return $\max(d_{\mathsf{in}}[1], d_{\mathsf{out}}[1])$.

Running time: Computing a topological order takes $O(m + n)$ time. Since T is a tree, $m = n-1$, so $O(m+n) = O(n)$. For each vertex u, computing $d_{\mathsf{in}}[u]$ and $d_{\mathsf{out}}[u]$ takes $O(|T[u]|)$ time. Since $\sum_u |T[u]| = m = O(n)$, the total running time is $O(n) + O(n) = O(n)$.

Summary

We conclude this chapter with a list of common DP patterns. In the next chapter, we'll see more dynamic programming!

Common DP Patterns

If the input is...

- A, an array of length n:

1. $\forall i \in [n]$: $\mathsf{OPT}[i] = \mathsf{OPT}$ given $A[1:i]$
2. $\forall i \in [n]$: $\mathsf{OPT}[i] = \mathsf{OPT}$ given $A[i:n]$
3. $\forall i \in [n]$: $\mathsf{OPT}[i] = \mathsf{OPT}$ given $A[1:i]$ that somehow involves $A[i]$
4. $\forall i \in [n] \forall j \in [i, n]$: $\mathsf{OPT}[i][j] = \mathsf{OPT}$ given $A[i:j]$

- (A, k), where A is an array of length n and k is a non-negative integer:

5. $\forall i \in [n] \forall j \in \{0, 1, \ldots, k\}$: $\mathsf{OPT}[i][j] = \mathsf{OPT}$ given $(A[1:i], j)$

- (A, B), where A and B are arrays of length m and n:

6. $\forall i \in [m] \forall j \in [n]$: $\mathsf{OPT}[i][j] = \mathsf{OPT}$ given $(A[1:i], B[1:j])$

- T, a rooted tree with vertex set V:

7. $\forall u \in V$: $\mathsf{OPT}[u] = \mathsf{OPT}$ given T_u
8. $\forall u \in V$: $\mathsf{OPT}[u] = \mathsf{OPT}$ given T_u that somehow involves u

5 Shortest Paths

Many algorithms for the shortest path problem use dynamic programming.

Overview

Section	Summary	Time
5.1: DAG DP	$\mathsf{OPT}[v]$ = distance from s to v $= \min_{u:(u,v)\in E} \mathsf{OPT}[u] + \ell(u,v)$.	$O(m+n)$
5.2: Bellman-Ford	$\mathsf{OPT}[v][j]$ = length of shortest walk from s to v containing at most j edges $= \min\{\mathsf{OPT}[v][j-1], \min_u \mathsf{OPT}[u][i-1] + \ell(u,v)\}$	$O(mn)$
5.3: Dijkstra's Algorithm	BFS but in each iteration, process the vertex with the smallest d-value (i.e., queue $\to$ priority queue).	$O(n^2)$
5.4: Floyd-Warshall	$\mathsf{OPT}[u][v][r]$ = distance from u to v with $[r]$ available as intermediate vertices $= \min\{\mathsf{OPT}[u][v][r-1], \mathsf{OPT}[u][r][r-1] + \mathsf{OPT}[r][v][r-1]\}$.	$O(n^3)$

Sections 5.1 through 5.3 consider variants of the Single-Source Shortest Path (SSSP) problem described below. Section 5.4 considers the All-Pairs Shortest Path (APSP) problem.

Problem Statement

SSSP: The input is (G, s), where G is a directed graph with edge lengths ℓ, and a "source" vertex $s \in V$. Our goal is to return the shortest paths from s to all $v \in V$.

The algorithms in this section only return an array d such that $d[v]$ is the distance from s to v. But as usual, we can maintain parent pointers and trace them at the end to recover the shortest paths themselves.

5.1 DAG DP

Problem Statement

SSSP, where G is a DAG. Describe an $O(m+n)$-time algorithm for this problem, prove that it's correct, and analyze its running time.

Subproblems: For all $v \in V$, let $\mathsf{OPT}[v]$ denote the distance from s to v. We will return $d = \mathsf{OPT}$.

Recurrence: The base case is $\mathsf{OPT}[s] = 0$. For all $v \neq s$, we have

$$\mathsf{OPT}[v] = \min_{u:(u,v)\in E} \mathsf{OPT}[u] + \ell(u,v)$$

since the shortest path P from s to v must arrive at some in-neighbor u of v and traverse the edge $\ell(u, v)$. The subpath of P from s to u must be a shortest path from s to u.

Algorithm: Before computing $d[v]$, we need to compute $d[u]$ for every in-neighbor u of v, so we process V in topological order. (This is like Section 4.6 but flipped.) We also need to iterate through the in-neighbors of v, but $G[u]$ only contains the out-neighbors of u. Computing the in-neighbors of v could take $\Omega(m)$ time, which is too slow. So instead, we pre-compute the in-neighbors of every vertex by creating a separate G', which is G but with every edge reversed, so in-neighbors in G are out-neighbors in G'.

```
DAG-DP(G, s):
1 $d = [\infty] * n$  $d[s] = 0$
2 sort $V$ in topological order
3 $G' = G$ with each edge reversed
4 for $v \in V$:
5     for $u \in G'[v]$:
6         $d[v] = \min(d[v], d[u] + \ell(u, v))$
7 return $d$
```

Running time: We can sort V in topological order using DFS (Section 2.2) in $O(m + n)$, and we can also compute G' in $O(m + n)$ time. Computing $d[v]$ takes $O(\text{in-deg}(v))$ time, and the sum of in-degrees is m. Thus, the total running time is $O(m + n)$.

Edge relaxations

In DAG-DP, we update $d[v]$ by setting $d[v] = \min(d[v], d[u] + \ell(u, v))$; this operation is known as "relaxing" the edge (u, v). Intuitively, when we relax (u, v), we're checking if we can decrease our current estimate of the distance from s to v by following an s-u path plus the edge (u, v). As we'll see, edge relaxations also play an important role in other shortest path algorithms.

5.2 Bellman-Ford

A cycle C is a *negative cycle* if $\sum_{e \in C} \ell(e) < 0$. If G has a negative cycle C, it's unclear what we mean by "shortest path" because if a path P can reach C, we can keep making P "shorter" by cycling around C. One idea is to forbid paths from repeating vertices, but then the SSSP problem becomes much more difficult.[1] So let's assume that G has no negative cycles.

Problem Statement

SSSP, where G has no negative cycles. Describe an $O(mn)$-time algorithm for this problem, prove that it's correct, and analyze its running time.

Subproblems: For all $v \in V$ and $j \in \{0, 1, \ldots, n-1\}$, let $\mathsf{OPT}[v][j]$ denote the length of the shortest s-v walk that contains at most j edges. Since G has no negative cycles, every shortest walk has at most $n - 1$ edges, so we will return $\mathsf{OPT}[\cdot][n-1]$ (i.e., the last column of the DP table).

[1]More specifically, the problem becomes NP-hard, which means it appears to be impossible to solve in polynomial time (see Chapter 7). Also, a path is not allowed to repeat vertices by definition, but in this context, it's common to use "path" when we really mean "walk."

Recurrence: The base cases are $\mathsf{OPT}[s][0] = 0$ and $\mathsf{OPT}[v][0] = \infty$ for all $v \in V \setminus \{s\}$. If $j \geq 1$, then the path for $\mathsf{OPT}[v][j]$ either contains at most $j-1$ edges, or it consists of an s-u path that contains at most $j-1$ edges followed by the edge (u,v). This means

$$\mathsf{OPT}[v][j] = \min(\mathsf{OPT}[v][j-1], \min_{u:(u,v \in E} \mathsf{OPT}[u][j-1] + \ell(u,v)).$$

Remark

One way to think of Bellman-Ford is the following: we want to apply the recurrence from DAG-DP, but cycles in G create dependency problems. However, we can still "shrink" the SSSP problem by drawing inspiration from 0/1 Knapsack (Section 4.4). To solve that problem, we shrunk the knapsack capacity from B to j; similarly, in Bellman-Ford, we shrink the number of edges we can use from $n-1$ to j.

Algorithm: Column j depends on column $j-1$, so we compute the recurrence column by column. Within column j, we calculate every $d[v][j]$ according to the recurrence. Again, when considering vertex v, we need to access the in-neighbors of v, so we start by computing $G' = G$ with each edge reversed.

```
Bellman-Ford($G, s$):
$d = [\infty] * (n \times n)$; $d[s][0] = 0$
$G' = G$ with each edge reversed
for $j = 1, \ldots, n-1$:
    for $v \in V$:
        $d[v][j] = d[v][j-1]$
        for $u \in G'[v]$:
            $d[v][j] = \min(d[v][j], d[u][j-1] + \ell(u,v))$
return $d[\cdot][n-1]$        // return last column of $d$
```

Notice that in round j of the algorithm, we relax every edge, and the order in which we relax these edges does not matter. Furthermore, since column j only depends on column $j-1$, our DP table does not actually need to store n separate columns. In fact, it suffices for the algorithm to store just one column (i.e., one value $d[v]$ per $v \in V$) and overwrite the values in this column as necessary. This leads us to Bellman-Ford-2, another implementation of essentially the same algorithm.

```
Bellman-Ford-2($G, s$):
$d = [\infty] * n$; $d[s] = 0$
for $j = 1, \ldots, n-1$:
    for $(u,v) \in E$:
        $d[v] = \min(d[v], d[u] + \ell(u,v))$
return $d$
```

Running time: In both implementations, the algorithm makes $n-1$ rounds. In each round, each of the m edges gets relaxed, and relaxing an edge takes $O(1)$ time. Thus, the total running time is $O(mn)$.

5.3 Dijkstra's algorithm

If all edge lengths are non-negative, then we can run Dijkstra's algorithm, which is closely related to both BFS (Section 2.1) and Prim's algorithm (Section 3.1).

Problem Statement

SSSP, where all edge lengths are non-negative. Describe an $O(n^2)$-time algorithm for this problem, prove that it's correct, and analyze its running time.

```
Dijkstra($G, s$):
d = [∞] * n; d[s] = 0; S = ∅
while |S| < n:
    u = vertex in V \ S with smallest d[u]
    add u to S
    for v ∈ G[u]:
        d[v] = min(d[v], d[u] + ℓ(u, v))
return d
```

Algorithm: Initialize $S = \emptyset$ and $d[s] = 0$. While $|S| < n$, find the vertex $u \in V \setminus S$ with the smallest d-value, add it to S, and process it (i.e. relax every edge that leaves u). Once every vertex has been added to S, return d.

Dijkstra's algorithm is greedy in the sense that once a vertex u has the smallest d-value, we "finalize" the value of $d[u]$. Dijkstra's algorithm is also a DP in some sense, since it (along with DAG-DP and Bellman-Ford) relies on the concept of edge relaxations: a shortest s-v path comprises a shortest s-u path plus the edge (u, v).

Correctness: Let $\delta(u)$ denote the distance from s to u; we will show by induction that when u gets added to S, $d[u] = \delta(u)$. The base case is $u = s$, in which case $d[u] = 0 = \delta(u)$.

Now suppose u is the $(k+1)$-th vertex added to S. Let P denote the s-u path found by the algorithm, and let P^* denote any s-u path. Since u is not in S yet, P^* must leave S through some edge (x, y). Since Dijkstra's is about to add u to S, at this point, we have $d[u] \leq d[y] \leq d[x]+\ell(x, y)$. By induction, $d[x] = \delta(x)$, and all edge lengths are positive, so $\ell(P^*) \geq \delta(x) + \ell(x, y)$. Putting this all together, we can conclude $d[u] \leq \ell(P^*)$, which implies $d[u] = \delta(u)$.

Running time: The algorithm makes $O(n)$ iterations. In each iteration, it takes $O(n)$ time to find the vertex $u \in V \setminus S$ that minimizes $d[u]$, and $O(\text{out-deg}(u)) = O(n)$ time to process u. Thus, the total running time is $O(n^2)$.

Remark

The implementation described above uses an array as a priority queue. But like Prim's algorithm (Section 3.1), Dijkstra's algorithm can be implemented in $O(m \log n)$-time using a heap-based priority queue. And if we use a Fibonacci heap, we can reduce this running time to $O(m + n \log n)$.

5.4 Floyd-Warshall

In this section, we solve the All-Pairs Shortest Path (APSP) problem. The edge lengths could be negative, but again, we assume that G has no negative cycles (see Section 5.2).

Problem Statement

The input is G, a directed graph with edge lengths ℓ. Our goal is to return an $n \times n$ array d, where for all $u, v \in V$, $d[u][v]$ is the distance from u to v. Describe an $O(n^3)$-time algorithm for this problem, explain why it's correct, and analyze its running time.

One correct solution is to run Bellman-Ford n times (once per $s \in V$) but its running time is $n \cdot O(mn) = O(mn^2) = O(n^4)$, which is quite large. So instead, we'll use dynamic programming again. This time, instead of reducing the number of edges the path can use, we shrink the problem by reducing the set of vertices the path can use.

Subproblems: For all $u, v \in V$ and $r \in \{0, 1, \ldots, n\}$, let $\mathsf{OPT}[u][v][r]$ denote the length of the shortest walk from u to v that with vertices $[r]$ available as intermediate vertices. (Recall that $V = [n] = \{1, \ldots, n\}$.) We will return $\mathsf{OPT}[\cdot][\cdot][n]$.

Recurrence: If $r = 0$, then we are not allowed to use any vertices as intermediate vertices, so

$$\mathsf{OPT}[u][v][0] = \begin{cases} 0 & \text{if } u = v \\ \ell(u,v) & \text{if } (u,v) \in E \\ \infty & \text{otherwise.} \end{cases}$$

If $r \geq 1$, then the walk for $\mathsf{OPT}[u][v][r]$ either contains r as an intermediate vertex or doesn't. If it doesn't, then it is as if vertex r were not allowed as an intermediate vertex. Otherwise, the walk can be split into a u-r walk plus an r-v walk. Thus, we have

$$\mathsf{OPT}[u][v][r] = \min(\mathsf{OPT}[u][v][r-1], \mathsf{OPT}[u][r][r-1] + \mathsf{OPT}[r][v][r-1]).$$

```
**Floyd-Warshall**($G$):
$d = [\infty] * (n \times n \times (n+1))$
**for** $u \in V$:
    $d[u][u][0] = 0$
    **for** $v \in G[u]$:
        $d[u][v][0] = \ell(u,v)$
**for** $r \in V$:
    **for** $u \in V$:
        **for** $v \in V$:
            $d[u][v][r] = \min(d[u][v][r-1], d[u][r][r-1] + d[r][v][r-1])$
**return** $d[\cdot][\cdot][n]$                                    // return table $n$
```

Algorithm: We can think of the DP "table" as $n+1$ tables, one per $r \in \{0, 1, \ldots, n\}$, where table r is an $n \times n$ array containing the values of $\mathsf{OPT}[u][v][r]$. Table r only depends on values in table $r-1$, so the algorithm can compute the recurrence table by table. Within each table, the order of computation does not matter. At the end, we return table n.

Running time: There are $O(n^3)$ subproblems, and we solve each one in $O(1)$ time, so the total running time is $O(n^3)$.

6 Flows and Cuts

If the shortest path problem is about sending one truck from a location s to another location t, the maximum flow problem is about sending as many trucks as possible from s to t. It also has many surprising applications and extensions.

Overview

Section	Summary	Time
6.1: Ford-Fulkerson	While the residual graph G_f has an s-t path P: augment f along P and update G_f. Vertices reachable from s in last G_f form a minimum s-t cut.	$O(mv)$
6.2: Bipartite Matching	Add edges from s to L and R to t, set all $c(e) = 1$, find maximum flow, return edges with flow.	$O(mn)$
6.3: Bipartite VC	Add edges from s to L and R to t, new edges have $c(e) = 1$, old edges have $c(e) = \infty$, find minimum cut S, return $(L \setminus S) \cup (R \cap S)$.	$\uparrow$

A *flow network* comprises (G, s, t), where G is a directed graph, each edge e has *capacity* $c(e) \in \mathbb{Z}^+$, s is a *source* vertex, and t is a *sink* vertex. We assume that no edges enter s, no edges leave t, and G (ignoring edge directions) is connected. A *cut* is a subset S of vertices; S is an *s-t cut* if $s \in S$ and $t \notin S$. For any cut S, we let $\delta^{\mathsf{out}}(S) = \{(u, v) \in E : u \in S, v \notin S\}$ denote the set of edges leaving (or *crossing*) the cut and $\delta^{\mathsf{in}}(S)$ denote the set of edges entering S.

A *flow* is a function $f\colon E \to \mathbb{R}$; for any flow f and cut S, we let $f^{\mathsf{out}}(S) = \sum_{e \in \delta^{\mathsf{out}}(S)} f(e)$ and define $f^{\mathsf{in}}(S)$ analogously. (If $S = \{u\}$, we write $f^{\mathsf{out}}(u)$ instead of $f^{\mathsf{out}}(\{u\})$.) A flow is *feasible* if it satisfies both of the following conditions:

1. Capacity constraints: For every $e \in E$, $0 \le f(e) \le c(e)$.
2. Conservation: For every $u \in V \setminus \{s, t\}$, $f^{\mathsf{in}}(u) = f^{\mathsf{out}}(u)$.

The *value* of a flow f is defined as $|f| = f^{\mathsf{out}}(s)$; a *maximum flow* is a feasible flow such that $|f|$ is maximized. The *capacity* of a cut S is defined as $c(S) = \sum_{e \in \delta^{\mathsf{out}}(S)} c(e)$; a *minimum s-t cut* is an s-t cut S such that $c(S)$ is minimized.

We now prove a useful lemma that relates flows to cuts.

Lemma 6.1. *For any feasible flow f and s-t cut S, $|f| = f^{\mathsf{out}}(S) - f^{\mathsf{in}}(S) \le c(S)$.*

Proof. Since $|f| = f^{\mathsf{out}}(s) - f^{\mathsf{in}}(s)$ and $f^{\mathsf{out}}(u) - f^{\mathsf{in}}(u) = 0$ for all $u \in S \setminus \{s\}$,

$$|f| = \sum_{u \in S} f^{\mathsf{out}}(u) - f^{\mathsf{in}}(u) = \sum_{u \in S} \left(\sum_{e \in \delta^{\mathsf{out}}(u)} f(e) - \sum_{e \in \delta^{\mathsf{in}}(u)} f(e) \right).$$

The above is the "vertex-centric" way of viewing $|f|$; let's convert it to an "edge-centric" perspective. For any $e = (x, y)$, if $x, y \in S$ then $f(e)$ gets added and subtracted in the sum above, so it contributes nothing to $|f|$. Thus, the only edges that contribute a non-zero amount to $|f|$ are those with exactly one endpoint in S. More specifically, if $x \in S$ and $y \notin S$, then $f(e)$ only contributes positively, and if $x \notin S$ and $y \in S$, then $f(e)$ only contributes negatively. In other words, we have

$$|f| = \sum_{e \in \delta^{\mathsf{out}}(S)} f(e) - \sum_{e \in \delta^{\mathsf{in}}(S)} f(e) = f^{\mathsf{out}}(S) - f^{\mathsf{in}}(S) \leq \sum_{e \in \delta^{\mathsf{out}}(S)} c(e) = c(S). \qquad \square$$

Note that $f(e)$ is just another edge attribute, so if we need to return f, we can incorporate it into an adjacency list as we did for edge weights: for each $u \in V$, $G[u]$ is a list of tuples (v, c, f) where v is an out-neighbor of u, $c = c(u, v)$, and $f = f(u, v)$.

6.1 Ford-Fulkerson

Problem Statement

The input is a flow network (G, s, t), and our goal is to return a maximum s-t flow. Describe an $O(mv)$-time algorithm for this problem (where v is the value of the maximum flow), prove that it's correct, and analyze its running time.

One greedy approach is the following: use DFS (or any pathfinding algorithm) to find an s-t path P, increase $f(e)$ by $\Delta = \min_{e \in P} c(e)$ for all $e \in P$, decrease $c(e)$ by Δ for all $e \in P$, and repeat until we can't make any more progress. This is almost the right idea, but instead of searching for P in G, we should search for P in the *residual network* G_f defined below.

```
Residual(G, f):
1 $G_f = (V(G), E_f = \emptyset)$
2 for $e = (u, v) \in E(G)$:
3     if $f(e) < c(e)$:
4         add $(u, v)$ to $E_f$ with capacity $c(e) - f(e)$    // forward edge
5     if $f(e) > 0$:
6         add $(v, u)$ to $E_f$ with capacity $f(e)$           // backward edge
7 return $G_f$
```

For any flow f, the "forward" edges in G_f represent how much capacity remains in G, and the "backward" edges in G_f represent how much flow in f we can "undo." Notice that $m \leq |E(G_f)| \leq 2m$ where $m = |E(G)|$, as usual.

Algorithm: The Ford-Fulkerson algorithm is the greedy approach described above, except we do not modify G. Instead, in each iteration, we search for an s-t path P using DFS (or any pathfinding algorithm) in G_f, augment f along P by the minimum residual capacity $c_f(e)$ over all $e \in P$, and update G_f by setting $G_f = \text{Residual}(G, f)$.

Correctness: We first prove that the algorithm terminates within v iterations by showing that $|f|$ increases by at least 1 in each iteration. 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 $|f| = f^{\mathsf{out}}(s)$ increases by $\Delta_P \geq 1$.

```
Ford-Fulkerson($G, s, t$):
set $G_f = G$ and all $f(e) = 0$
while $G_f$ has an $s$-$t$ path $P$:
    $\Delta_P = \min_{e \in P} c_f(e)$
    for $e = (u, v) \in P$:
        if $e$ is a forward edge:
            $f(u, v) \mathrel{+}= \Delta_P$
        else:
            $f(v, u) \mathrel{-}= \Delta_P$
    $G_f = \text{Residual}(G, f)$
return $f$
```

Now we prove that the flow f returned by Ford-Fulkerson is a maximum flow.[1] By Lemma 6.1, it suffices to find a cut S^* such that $|f| = c(S^*)$. Let S^* denote the set of vertices reachable from s in G_f at the end of the algorithm (so $t \notin S^*$). For every $e = (u, v)$ leaving S^*, $f(e) = c(e)$ because otherwise, e would be a forward edge in G^f, making v reachable from s, contradicting the assumption that $v \notin S^*$. Similarly, we can show that $f(e') = 0$ for every $e' \in \delta^{\text{in}}(S^*)$. Combining this with Lemma 6.1, we get

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

Running time: As shown in the correctness proof, the algorithm makes at most v iterations. If we use any linear-time pathfinding algorithm (e.g., DFS) to find each augmenting path, then each iteration takes $O(m)$ time, so the total running time is $O(mv)$.

Minimum s-t Cut

The correctness proof of Ford-Fulkerson essentially shows us how to find a minimum s-t cut.

Problem Statement

The input is a flow network (G, s, t), and our goal is to return a minimum s-t cut. Describe an $O(mv)$-time algorithm for this problem (where v is the value of the maximum flow), prove that it's correct, and analyze its running time.

Algorithm: Return S, the set of vertices reachable from s in G_f at the end of Ford-Fulkerson.

Correctness: In the correctness proof of Ford-Fulkerson, we showed that the maximum flow value is $|f| = c(S)$. If there exists an s-t cut S' such that $c(S') < c(S)$, then $c(S') < |f|$, contradicting Lemma 6.1.

Running time: The algorithm consists of Ford-Fulkerson followed by one call of DFS (or any pathfinding algorithm), so (assuming $v \geq 1$) its running time is $O(mv) + O(m + n) = O(mv)$.

[1] Technically, we should also show that f is a feasible flow, but the proof is relatively straightforward and somewhat tedious, so we leave it as an exercise.

6.2 Bipartite Matching

In this section, we describe an application of the maximum flow problem. An undirected graph G is a *bipartite graph* if V can be partitioned into L, R such that every edge in G has exactly one endpoint in L. Given a bipartite graph, in $O(m+n)$ time, we can label each vertex with either L or R. In any undirected graph, a subset of edges is a *matching* if no two edges in the subset share an endpoint.

Problem Statement

The input is a bipartite graph $G = (L \cup R, E)$. Our goal is to return a maximum matching in G. Describe an $O(mn)$-time algorithm for this problem, prove that it's correct, and analyze its running time.

```
Bipartite-Matching($G$):
 1 $G' = (V' = V, E' = E)$
 2 direct every edge in $E'$ from $L$ to $R$
 3 for $e \in E'$:
 4     $c(e) = n$
       /* any $c(e) \in \mathbb{Z}^+$ works, but $c(e) = n$ will be
          useful for the next section                      */
 5 add $s, t$ to $V'$
 6 for $u \in L$:
 7     add $(s, u)$ to $E'$ with capacity 1
 8 for $v \in R$:
 9     add $(v, t)$ to $E'$ with capacity 1
10 $f$ = Ford-Fulkerson($G', s, t$)
11 return $M = \{e \in E \mid f(e) = 1\}$
```

Algorithm: Construct $G' = (V', E')$ as follows: $V' = V \cup \{s, t\}$ where s and t are new vertices; E' contains every edge in G directed from L to R, as well as (s, u) for all $u \in L$ and (v, t) for all $v \in R$. Edges from L to R have capacity $n = |V(G)|$; all other edges have capacity 1. Call Ford-Fulkerson(G', s, t) to find a maximum s-t flow f in G', and return $M = \{e \in E \mid f(e) = 1\}$.

Correctness: We first show that M is a matching. For any $u \in L$, $f^{\mathsf{in}}(u) \leq 1$ and f is conserved at u, so $f^{\mathsf{out}}(u) \leq 1$. Thus, u is incident to at most one edge in M. Similarly, we can show that every $v \in R$ is incident to at most one edge in M.

Furthermore, we can see that $|f| = |M|$ by applying Lemma 6.1 to the cut $S = \{s\} \cup L$. Each edge of M contributes one unit of flow to $f^{\mathsf{out}}(S)$, and $f^{\mathsf{in}}(S) = 0$, so $|f| = |M|$.

Finally, we show that M is a maximum matching. For contradiction, suppose there exists a matching M^* such that $|M^*| > |M|$. We can convert M^* to a flow f^* such that $|M^*| = |f^*|$ by setting $f^*(s, u) = f^*(u, v) = f^*(v, t) = 1$ for all $(u, v) \in M^*$ (and $f^*(e) = 0$ for all other $e \in E'$). Since f^* consists of $|M^*|$ edge-disjoint paths, $|f^*| = |M^*|$. If $|M^*| > |M|$, then $|f^*| > |f|$, contradicting the optimality of Ford-Fulkerson.

Remark

The Bipartite Matching algorithm and its correctness proof follow a common format for optimization problems: given an instance X of some problem A, (1) convert X to an instance Y of another problem B, (2) optimally solve Y to obtain a solution S', and (3) convert S' to a solution S for X. Then show that S is optimal for X because S' is optimal for Y.

Running time: Constructing G' and M takes $O(m+n)$ time. In G', there are $O(m)$ edges and the maximum flow value is at most n (since the capacity of $\{s\}$ is at most n), so Ford-Fulkerson(G', s, t) takes $O(mn)$ time. Thus, the total running time is $O(m+n) + O(mn) = O(mn)$.

6.3 Bipartite Vertex Cover

Recall that Ford-Fulkerson can be extended to solve the minimum s-t cut problem. This allows us to solve another problem on bipartite graphs. A subset S of vertices is a *vertex cover* if every edge has at least one endpoint in S.

Problem Statement

The input is a bipartite graph $G = (L \cup R, E)$. Our goal is to return a minimum vertex cover of G. Describe an $O(mn)$-time algorithm for this problem, prove that it's correct, and analyze its running time.

```
Bipartite-VC(G):
1  construct $(G', s, t)$ as before
2  $S$ = minimum $s$-$t$ cut in $G'$
3  return $C = (L \setminus S) \cup (R \cap S)$
```

Algorithm: Construct $G' = (V', E')$ exactly as we did for Bipartite Matching (Section 6.2). Find a minimum s-t cut S in G' (using the extension of Ford-Fulkerson described in Section 6.1) and return $C = (L \setminus S) \cup (R \cap S)$.

Correctness: We first show that C is a vertex cover. For contradiction, suppose $\{u, v\} \in E$ is not covered by C. Then $u \in L \cap S$ and $v \in R \setminus S$, so in G', (u, v) leaves S. But this means $c(S) \geq n$, contradicting the fact that S is a minimum s-t cut. (The cut $\{s\}$ has capacity $|L| < n$.)

Now we show $|C| = c(S)$. There are two types of edges leaving S: those of the form (s, u) where $u \in L \setminus S$, and those of the form (v, t) where $v \in R \cap S$. (Again, since S is a minimum s-t cut, there cannot be an edge (u, v) leaving S where $u \in L$.) Each of these edges has capacity 1, so $c(S) = |L \setminus S| + |R \cap S| = |C|$.

Finally, we show that C is a minimum vertex cover. For contradiction, suppose C^* is a vertex cover such that $|C^*| < |C|$. Consider the s-t cut $S^* = \{s\} \cup (L \setminus C^*) \cup (R \cap C^*)$. By the same reasoning as above, we have $c(S^*) = |C^*|$, so $c(S^*) < |C| = c(S)$, contradicting the fact that S is a minimum s-t cut.

Running time: This analysis is essentially identical to that in Section 6.2.

7 NP-Hardness

If a problem is NP-hard, it is unlikely that there exists a polynomial-time algorithm for the problem. (In this chapter, exact running times won't be essential for our purposes.)

Overview

Section	Summary
7.1: Reductions, P, and NP	To show B is NP-hard, prove $A \leq B$ for some NP-hard problem A.
7.2: IndependentSet to VertexCover	Return $(G, n-k)$.
7.3: 3SAT to IndependentSet	Create one triangle per clause, add conflict edges.
7.4: VertexCover to DominatingSet	For each edge, add a parallel vertex.
7.5: DirHamCycle to HamCycle	Replace each $u \in V$ with a path $(u_{\mathsf{in}}, u, u_{\mathsf{out}})$ and each $(u,v) \in E$ with $\{u_{\mathsf{out}}, v_{\mathsf{in}}\}$.

This chapters differs from the others in two ways. First, we'll consider *decision problems*, which comprise an input specification and a yes/no question Q. The optimization problems we've seen can be translated to decision problems by introducing an additional input k and asking if the optimal value is at most k (if the problem is a minimization problem) or at least k (if the problem is a maximization problem). Some examples are below.

Optimization problem	Decision version
Minimum spanning tree	Input: (G, k) Q: Is the weight of the MST of G at most k?
0/1 Knapsack	Input: (v, w, B, k) Q: Is the value of the optimal solution at least k?
Maximum Flow	Input: (G, s, t, k) Q: Is the value of the maximum s-t flow at least k?

Second, instead of solving the decision problem, we'll show that it is very unlikely that there exists a polynomial-time algorithm that solves the problem. For a variety of reasons, theoretical computer scientists generally categorize a problem as "easy" if we know that it can be solved in polynomial time and "hard" otherwise.

7.1 Reductions, P, and NP

Let A be a decision problem. Informally, A is in the complexity class "P" if it can be solved in polynomial time, and A is in "NP" if it can be solved by a brute-force algorithm. (Note that P stands for "polynomial time" and NP stands for "nondeterministic polynomial time," *not* "not polynomial time.") It is known that P $\subseteq$ NP, and it is widely believed that P $\neq$ NP (because there are many problems in NP that appear impossible to solve in polynomial time, despite significant efforts), but proving P $\neq$ NP is a famous, open problem.

Remark

For our purposes, it is not critical to fully understand the definitions of P and NP. Instead, we will focus on describing reductions from one decision problem to another. For more on complexity classes, I recommend *Introduction to the Theory of Computation* by Sipser.

Imagine that we knew that a problem A cannot be solved in polynomial time, and we want to prove that another problem B also cannot be solved in polynomial time. For contradiction, we assume there exists a polynomial-time algorithm ALG_B for B, and we'll construct a polynomial-time algorithm for ALG_A. Fortunately, we have access to a polynomial-time algorithm f that transforms every "yes" instance of A into a "yes" instance of B and every "no" instance of A into a "no" instance of B. This makes the construction of ALG_A fairly straightforward: given X, return $\mathsf{ALG}_B(f(X))$.

This construction of ALG_A motivates the following definition: for any decision problems A and B, a *polynomial-time reduction* from A to B is a polynomial-time algorithm f that transforms every instance X of A into an instance $f(X)$ of B such that X is a "yes" instance of A if and only if $f(X)$ is a "yes" instance of B. We let $A \leq B$ denote that A is polynomial-time reducible to B.

Remark

It's natural to wonder if our definition of $A \leq B$ is too strict. That is, perhaps "$A \leq B$" should simply mean the following: if B can be solved in polynomial time, then so can A. (This is known as a *Cook* or *polynomial-time Turing* reduction.) However, for reasons beyond the scope of these notes, we stick to our definition of "$A \leq B$" (i.e., a *Karp* reduction) which is more common in the undergraduate algorithms literature.

So if we want to prove that B cannot be solved in polynomial, time, it suffices to prove $A \leq B$, which means describing a polynomial-time algorithm f that satisfies the following:

- The "forward" direction: If X is a "yes" instance X of A, $f(X)$ is a "yes" instance of B.
- The "backward" direction: If $f(X)$ is a "yes" instance of B, X is a "yes" instance of A.

Remark

It is much simpler to prove $A \leq B$ if we only cared about one direction. For example, if we only cared about the forward direction, the reduction can ignore its input and simply return a tiny "yes" instance of B.

A problem B is NP-**hard** if $A \leq B$ for all $A \in$ NP, and B is **NP-complete** if $B \in$ NP and B is NP-hard.[1] Notice that a polynomial-time algorithm for any NP-hard problem would yield a polynomial-time algorithm for every problem in NP. Furthermore, to show that B is NP-hard, it suffices to show that $A \leq B$ for some NP-hard problem A. It is not obvious that NP-hard problems even exist, but in this chapter, we'll see multiple examples.

[1]There are problems, such as the Halting problem, that are NP-hard and not in NP. On the other hand, Ladner's theorem states that if P $\neq$ NP, then there are problems in NP $\setminus$ P that are not NP-hard.

Remark

Where does the chain of reductions begin? In other words, what was the "first" NP-hard problem? One answer is the 3-Satisfiability Problem (Section 7.3); the *Cook-Levin* theorem proves (from scratch) that it is NP-hard. Again, for further reading, I recommend *Introduction to the Theory of Computation* by Sipser.

In the rest of this chapter, unless stated otherwise, G is an undirected graph and $k \in \mathbb{Z}$.

7.2 Independent Set to Vertex Cover

We start with a relatively simple reduction.

- IndependentSet: Input: (G, k). Q: Does G contain an independent set of size at least k?
- VertexCover: Input: (G, k). Q: Does G contain a vertex cover of size at most k?

Problem Statement

Prove IndependentSet $\leq$ VertexCover.

Reduction. Given (G, k), return $(G, n-k)$, where n is (as usual) the number of vertices in G.

Forward direction. If (G, k) is a "yes" instance of IndependentSet, then G has an indpendent set S of size $|S| \leq k$. Notice that $\overline{S} = V \setminus S$ has size $|\overline{S}| \leq n-k$, so it suffices to show that $\overline{S}$ is a vertex cover of G. For contradiction, suppose there exists $\{u, v\} \in E$ such that $u \notin \overline{S}$ and $v \notin \overline{S}$. This implies $u \in S$ and $v \in S$, which contradicts the fact that S is an independent set in G.

Backward direction. (The proof is similar. For many reductions, the proof of the backward direction is essentially the reverse of the proof of the forward direction.) If $(G, n-k)$ is a "yes" instance of VertexCover, then G has a vertex cover S of size $|S| \leq n-k$. Notice that $\overline{S} = V \setminus S$ has size $|\overline{S}| \geq k$, so it suffices to show that $\overline{S}$ is an independent set in G. For contradiction, suppose there exists $\{u, v\} \in E$ such that $u \in \overline{S}$ and $v \in \overline{S}$. This implies $u \notin S$ and $v \notin S$, which contradicts the fact that S is a vertex cover of G.

7.3 3-SAT to Independent Set

The 3-Satisfiability Problem (3SAT) looks quite different from the problems we've encountered so far, but it is historically significant and has many practical applications.

- 3SAT: Input: A set of variables $\{x_1, \ldots, x_n\}$ and clauses, where each clause is the $\vee$ ("OR") of 3 literals (x_i or $\overline{x_i}$). Q: Is there an assignment $\phi\colon X \to \{\mathsf{T}, \mathsf{F}\}$ such that every clause is T?
- IndependentSet: Input: (G, k). Q: Does G contain an independent set of size at least k?

Problem Statement

Prove 3SAT $\leq$ IndependentSet.

Reduction. For each clause C_j, add a triangle to G by creating one vertex per literal in C_j and connecting the three vertices together. Thus, if there are ℓ clauses, G has 3ℓ vertices and 3ℓ edges so far. For any two vertices in G, if their corresponding literals are negations of each other, then add a "conflict" edge between these two vertices. Return $(G, k = \ell)$.

Forward direction. Suppose ϕ satisfies all clauses. This means each clause contains at least one true literal. Let S be the set of ℓ vertices that correspond to these literals. Notice that $|S| = k$; we claim that S is an independent set. No triangle edge can have both endpoints in S because S contains exactly one vertex per triangle.

Now consider any conflict edge $\{u, v\}$. By construction, u represents a variable x_i and v represents its negation $\overline{x_i}$ (or vice versa). However, x_i cannot be true in one clause while $\overline{x_i}$ is true in another clause, so S cannot contain both u and v, as desired.

Backward direction. Let S be an independent set of size ℓ; we will set every $\phi(x_i)$ such that all clauses are satisfied. Since S has size ℓ and G has ℓ triangles, S contains exactly one vertex per triangle. Our assignment ϕ sets the literals corresponding to vertices in S to T (so if $u \in S$ corresponds to $\overline{x_i}$, we set $\phi(x_i) = \mathsf{F}$). This is a valid assignment because S cannot contain a vertex corresponding to some variable x_i and another vertex corresponding to $\overline{x_i}$. Thus, in every clause, we set at least one of its literals to be T, so ϕ satisfies all clauses.

7.4 Vertex Cover to Dominating Set

A *dominating set* is a subset S of vertices such that for all $u \in V$, $u \in S$ or u has a neighbor in S.

- VertexCover: Input: (G, k). Q: Does G contain a vertex cover of size at most k?
- DominatingSet: Input: (G, k). Q: Does G contain a dominating set of size at most k?

Problem Statement

Prove VertexCover $\leq$ DominatingSet.

```
VC-to-DS$(G, k)$:
G' = G
for $e = \{u, v\} \in E(G)$:
    add $x_e$ to $V(G')$
    add $\{u, x_e\}, \{v, x_e\}$ to $E(G')$
$k' = k + |I(G)|$
return $(G', k')$
```

Reduction. Given (G, k), initialize $G' = G$. For each edge $e = \{u, v\} \in E(G)$, add a vertex x_e to G', as well as the edges $\{u, x_e\}$ and $\{v, x_e\}$. (Intuitively, G' is G with a copy of each edge, and x_e is a new vertex placed "on" the copy.) Set $k' = k + |I(G)|$, where $I(G)$ is the set of isolated vertices in G. Return $(G', k' = k)$.

Forward direction. Let S be a vertex cover of G such that $|S| \leq k$; we claim that $S' = S \cup I(G)$ is a dominating set of G'. Consider any vertex $u \in V(G')$. In all three cases, $u \in S'$ or u has a neighbor in S':

1. If $u \in I(G)$, then $u \in S'$.

2. If u is a new vertex x_e, then the corresponding edge e is covered by some vertex in S, so u has a neighbor in S'.

3. If $u \in V(G) \setminus I(G)$, then u is incident to at least one edge $\{u, v\} \in E(G)$. Since S covers this edge, $u \in S$ or $v \in S$.

Backward direction. Let S_1 be a dominating set of G' such that $|S_1| \leq k + |I(G)|$. Notice that S_1 must contain all of $I(G)$, so $S_2 = S_1 \setminus I(G)$ has size $|S_2| \leq k$. Create S_3 by replacing any $x_e \in S_2$ with an endpoint of e (or nothing, if both endpoints are already in S_3), so $S_3 \subseteq V(G)$ and $|S_3| \leq k$. Notice that S_3 is still a dominating set of G' (excluding $I(G)$); we claim that S_3 is a vertex cover of G. Consider any $e = \{u, v\} \in E(G)$. Since x_e is dominating by S_3, $u \in S_3$ or $v \in S_3$, so e is covered by S_3.

Remark

When proving VertexCover $\leq$ DominatingSet, it might be tempting to return $(G', k') = (G, k)$. It is also might tempting to simply place a vertex x_e directly "on" each edge e rather than a copy of e. Neither of these reductions work, but in general, it can be helpful to consider what goes wrong if the reduction minimally modifies its input.

7.5 Directed to Undirected Hamiltonian Cycle

In any graph, a *Hamiltonian cycle* is a cycle that visits every vertex exactly once.

- DirHamCycle: Input: G, a directed graph. Q: Does G contain a Hamiltonian cycle?
- HamCycle: Input: G, an undirected graph. Q: Does G contain a Hamiltonian cycle?

Problem Statement

Prove DirHamCycle $\leq$ HamCycle.

Remark

Proving HamCycle $\leq$ DirHamCycle (assuming $n \geq 3$) is simpler: replace each $\{u, v\} \in E$ with the two directed edges (u, v) and (v, u). This idea often works when reducing an undirected problem to its directed version; the idea below often works for the opposite goal.

Reduction. Construct G' as follows: starting with the empty graph, for each $u \in V(G)$, add vertices $u_{\mathsf{in}}, u, u_{\mathsf{out}}$ to $V(G')$ and edges $\{u_{\mathsf{in}}, u\}, \{u, u_{\mathsf{out}}\}$ to $E(G')$. In other words, each vertex gets replaced by a path of length 2. Then, for each $(u, v) \in E(G)$, add an edge $\{u_{\mathsf{out}}, v_{\mathsf{in}}\}$ to $E(G')$. (So edges entering u in G are incident to u_{in}, and edges leaving u in G are incident to u_{out}.)

Forward direction. Suppose G has a Hamiltonian cycle C. We can construct a Hamiltonian cycle C' in G' by following C. More specifically, whenever C goes from some vertex u to another vertex v, C' takes the (undirected) path $(u, u_{\mathsf{out}}, v_{\mathsf{in}}, v)$.

```
DHC-to-HC($G$):
1 $G' =$ empty graph
2 for $u \in V(G)$:
3     add $u_{\mathsf{in}}, u, u_{\mathsf{out}}$ to $V(G')$
4     add $\{u_{\mathsf{in}}, u\}, \{u, u_{\mathsf{out}}\}$ to $E(G')$
5 for $(u, v) \in E(G)$:
6     add $\{u_{\mathsf{out}}, v_{\mathsf{in}}\}$ to $E(G')$
7 return $G'$
```

Backward direction. Suppose G' has a Hamiltonian cycle C', and consider any vertex $u \in V(G)$. Since u has exactly two neighbors in G', namely u_{in} and u_{out}, C' must contain $(u_{\mathsf{in}}, u, u_{\mathsf{out}})$ as a subpath. (Alternatively, C' could contain $(u_{\mathsf{out}}, u, u_{\mathsf{in}})$, but in that case, we reverse C'.) After visiting u_{out}, C' must visit some vertex v_{in} followed by v, v_{out}. Thus, if we remove all vertices in $V(G') \setminus V(G)$ from C', the result is a Hamiltonian cycle in G.

8 Approximation Algorithms

Approximation algorithms don't always return an optimal solution, but their solutions are often close to optimal. They always run in polynomial time, and they are often relatively simple. (Again, in this chapter, exact running times won't be essential for our purposes.)

Overview

Section	Summary	Approx. ratio
8.1: Vertex Cover	Return endpoints of any maximal matching	2
8.2: Load Balancing	Greedy: Assign each job to machine with smallest current load	↑
8.3: Metric k-Center	Greedy: Keep picking the vertex farthest from its current center	↑
8.4: Maximum Cut	Local search: Keep improving the current solution by moving one vertex at a time	1/2

A α-*approximation algorithm* is a polynomial-time algorithm that always returns a solution whose value ALG is within a factor α of the optimal value OPT. In other words, for minimization problems, there exists $\alpha \geq 1$ such that ALG satisfies the following on every instance:

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

The *approximation ratio* of an algorithm is the smallest α such that the algorithm is an α-approximation algorithm. Proving that an α-approximation algorithm is "correct" means showing that the algorithm satisfies $\mathsf{ALG} \leq \alpha \cdot \mathsf{OPT}$ on every instance. (For maximization problems, we flip the inequalities above, so $\alpha \leq 1$ and we want $\mathsf{OPT} \geq \mathsf{ALG} \geq \alpha \cdot \mathsf{OPT}$.)

8.1 Vertex Cover

Problem Statement

The input is an undirected graph G. Our goal is to return a minimum vertex cover of G. Describe a 2-approximation algorithm for this problem and prove that it's correct.

```
VC-Matching($G, k$):
1 $S$ = empty set
2 while $G$ has an edge $e = \{u, v\}$:
3     add $u, v$ to $S$
4     remove $u, v$ from $G$
5 return $S$
```

Algorithm: Set $S = \emptyset$. If G has an edge $e = \{u, v\}$, add both endpoints of e to S, and remove u and v from G. (Note that when we remove a vertex from a graph, we also remove all edges incident to the vertex.) Repeat this process until G has no edges and return S.

Correctness: Notice that the edges considered by the algorithm form a matching M, so $|\mathsf{ALG}| = 2 \cdot$

$|M|$. Furthermore, each vertex in OPT can cover at most one edge in M, so $|\mathsf{OPT}| \geq |M|$. Thus, $|\mathsf{ALG}| = 2 \cdot |M| \leq 2 \cdot |\mathsf{OPT}|$.

8.2 Load Balancing

Scheduling is a rich topic with many applications. In a typical scheduling problem, there are n jobs and m machines, and we need to assign each job to a machine. Every job has a length, and the *load* on a machine is the sum of the lengths of the jobs assigned to that machine. The *makespan* of an assignment is the maximum load created by that assignment.

Problem Statement

The input is (ℓ, m), where ℓ is an array of n positive integers representing job lengths and m is the number of machines. Our goal is to return an assignment of jobs to machines that minimizes the makespan. Describe a 2-approximation algorithm for this problem and prove that it's correct.

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

Remark

Notice that this algorithm makes its final decisions without knowing what the future holds. Algorithms that satisfy this property are known as *online* algorithms.

Correctness: Suppose ALG achieves a makespan of T, the load of $i \in [m]$ is T, and k was the last job assigned to i. When ALG assigned k to i, i had the smallest load. This means the load on every machine is at least $T - \ell[k]$, so $\sum_{j=1}^{n} \ell[j] \geq m(T - \ell[k])$. Rearranging yields

$$T \leq \frac{\sum_{j=1}^{n} \ell[j]}{m} + \ell[k].$$

Now let T^* denote the optimal makespan. Since the average load in any assignment is $\sum_j \ell[j]/m$, we have $T^* \geq \sum_j \ell[j]/m$. Combining this with the inequality above, we get

$$T \leq \frac{\sum_{j=1}^{n} \ell[j]}{m} + \ell[k] \leq T^* + T^* = 2T^*,$$

where $\ell[k] \leq T^*$ because OPT must assign job k to some machine.

An improved analysis

In ALG, since every machine has load at least $T - \ell[k]$ when k was assigned to i, the sum of all job lengths *excluding job i* is at least $m(T - \ell[k])$. Thus, we can improve the analysis from above:

$$T \leq \frac{\sum_{j=1}^{n} \ell[j] - \ell[k]}{m} + \ell[k] = \frac{\sum_{j=1}^{n} \ell[j]}{m} + \left(1 - \frac{1}{m}\right) \ell[k] \leq \left(2 - \frac{1}{m}\right) T^*.$$

This implies that the algorithm is a $(2 - 1/m)$-approximation algorithm.

8.3 Metric k-Center

Clustering is another rich topic with many applications; here is a classical problem. A distance function d on a set of points X is a *metric* if it satisfies all of the following properties:

- $d[u][v] \geq 0$ for all $u, v \in X$,
- $d[u][v] = 0$ if and only if $u = v$,
- (Symmetry) $d[u][v] = d[v][u]$ for all $u, v \in X$, and
- (Triangle inequality) $d[u][w] \leq d[u][v] + d[v][w]$ for all $u, v, w \in X$.

Given any $S \subseteq X$, we create clusters by assigning each point in X to its closest point in S and think of each $s \in S$ as the center of its cluster. The *radius* of a cluster is the maximum distance from its center to a point in the cluster.

Problem Statement

The input is (d, k), where d is an $n \times n$ array representing a metric on a set of points $X = [n]$ and $k \in \mathbb{Z}^+$. Our goal is to return a subset S of k points such that the maximum radius across the k clusters is minimized. Describe a 2-approximation algorithm for this problem and prove that it's correct.

Algorithm: The algorithm is greedy: 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.

Remark

This situation is similar to that of Load Balancing (Section 8.2): in both cases, the algorithm greedily minimizes the maximum value across multiple entities (i.e., machines or clusters).

Correctness: Let r^* denote the optimal maximum radius, and for any center $c \in X$, let $\mathsf{OPT}(c)$ denote the set of points OPT assigns to c. Consider any point $p \in X$, and suppose $p \in \mathsf{OPT}(t^*)$. There are two cases:

1. If ALG contains a center $t \in \mathsf{OPT}(t^*)$, then

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

where the first inequality is the triangle inequality and the second inequality holds because OPT assigns both p and t to t^*.

2. If $\mathsf{ALG} \cap \mathsf{OPT}(t^*) = \emptyset$, by the pigeonhole principle, there must exist $c \in \mathsf{OPT}$ such that $\mathsf{OPT}(c)$ contains (at least) two centers $u_1, u_2 \in \mathsf{ALG}$. Again, we have

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

Let's assume u_1 was added to ALG before u_2. When u_2 was added, it was the farthest point in X from its closest center, so every $p \in X$ is within distance $2r^*$ to its closest center.

In both cases, we have shown that the distance from p to its closest center in ALG is at most $2r^*$.

8.4 Maximum Cut

Suppose G is an undirected graph. For any cut $S \subseteq V$, we let $\delta(S)$ denote the set of edges crossing S (i.e., those with exactly one endpoint in S). If each edge e has weight $w(e)$, for any $F \subseteq E$, we let $w(F) = \sum_{e \in F} w(e)$.

Problem Statement

The input is G, an undirected graph where each edge e has weight $w(e) \in \mathbb{Z}^+$. Our goal is to return a cut S that maximizes $w(\delta(S))$. Describe a 1/2-approximation algorithm for this problem and prove that it's correct.

```
**Max-Cut**($G$):
1 $S$ = empty set                              // $S$ can be any cut
2 while we can increase $w(\delta(S))$ by moving one vertex $u$:
3     move $u$ from $S$ to $V \setminus S$ (or vice versa)
4 return $S$
```

Algorithm: The algorithm is a "local search" algorithm: Start with any 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. (It might not be obvious that this algorithm even terminates, but we'll show that it does.)

Correctness: In each iteration, $w(\delta(S))$ increases by at least 1, and the maximum possible value of $w(\delta(S))$ is $w(E)$. Thus, the algorithm terminates in at most $w(E)$ iterations.

Remark

The running time of Max-Cut is not polynomial because $w(E)$ could be, say, 2^n. However, there is a polynomial-time 0.49-approximation algorithm that is only slightly more complicated than Max-Cut. (The idea is to move u only if $w(\delta(S))$ increases by at least a factor of $(1 + 0.1/n)$ or so; the analysis is similar.)

Now consider any vertex u at the end of the algorithm, and let $\alpha(u)$ denote the total weight of edges incident to u that contribute to $w(\delta(S))$. If we move u across the partition $(S, V \setminus S)$, $w(\delta(S))$ would decrease by $\alpha(u)$ and increase by $w(\delta(u)) - \alpha(u)$. Since the algorithm terminated, the net increase is at most 0, so $\alpha(u) \geq w(\delta(u))/2$. In other words, every vertex is contributing at least half as much as it possibly could to $w(\delta(S))$. Thus, we can take the sum over all vertices to obtain the desired result. More specifically, $\sum_{u \in V} \alpha(u) = 2 \cdot w(\delta(S))$ and $\sum_{u \in V} w(\delta(u)) = 2 \cdot w(E)$, so

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