Algorithms

CS 341

Lap Chi Lau

LaTeXed by Sibelius Peng

Preface

Disclaimer Much of the information on this set of notes is transcribed directly/indirectly from the lectures of CS 341 during Spring 2021 as well as other related resources. I do not make any warranties about the completeness, reliability and accuracy of this set of notes. Use at your own risk.

I couldn't create a good set of notes from Winter 2020 (the term I was in), so I decided to create a set of notes based on Prof. Lau's version. The notes is solely based on his written course notes, not his videos, not his PPTs. Because I am familiar with the concepts, I might skip lots of details in the notes. Please use this set of notes at your own risk.

The main focus of the course is on the design and analysis of efficient algorithms, and these are fundamental building blocks in the development of computer science. We will cover

- divide and conquer and solving recurrence,
- simple graph algorithms using BFS and DFS,
- greedy algorithms,
- dynamic programming,
- bipartite matching,
- NP-completeness and reductions.

The idea of reduction can also be found in CS 365 and CS 360.

There are three reference books and we refer to them using the following shorthands:

- [DPV] Algorithms, by Dasgupta, Papadimitriou and Vazirani.
- [KT] Algorithm design, by Kleinberg and Tardos.
- [CLRS] Introduction to algorithms, by Cormen, Leiserson, Rivest, and Stein.

Note that Ronald **R**ivest is one of the creators of RSA cryptosystem. See CO 487 for more details.

For any questions, send me an email via `https://notes.sibeliusp.com/contact`.

You can find my notes for other courses on `https://notes.sibeliusp.com/`.

Sibelius Peng

Contents

List of Algorithms

1

Introduction

To introduce you to the course, different instructors use different examples. Stinson uses 3SUM problem. During Fall 2019, Lubiw & Blais (and possible for several future offerings) develop algorithms for merging two convex hulls, which is quite interesting. However, in Winter 2020, the motivating example was max subarray problem, which was studied already in CS 136, maybe not in CS 146. Now let's dive into Spring 2021 offering.

1.1 Two Classical Problems

We are given an undirected graph with n vertices and m edges, where each edge has a non-negative cost. The **traveling salesman problem** asks us to find a minimum cost tour to visit every vertex of the graph at least once (visit all cities). The **Chinese postman problem** asks us to find a minimum cost tour to visit every edge of the graph at least once (visit all streets).

A naive algorithm to solve the TSP is to enumerate all permutations and return the minimum cost one. This takes $O(n!)$ time which is way too slow. By using dynamic programming, we can solve it in $O(2^n)$ time, and this is essentially the best known algorithm that we know of.

We will prove this problem is "NP-complete", and probably efficient algorithms for this problem do not exist. However, people may design some approximation algorithms, which will be covered in CS 466 and CO 754.

Surprisingly, the Chinese postman problem, which looks very similar, can be solved in $O(n^4)$, using techniques from graph matching.

1.2 Time Complexity

How do we define the time complexity of an algorithm?

Roughly speaking, we count the number of operations that the algorithm requires. One may count exactly how many operations. The precise constant is probably machine-dependent and may be also difficult to work out. So, the standard practice is to use asymptotic time complexity to analyze algorithms.

Asymptotic Time Complexity

Given two functions $f(n), g(n)$, we say

- (upper bound, big-O) $g(n) = O(f(n))$ if $\lim_{n\to\infty} \frac{g(n)}{f(n)} \leq c$ for some constant c (independent of n).
- (lower bound, big-Ω) $g(n) = \Omega(f(n))$ if $\lim_{n\to\infty} \frac{g(n)}{f(n)} \geq c$ for some constant c.
- (same order, big-Θ) $g(n) = \Theta(f(n))$ if $\lim_{n\to\infty} \frac{g(n)}{f(n)} = c$ for some constant c.
- (loose upper bound, small-o) $g(n) = o(f(n))$ if $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$.
- (loose lower bound, small-ω) $g(n) = \omega(f(n))$ if $\lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty$.

Worst Case Complexity

We say an algorithm has time complexity $O(f(n))$ if it requires at most $O(f(n))$ primitive operations for all inputs of size n (e.g., n bits, n numbers, n vertices, etc). By adopting the asymptotic time complexity, we are ignoring the leading constant and lower order terms.

"Good" Algorithms

For most optimization problems, such as TSP, there's a straightforward exponential time algorithm. For those problems, we are interested in designing a polytime algorithm for it.

1.3 Computation models

Note that this section often appears in the short answer questions from in-person midterms, in order to trick the student...

When we say that we have an $O(n^4)$-time algorithm, we need to be more precise about what are the primitive operations that we assume. In this course, we usually assume the **word-RAM** model, in which we assume that we can access an arbitrary position of an array in constant time, and also that each word operation (such as addition, read/write) can be done in constant time. This model is usually good in practice, because the problem size can usually be fit in the main memory, and so the word size is large enough and each memory access can be done in more or less the same time.

For problems like computing the determinant, we usually consider the bit-complexity, i.e., how many bit operations involved in the algorithm. So please pay some attention to these assumptions when we analyze the time complexity of an algorithm, especially for numerical problems. That said, the word-complexity and bit-complexity usually don't make a big difference in this course (e.g., at most a $\log(n)$ factor) and so we just use the word-RAM model.

However, in some past midterms, the questions might trick you on this. Also, it does make a difference when we analyze an algorithm in terms of the input size. Read the trial division example in Section 7.4.1 of CO 487.

1.4 3SUM

3SUM

We are given $n+1$ numbers $a_1, a_2, \ldots, a_n$ and c and we would like to determine if there are i, j, k such that $a_i + a_j + a_k = c$.

Algorithm 1 Enumerate all triples and check whether its sum is c. Time: $O(n^3)$.

Algorithm 2 Observe that $a_i + a_j + a_k = c$ can be rewritten as $c - a_i - a_j = a_k$. We can enumerate all paris $a_i + a_j$ and check whether $c - a_i - a_j$ is equal to some a_k.

To do this efficiently, we can first sort the n numbers so that $a_1 \leq a_2 \leq \cdots \leq a_n$. Then checking whether $c - a_i - a_j = a_k$ for some k via binary search in $O(\log n)$. So the total complexity is

$$O(\underbrace{n \log n}_{\text{sorting}} + \underbrace{n^2 \log n}_{\text{binary search for each pair}}) = O(n^2 \log n).$$

Algorithm 3 We can rewrite the condition: $a_i + a_j = c - a_k$. Suppose again that we sort n numbers so that $a_1 \leq a_2 \leq \cdots \leq a_n$. The idea is that given this sorted array and the number $b := c - a_k$, we can check whether there are i, j such that $a_i + a_j = b$. In other words, the 2-SUM problem can be solved in $O(n)$ time given the sorted array. If this is true, we then get an $O(n^2)$-time algorithm for 3-SUM, by trying $b := c - a_k$ for each $1 \leq k \leq n$. That is, we reduce the 3-SUM problem to n instances of the 2-SUM problem.

Algorithm 1: 2-SUM

```
1 L := 1, R := n // left index and right index
2 while L ≤ R do
3     if a_L + a_R = c then
4         DONE
5     else if a_L + a_R > c then
6         R ← R − 1
7     else
8         L ← L + 1
```

Proof of correctness If there are no i, j such that $a_i + a_j = b$, then we won't find them. Now suppose $a_i + a_j = b$ for some $i \leq j$. Since the algorithm runs until $L = R$, there is an iteration such that $L = i$ or $R = j$. WLOG, suppose that $L = i$ happens first, and $R > j$; the other case is symmetric. As long as $R > j$, we have $a_i + a_R > a_i + a_j = b$, and so we will decrease R until $R = j$, and so the algorithm will find this pair. □

Time Complexity $O(n)$m because $R - L$ decrease by one in each iteration, so the algorithm will stop within $n - 1$ iterations.

2

Divide and Conquer

2.1 Merge Sort

Sorting is a fundamental algorithmic task, and merge sort is a classical algorithm using the idea of divide and conquer. This divide and conquer approach works if there is a nice way to reduce an instance of the problem to smaller instances of the same problem. The merge sort algorithm can be summarized as follows:

Algorithm 2: Merge sort

```
1 Function Sort(A[1, n]):
2     if n = 1 then
3         return
4     Sort(A[1, ⌈n/2⌉])
5     Sort(A[⌈n/2⌉ + 1, n])
6     merge(A[1, ⌈n/2⌉], A[⌈n/2⌉ + 1, n])
```

The correctness of the algorithm can be proved formally by a standard induction. We focus on analyzing the running time. We can draw a recursion tree as shown in the course note. Then the asymptotic complexity of merge-sort is $O(n \log n)$. This complexity can also be proved by induction, by using the "guess and check" method.

2.2 Solving Recurrence

By drawing the recursion tree and discussing by cases, we can derive the master theorem:

Master Theorem

If $T(n) = aT\left(\frac{n}{b}\right) + n^c$ for constants $a > 0, b > 1, c \geq 0$, then

$$T(n) = \begin{cases} O(n^c) & \text{if } c > \log_b a \\ O(n^c \log n) & \text{if } c = \log_b a \\ O(n^{\log_b a}) & \text{if } c < \log_b a \end{cases}$$

Single subproblem

This is common in algorithm analysis. For example,

- $T(n) = T\left(\frac{n}{2}\right) + 1$, we have $T(n) = O(\log n)$, binary search.
- $T(n) = T\left(\frac{n}{2}\right) + n$, we have $T(n) = O(n)$, geometric sequence.
- $T(n) = T(\sqrt{n}) + 1$, we have $T(n) = O(\log \log n)$, counting levels.

  (In level i, the subproblem is of size $n^{2^{-i}}$. When $i = \log \log n$, it becomes $n^{\frac{1}{\log n}} = O(1)$.)

Non-even subproblems

We will see one interesting example later.

- $T(n) = T\left(\frac{2n}{3}\right) + T\left(\frac{n}{3}\right) + n$. We have $T(n) = O(n \log n)$.
- $T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{4}\right) + n$. We have $T(n) = O(n)$.

Exponential time

$T(n) = 2T(n-1) + 1$. We have $T(n) = O(2^n)$. Can we improve the runtime if we have $T(n) = T(n-1) + T(n-2) + 1$? This is the same recurrence as the Fibonacci sequence. Using "computing roots of polynomials" from MATH 249, it can be shown that

$$T(n) = O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right) = O(1.618^n),$$

faster exponential time.

Consider the maximum independent set problem:

Maximum independent set

Given $G = (V, E)$. We want to find a maximum subset of vertices $S \subseteq V$ suc that there are no edges between every pair of vertices $u, v \in S$.

A naive algorithm is to enumerate all subsets, taking $\Omega(2^n)$ time. Now consider a simple variant.

Pick a vertex v with maximum degree.

- If $v \notin S$, delete v and reduce the graph size by one.
- If $v \in S$, we choose v, and then we know that all neighbors of v cannot be chosen, and so we can delete v and all its neighbors, so that the graph size is reduced by at least two.

So $T(n) \leq T(n-1) + T(n-2) + O(n)$, and it is strictly smaller than $O(2^n \cdot n)$.

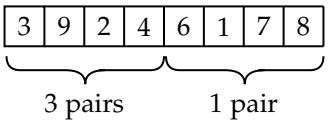
2.3 Counting Inversions

Input: n distinct numbers, $a_1, a_2, \ldots, a_n$
Output: number of paris with $i < j$ but $a_i > a_j$

For example, given $(3, 7, 2, 5, 4)$, there are five pairs of inversions $(3, 2)$, $(7, 2)$, $(7, 5)$, $(7, 4)$, $(5, 4)$.

We can think of this problem as computing the "unsortedness" of a sequence. We may also imagine that this is measuring how different are two rankings.

For simplicity, we again assume that $n = 2^k$ for k integer. Using the idea of divide and conquer, we try to break the problem into two halves. Suppose could count the number of inversions in the first half, as well as in the second half. Would it then be easier to solve the remaining problem?



It remains to count the number of inversions with one number in the first half and the other number in the second half. These "cross" inversion pairs ar easier to count, because we know their relative positions. In particular, to facilitate the counting, we could sort the first half and the second half, without worrying losing information as we already counted the inversion pairs with each half.

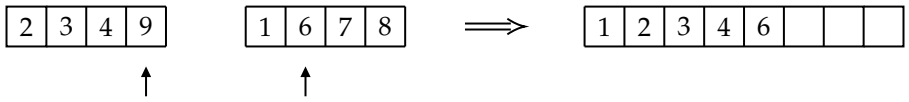
Now, for each number c in the second half, the number of "cross" inversion pairs involving c is precisely the number of numbers in the first half that is larger than c. In this example, 1 is involved in 4 cross pairs, $6, 7, 8$ are all involved in 1 cross pair (with 9), and so the number of "cross" inversion pairs is 7.

How to count the number of cross inversion pairs involving a number a_j in the second half efficiently?

Idea 1 as the first half is sorted, we can use binary search to determine how many numbers in the first half are greater than a_j. This takes $O(\log n)$ time for one a_j, and totally $O(n \log n)$ time for all numbers in the second half. This is not too slow, but we can do better.

Idea 2 Observe that this information can be determined when we merge the two sorted list in merge sort. When we insert a number in the second half to the merged list, we know how many numbers in the first half that are greater than it.

For example,



we know that there is only one number in the first half that is greater than 6. As in merge sort, this can be done in $O(n)$ time.

Algorithm 3: Count cross inversions

```
1 Function count(A[1, n]):
2     if n = 1 then return 0
3     count(A [1, n/2])
4     count(A [n/2 + 1, n])
5     merge-and-count-cross-inversions(A [1, n/2], A [n/2 + 1, n])
```

Total time complexity is $T(n) = 2T(\frac{n}{2}) + O(n \log n)$. Solving this will give $T(n) = \Theta(n \log^2 n)$.

The sorting step is the bottleneck and unnecessary as we have sorted them in the merge step. As it

turns out, we can just modify the merge sort algorithm to count the number of inversion pairs.

Algorithm 4: Count cross inversions (final version)

Function `count-and-sort`$(A[1,n])$**:**
  if $n = 1$ **then return** 0
  $s_1 \leftarrow$ `count-and-sort`$\left(A\left[1, \frac{n}{2}\right]\right)$
  $s_2 \leftarrow$ `count-and-sort`$\left(A\left[\frac{n}{2}+1, n\right]\right)$
  $s_3 \leftarrow$ merge-and-count-cross-inversions$\left(A\left[1, \frac{n}{2}\right], A\left[\frac{n}{2}+1, n\right]\right)$
  return $s_1 + s_2 + s_3$

Total time complexity $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$. Solving this will give us $T(n) = O(n \log n)$.

2.4 Maximum Subarray

See CLRS 4.1. Skipped. $O(n)$ solution is using dynamic programming.

2.5 Finding Median

Input: n distinct numbers $a_1, a_2, \ldots, a_n$.
Output: the median of these numbers.

It's clear thar problem can be solved in $O(n \log n)$ time by first sorting the numbers, but it turns out that there is an interesting $O(n)$-time algorithm. To solve the median problem, it is more convenient to consider a slightly more general problem.

Input: n distinct numbers $a_1, a_2, \ldots, a_n$ and an integer $k \geq 1$.
Output: the k-th smallest number in $a_1, \ldots, a_n$.

The reason is that the median problem doesn't reduce to itself (and so we can't recurse), while the k-th smallest number lends itself to reduction as we will see.

The idea is similar to that in quicksort (which is a divide and conquer algorithm). We choose a number a_i. Split the n numbers into two groups, one group with numbers smaller than a_i, called it S_1, and the other group with numbers greater than a_i, called it S_2.

Let r be the rank of a_i, i.e., a_i is the r-th smallest number in $a_1, \ldots, a_n$.

- If $r = k$, then we are done.
- If $r > k$, the nwe find the k-th smallest number in S_1.
- If $r < k$, then find the $(k-r)$-th smallest number in S_2.

Observe that when $r > k$, the problem size is reduced to $r - 1$ as $|S_1| = r - 1$, and when $r < k$, the problem size is reduced to $n - r$ as $|S_2| = n - r$. So if somehow we could choose a number "in the middle" as a pivot as in quicksort, then we can reduce the problem size quickly and making good progress, but finding a number in the middle is the very question that we want to solve. But obverse that we don't need the pivot to be exactly in the middle, just that it is not too close to the boundary.

Suppose we can choose a_i such that its rank satisfies say $\frac{n}{10} \leq r \leq \frac{9n}{10}$, then we know that the problem size would have reduced by at least $\frac{n}{10}$, as $|S_1| = r - 1 \leq \frac{9n}{10}$ and $|S_2| = n - r \leq \frac{9n}{10}$. So, the recurrence relation for the time complexity is $T(n) \leq \left(\frac{9n}{10}\right) + P(n) + cn$, where $P(n)$ denotes the time to find a good pivot and cn is the number of operations for splitting. if we manage to find a good pivot point in $O(n)$ time, i.e., $P(n) = O(n)$, the it implies that $T(n) = O(n)$. We have made some progress to the

median problem, by reducing the problem of finding the number exactly in the middle to the easier problems of finding a number not too far from the middle.

It remains to figure out a linear time algorithm to find a good pivot.

Randomized solution If we have seen randomized quicksort before, then we would have guessed that keep choosing a random pivot would work. This is indeed the case and we can take a look at [DPV 2.4] for a proof. Details not included in this course, but in CS 761.

Deterministic solution There is an interesting deterministic algorithm that would always return a number a_i with rank $\frac{3n}{10} \leq r \leq \frac{7n}{10}$ in $O(n)$ time. As seen above, this means $P(n) = O(n)$ and it follows that $T(n) = O(n)$.

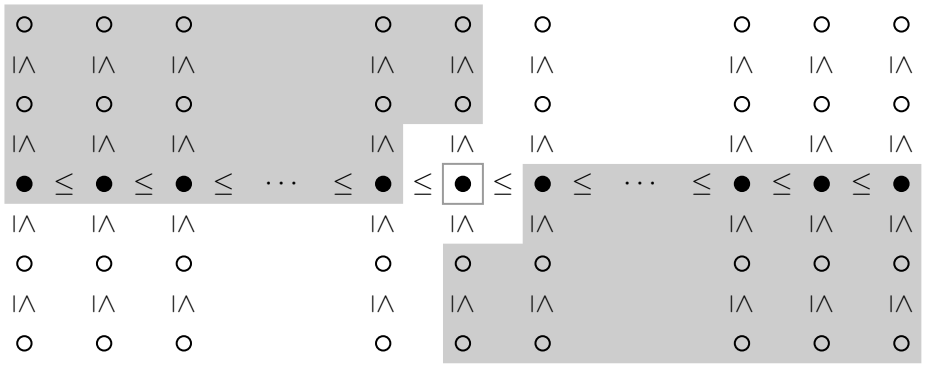
Finding a good pivot The idea of the algorithm is to find the median of medians.

1. Divide the n numbers into $\frac{n}{5}$ groups, each of 5 numbers. Time: $O(n)$.
2. Find the median in each group. Call them $b_1, b_2, \ldots, b_{\frac{n}{5}}$. Time: $O(n)$.
3. Find the median of these $\frac{n}{5}$ medians $b_1, b_2, \ldots, b_{\frac{n}{5}}$. Time: $O(n)$.

Lemma

Let r be the rank of the median of medians. Then $\frac{3n}{10} \leq r \leq \frac{7n}{10}$.

Proof:



The square is the median of the medians.

In the picture, we sort each group, and then order the groups by an increasing order of the medians. We emphasize that this is just for the analysis, and we don't need to do sorting in the algorithm. It should be clear that the square is greater than the numbers in the top-left corner, and is smaller than the numbers in the bottom-right corner.

There are about $3 \cdot \frac{n}{10} = \frac{3n}{10}$ numbers in the top-left and bottom-right corners. This implies that $\frac{3n}{10} \leq 4 \leq \frac{7n}{10}$. □

This lemma proves the correctness of the pivoting algorithm.

Time complexity

We have $P(n) = T\left(\frac{n}{5}\right) + c_1 n$, where c_1 is a constant. By the reduction above,

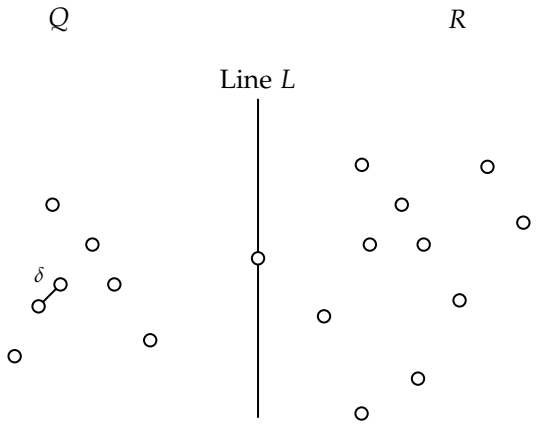
$$T(n) \leq T\left(\frac{7n}{10}\right) + P(n) + c_2 n = T\left(\frac{7n}{10}\right) + T\left(\frac{n}{5}\right) + (c_1 + c_2)n$$

Then $T(n) = O(n)$.

2.6 Closest Pair

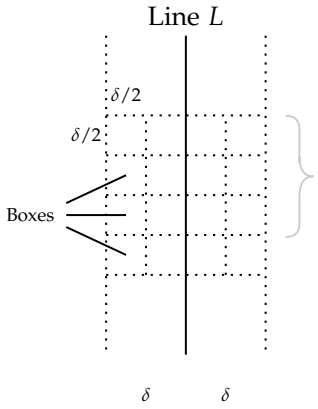
Input: n points (x_1, y_1), (x_2, y_2), ..., (x_n, y_n) on the 2D-plane.
Output: $1 \leq i < j \leq n$ that minimizes the Euclidean distance $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$

It is clear that this problem can be solved in $O(n^2)$ time, by trying all pairs. We use the divide and conquer approach to given an improved algorithm.



We find a vertical line L to separate the point set into two halves: call the set of points on the left of the line Q, and the set of points on the right of the line R. For simplicity, we assume that every point has a distinct x-value. We leave it as an exercise to see where this assumption is used and also how to remove it. The vertical line can be found by computing the median based on the x-value, and put the first $\lceil \frac{n}{2} \rceil$ points in Q, and the last $\lfloor \frac{n}{2} \rfloor$ points in R.

It doesn't seem that the closest crossing pair problem is easier to solve. The idea is that we only need to determine whether there is a crossing pair with distance $< \delta$. This allows us to restrict attention to the points with x-value within δ to the line L, but still all the points can be here. We divide the narrow region into square boxes of side length $\frac{\delta}{2}$ as shown in the picture. Here comes the important observations.



Observation 1 Each square box has at most one point.

Proof:

If two points are in the same box, their distance is at most $\sqrt{\left(\frac{\delta}{2}\right)^2 + \left(\frac{\delta}{2}\right)^2} = \frac{\delta}{\sqrt{2}} < \delta$. This would contradict that the closes pairs within Q and within R have distance $\geq \delta$. □

Observaion 2 Each point needs only to compute distances with points within two horizontal layers.

Proof:
For two points which are separated by at least two horizontal layers, then their distance would be more than δ and would not be closest. □

With observation 2, every point only needs to compute distances with at most eleven other points, in order to search for the closest pairs (i.e., pairs with distance $\leq \delta$). This cuts down the search space from $\Omega(n^2)$ to $O(n)$ pairs.

Algorithm 5: Finding the minimum distance

1. Find the dividing line L by computing the median using the x-value.
2. Recursively solve the closest pair problem in Q and in R. Get δ.
3. Using a linear scan, remove all the points not within the narrow region defined by δ.
4. Sort the points in non-decreasing order by their y-value.
5. For each point, we compute its distance to the next eleven points in this y-ordering. `// Note that two points within two layers must be within 11 points in the y-order, as` ≤ 10 `boxes in between.`
6. Return the minimum distance found.

The correctness of the algorithm is established by the two observations, justifying that it suffices for each point to compute distance to $O(1)$ other points as described in step 5.

Time complexity $T(n) = 2T(\frac{n}{2}) + O(n \log n)$. Solving this gives us $T(n) = O(n \log^2 n)$.

Note that the bottleneck is in the sorting step and it is not necessary to do sorting within recursion. We can sort the points by y-value once in the beginning and use this ordering throughout the algorithm. This reduces the time complexity to $T(n) = 2T(\frac{n}{2}) + O(n) \Rightarrow T(n) = O(n \log n)$.

Similarly, we don't need to compute the medians within the recursion. We can sort the points by x-value once in the beginning and use it for the dividing step. This would not improve the worst case time complexity but would improve its practical performance.

Questions:

1. Where did we use the assumption that the x-values are distinct?
2. What do we need to change so that the algorithm would work without this assumption?

There is a randomized algorithm to find a closest pair in expected $O(n)$ time.

2.7 Arithmetic Problems

Arithmetic problems are where the divide and conquer approach is most powerful. Many fastest algorithms for basic arithmetic problems are based on divide and conquer. Today we will see some basic ideas how this approach works, but unfortunately we will not see the fastest algorithms as they require sme background in algebra.

Integer Multiplication

Given two n-bit numbers $a = a_1 a_2 \cdots a_n$ and $b = b_1 b_2 \cdots b_n$, we would like to compute ab efficiently. The multiplication algorithm learnt in elementary school takes $\Theta(n^2)$ bit operations.

Let's apply the divide and conquer approach to integer multiplication. Suppose we know how to multiply n-bit numbers efficiently, we would like to apply it to $2n$-bit numbers.

Given two $2n$-bit numbers x and y, we write $x = x_1 x_2$ and $y = y_1 y_2$, where x_1, y_1 are higher-order

n-bits and x_2, y_2 are the lower-order n-bits. In other words, $x = x_1 \cdot 2^n + x_2$ and $y = y_1 \cdot 2^n + y_2$. Then

$$xy = (x_1 \cdot 2^n + x_2)(y_1 \cdot 2^n + y_2) = x_1y_1 \cdot 2^{2n} + (x_1y_2 + x_2y_1) \cdot 2^n + x_2y_2 \,.$$

Since x_1, x_2, y_1, y_2 are n-bit numbers, then the products here can be computed recursively. Therefore, $T(n) = 4\left(\frac{n}{2}\right) + O(n)$, where the additional $O(n)$ bit operations are used to add the numbers. (Note that $x_1y_1 \cdot 2^{2n}$ is simply shifting x_1y_1 to the left by $2n$ bits; we don't need a multiplication operation.) Solving the recurrence will give $T(n) = O(n^2)$, not improving the elementary school algorithm.

This should not be surprising, since we haven't done anything clever to combine the subproblems, and we should not expect that just by doing divide and conquer, some speedup would come automatically.

Consider **Karatsuba's algorithm**, which is a clever way to combine the subproblems. Instead of computing four subproblems, Karatsuba's idea is to use three subproblems to compute x_1y_1, x_2y_2 and $(x_1 + x_2)(y_1 + y_2)$. After that we can compute the middle term $x_1y_2 + x_2y_1$ by noticing that

$$(x_1 + x_2) \cdot (y_1 + y_2) - x_1y_1 - x_2y_2 = x_1y_1 + x_1y_2 + x_2y_1 + x_2y_2 - x_1y_1 - x_2y_2 = x_1y_2 + x_2y_1 \,.$$

That is, the middle term can be computed in $O(n)$ bit operations after solving the three subproblems. Therefore, the total complexity is $T(n) = 3T\left(\frac{n}{2}\right) + O(n)$, and it follows from master theorem that

$$T(n) = O(n^{\log_2 3}) = O(n^{1.59}) \,.$$

This is the first and significant improvement over the elementary school algorithm.

Polynomial Multiplication

Given two degree n polynomials, $A(x) = \sum_{i=0}^{n} a_i x^i$ and $B(x) = \sum_{i=0}^{n} b_i x^i$. We can use the same idea to compute $A \cdot B(x)$ in $O(n^{1.59})$ word operations, where we assume that a_ib_j can be computed in $O(1)$ word operations.

Matrix multiplication

We have already known $O(n^3)$ *word* operations algorithm to multiply two $n \times n$ matrices. If we use divide and conquer, namely divide a matrix into 4 blocks, and get 8 subproblems. Then we still get $O(n^3)$ complexity because we haven't done anything clever.

See Strassen's algorithm for $O(n^{2.81})$ complexity. After that, $O(n^{2.37})$ was achieved. Some researchers believe that it can be done in $O(n^2)$ word operations. This is currently of theoretical interest only, as the algorithms are too complicated to be implemented. Strassen's algorithm can be implemented and it will be faster than the standard algorithm when $n \gtrsim 5000$.

There are many combinatorial problems that can be reduced to matrix multiplication, and Strassen's result implies that they can be solved faster than $O(n^3)$ time. As an example, the problem determining whether a graph has a triangle can be reduced to matrix multiplication, and we leave it as a puzzle to you to figure out how. There are many combinatorial problems in the literature where the fastest algorithm is by matrix multiplication.

Faster Fourier Transform

This is a very nice algorithm to solve integer multiplication and polynomial multiplication in $O(n \log n)$ time. See [DPV 2.6] or CS 371.

3

Graph Algorithms

We study simple graph algorithms based on graph searching. There are two most common search methods: breadth first search (BFS) and depth first search (DFS).

3.1 Graphs Basics

Many problems in computer science can be modeled as graph problems.

3.1.1 Graph Representations

Let $G = (V, E)$ be an undirected graph. We use throughout that $n = |V|$ and $m = |E|$. There are two standard representations of a graph. One is adjacency matrix: an $n \times n$ matrix A with

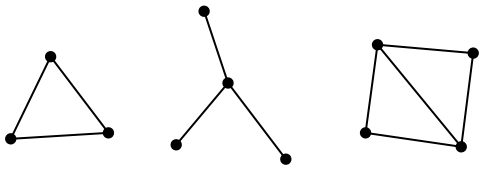
$$A[i, j] = \begin{cases} 1 & \text{if } ij \in E \\ 0 & \text{if } ij \notin E \end{cases}.$$

Another is the adjacency list, where each vertex maintains a linked list of its neighbors.

We will mostly use the adjacency list representation, as its space usage depends on the number of edges, while we need to use $\Theta(n^2)$ space to store an adjacency matrix. Only the adjacency list representation allows us to design algorithms with $O(m + n)$ word operations.

3.1.2 Graph Connectivity

Given a graph, we say two vertices are connected if there is a path from u to v. A subset of vertices $S \subseteq V$ is connected if $u, v \in S$ are connected for all $u, v \in S$. A graph is connected if $s, t \in V$ are connected for all $s, t \in V$. A connected component is a maximally connected subset of vertices.



3 components

Some of the most basic question about a graph are:

1. to determine whether it is connected.

2. to find all the connected components.
3. to determine whether u, v are connected for given $u, v \in V$.
4. to output a shortest path between u and v for given $u, v \in V$.

Breadth first search (BFS) can be used to answer all these questions in $O(n+m)$ time.

3.2 Breadth First Search

To motivate breadth first search, imagine we are searching for a person in a social network. A natural strategy is to ask our friends, and then ask our friends to ask their friends, and so on. A basic version of BFS is described as follows.

Algorithm 6: Breadth First Search (basic version)

Input: $G = (V, E)$, $s \in V$
Output: all vertices reachable from s

```
1  visited[v] = False for all v ∈ V
2  Q ← ∅ // queue Q
3  enqueue(Q, s)
4  visited[s] = True
5  while Q ≠ ∅ do
6      u ← dequeue(Q)
7      foreach neighbor v of u do
8          if visited[v] = False then
9              enqueue(Q, v)
10             visited[v] = True
```

Each vertex is enqueued at most once (when $\text{visited}[v] = \text{False}$). When a vertex is dequeued, the for loop is executed for $\deg(v)$ iterations. So, the total time complexity is

$$O\left(n + \sum_{v \in V} \deg(v)\right) = O(n+m)\,.$$

Lemma

There is a path from s to v if and only if $\text{visited}[v] = \text{True}$ at the end.

Proof:
Skipped. □

The correctness of BFS is supported by the lemma. With this claim, we see that this basic version of BFS can already be used to answer:

- whether the graph is connected or not, by checking $\text{visited}[v] = \text{True}$ for all $v \in V$.
- the connected component containing s, by returning all the vertices with $\text{visited}[v] = \text{True}$.
- whether there is a path from s to v, by checking whether $\text{visited}[v] = \text{True}$.

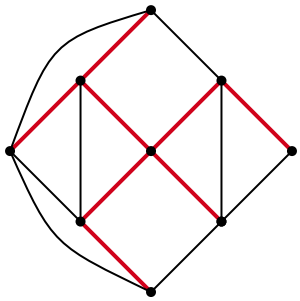
Then we can find all connected components of the graph in $O(n+m)$ time.

3.2.1 BFS Tree

How to trace back a path from s to v (if such a path exists)? This follows from the proof of the lemma above, although not presented here. We can add an array $\mathsf{parent}[v]$. When a vertex v is first visited, within the for loop of vertex u, then we set $\mathsf{parent}[v] = u$. Now, to trace out a path from v to s, we just need to write a for loop that starts from v, and keep going to its parent until we reach vertex s.

For all vertices reachable from s, the edges $(v, \mathsf{parent}[v])$ from a tree, called the **BFS tree**/ Why is it a tree in the connected component containing s? Say the connected component has n vertices. Every vertex has one edge to its parent. These edges can't form a cycle because the parent of a vertex is visited earlier. So these edges form an acyclic subgraph and there are $n-1$ edges (as s has no parent). Therefore, the edges $(v, \mathsf{parent}[v])$ must form a tree in the component containing s.

An example of BFS tree is shown below:



3.2.2 Shortest Paths

Not only can we trace back a path from v to s using a BFS tree, this path is indeed a shortest path from s to v!

To see this, let's think about how a BFS tree was created. These edges record the first edges to visited a vertex. Initially, s is the only vertex in the queue, and then every neighbor of s is visited within s being their parent, and these edges are put in the BFS tree. At this time, all vertices with distance one from s are visited and are put in the queue, before all other vertices with distance at least two from s are put in the queue.

A vertex v is said to have distance k from s if the shortest path length from s to v is k.

Then, all vertices with distance one will be dequeued, and then all vertices with distance two will be enqueued before all other vertices with distance at least three.

Repeating this argument inductively will show that all the shortest path distances from s are computed correctly, and a shortest path can be traced back from the BFS tree.

This is also very intuitive (friends before friends etc). Being able to compute the shortest paths from s

is the main feature of BFS. We summarize below the BFS algorithm with shortest path included.

Algorithm 7: Breadth First Search (with shortest path distances)

Input: $G = (V, E)$, $s \in V$
Output: all vertices reachable from s and their shortest path distance from s

```
1  visited[v] = False for all v ∈ V.
2  Q ← ∅ // queue Q
3  enqueue(Q, s)
4  visited[s] = True
5  distance[s] = 0
6  while Q ≠ ∅ do
7      u ← dequeue(Q)
8      foreach neighbor v of u do
9          if visited[v] = False then
10             enqueue(Q, v)
11             visited[v] = True
12             parent[v] = u
13             distance[v] = distance[u] + 1
```

3.2.3 Bipartite Graphs

One application of BFS is to check whether a graph is bipartite or not. There is not much freedom allowed to design an algorithm for checking bipartiteness. Given a vertex s, all its neighbors must be on the other side, and then neighbors of neighbors must be on the same side as s, and so on. With this observation, we can run the BFS algorithm above and put all vertices with even distance fro ms on the same side as s and all other vertices on the other side.

Algorithm 8: Check bipartiteness using BFS

```
1  L := {v ∈ V | dist(s, v) is even}
2  R := {v ∈ V | dist(s, v) is odd}
3  if ∃e = uv s.t. u, v ∈ L or u, v ∈ R then
4      return "non-bipartite"
5  else
6      return "bipartite" and (L, R) as the bipartition
```

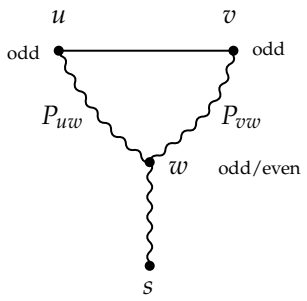
Note that we assume the graph is connected, as otherwise we can solve the problem in each component. The time complexity is $O(m + n)$ as we just do a BFS and then check every edge once.

Correctness

It is clear that when the algorithm says "bipartite" it is correct, as (L, R) is indeed a bipartition. The more interesting part is to show that when the algorithm say "non-bipartite", it is also correct. When can we say for sure that a graph is non-bipartite?

An iff condition is when the graph has an odd cycle from MATH 249. Thus we would like to show when the graph says "non-bipartite", the graph has an odd cycle.

Suppose WLOG that there is an edge uv between two vertices $u, v \in L$. We look at the BFS tree T.



Let w be the lowest common ancestor of u, v in T. Since $\mathsf{dist}(s,u)$ and $\mathsf{dist}(s,v)$ are both odd (i.e., having the same parity), regardless of whether $\mathsf{dist}(s,w)$ is even or odd, the same of the path lengths of uw and vw on T is an even number. This implies $P_{vw} \cup P_{uw} \cup \{uv\}$ is an odd cycle.

This provides an algorithmic proof that a graph is bipartite iff it has no odd cycles. This also provides a linear time algorithm to find an odd cycle of an undirected graph. Having an odd cycle is a "short proof" of non-bipartiteness, which is much better than saying "we tried all bipartitions but all failed".

3.3 Depth First Search

Depth first search is another basic search method in graphs, and this will be useful in identifying more refined connectivity structures.

Motivating Example

Last time we imagined that we would like to search for a person in a social network, and BFS is a very natural strategy (asking friends, then friends of friends, and so on). There are other situations that using depth first search is more natural. Imagine that we are in a maze search for the exit. We could model this problem as a $s-t$ connectivity problem in graphs. Each square of the maze is a vertex, and two other vertices have an edge if and only if the two squares are reachable in one step. Then finding a path from our current position to the exit is equivalent to finding a path between two specified vertices in a graph (or determine that none exists).

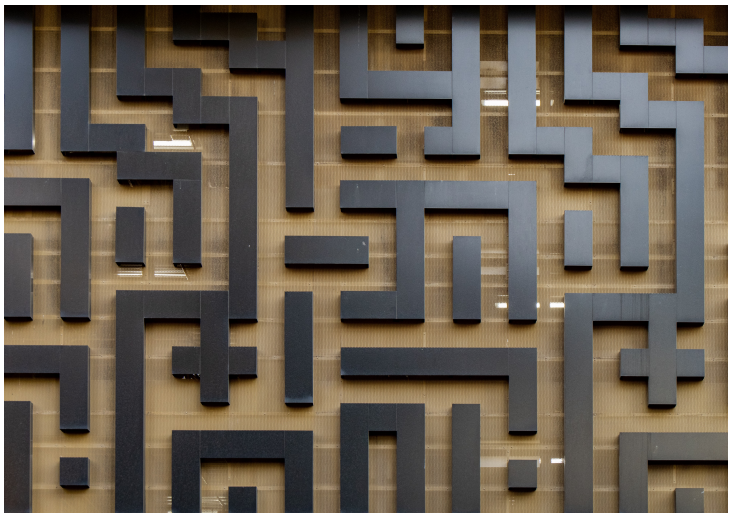


Photo by Mitchell Luo on Unsplash

How would we search for a path in the maze? There are no friends to ask, and it doesn't look efficient anymore to explore all vertices with distance one, then distance two and so on (as we have to move back and forth).

Assuming we have a chalk and can make marks on the ground. Then it is more natural to keep going

on one path bravely until we hit a dead end, and make some marks on the way and also on the way back so that we won't come back to this dead end again, and only explore yet unexpected places. This is essentially depths first search (DFS).

As for BFS, we define DFS by an algorithm. DFS is most naturally defined as a recursive algorithm.

Algorithm 9: Depth First Search

Input: an undirected graph $G = (V, E)$, a vertex $s \in V$
Output: all vertices reachable from s

```
visited[v] = False for all v ∈ V
visited[s] = True
explore(s)

Function explore(u):
    foreach neighbor v of u do
        if visited[v] = False then
            visited[v] = True
            explore(v)
```

Time Complexity The analysis of the time complexity is similar to that in BFS. For each vertex u, the recursive function `explore`(u) is called at most once. When `explore`(u) is called, the for loop is executed at most $\deg(u)$ times. Thus the total time complexity is $O\big(n + \sum_{v \in V} \deg(v)\big) = O(n + m)$ word operations.

There is a way to write DFS non-recursively using the idea of stack.

The basic lemma about graph connectivity still holds for DFS.

Lemma

There is a path from s to t if and only if visited$[t]$ = True at the end.

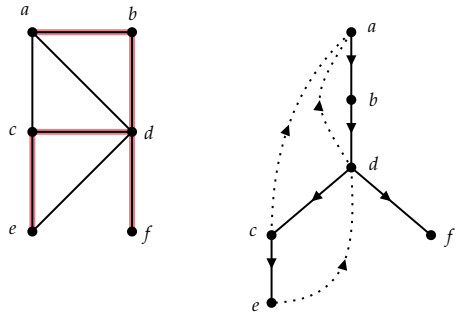
The lemma shows that DFS can also be used to check $s - t$ connectivity, to find the connected component containing s, and to check graph connectivity, all in $O(m + n)$ time. As we can also find all connected components in $O(m + n)$ time using DFS.

The main difference from BFS is that DFS cannot be used to compute the shortest path distances, and this is the main feature of BFS. But as we shall see, DFS can solve some interesting problems that BFS cannot solve.

3.3.1 DFS Tree

As for BFS, we can construct a DFS tree to trace out the path from s. Again, when a vertex v is first visited when we explore vertex u, we say vertex u is the parent of vertex v.

By the same argument as in BFS, these edges $(v, \text{parent}[v])$ form a tree, and we can use them to find a path to s. We call this a DFS tree of the graph. Note that a graph could have many different DFS trees depending on the order of exploring the neighbors of vertices. The same can be said for BFS trees.



Definitions/Terminology for DFS trees

- The starting vertex s is regarded as the **root** of the DFS tree.
- A vertex u is called the **parent** of a vertex v if the edge uv is in the DFS tree, and u is closer to the root than b is to the root.
- A vertex u is called an **ancestor** of a vertex b if u is closer to the root than v, and u is on the path from v to the root. In this situation, we also say v is a descendant of vertex u.
- A non-tree edge uv is called a **back edge** if either u is an ancestor or descendant of v. It is called a back edge because this edge is from the descendant to the ancestor.

In the above example, b is an ancestor of e and f, but c is neither an ancestor nor descendant of f. The following example is simple but has an important property that we will use.

Property (back edges)

In an undirected graph, all non-tree edges are back edges.

Proof:
Suppose by contradiction that there is an edge between u and v but u and v are not an ancestor-descendant pair. WLOG assume that u is visited before v. Then, since $uv \in E$, v will be explored before u is finished, and thus u will be an ancestor of v, a contradiction. □

Starting Time and Finishing Time

We record the time when a vertex is first visited and the time when its exploring is finished. These information will be very useful in design adn analysis of algorithms. To be precise, we include the

pseudocode in the following.

Algorithm 10: Depth First Search (with timer)

Input: an undirected graph $G = (V, E)$, a vertex $s \in V$
Output: all vertices reachable from s

```
1  visited[v] = False for all v ∈ V
2  time = 1
3  visited[s] = True
4  explore(s)
5
6  Function explore(u):
7      start[u] = time
8      time ← time + 1
9      foreach neighbor v of u do
10         if visited[v] = False then
11             visited[v] = True
12             explore(v)
13     finish[u] = time
14     time ← time + 1
```

Property (parenthesis)

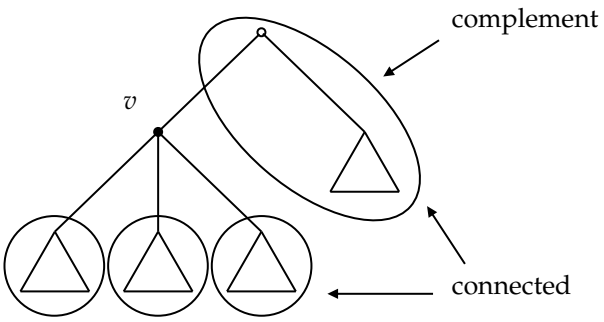
The intervals $[\text{start}(u), \text{finish}(u)]$ and $[\text{start}(v), \text{finish}(v)]$ for two vertices u and v are either disjoint or one is contained in another.

The latter case happens when u, v are an ancestor-descendant pair.

3.4 Cut Vertices and Cut Edges

Suppose an undirected graph is connected. We would like to identify vertices and edges that are critical in the graph connectedness. A vertex v is a **cut vertex** (aka an articulation point, or a separating vertex) if $G - v$ is not connected, i.e., removal of v and its incident edges disconnects the graph. An edge e is a **cut edge** (aka a bridge) if $G - e$ is not connected. See examples in CO 342.

The idea is to use a DFS tree to identify all cut vertices and cut edges. Consider a vertex v which is not the root. We would like to determine whether v is a cut vertex. When we look at the DFS tree, all the subtrees below v are connected, as well as the complement of the subtree at v.



The main observation is the property that all the non-tre edges are back edges (proved above), and so the only way for a subtree below v to be connected outside is to have edges going to an ancestor of v.

Claim A subtree T_i below v is a connected component in $G - v$ if and only if there are no edges with one endpoint in T_i and another endpoint in a (strict) ancestor of v.

Proof:

$\Leftarrow$ By the back edge property, all the non-tree edges are back edges, so there are no edges going to another subtree below v nor edges going to another subtree of the root. So, if there are no such edges going to a (strict) ancestor of v, then T_i must be a component in $G - v$.

$\Rightarrow$ On the other hand, if such edges exist, then T_i is connected to the complement even after v is removed, and so T_i won't be a connected component in $G - v$. □

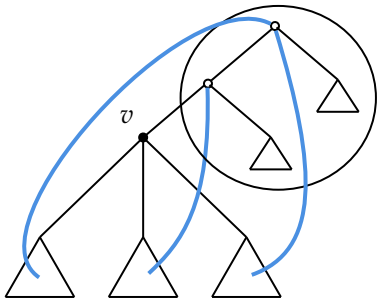
The same argument applies to each subtree below v gives the following characterization of a cut vertex.

Lemma

For a non-root vertex v in a DFS tree, v is a cut vertex if and only if there is a subtree below v with no edges going to an ancestor of v.

Proof:

$\Rightarrow$ If every subtree below v has some edges going to an ancestor of v, then every subtree is connected to the complement.



So, $G - v$ is connected and thus v is not a cut vertex.

$\Leftarrow$ If some subtree T_i below v has no edges going to an ancestor of v, then T_i will be a connected component in $G - v$ by the previous claim, and thus v is a cut vertex. □

It remains to consider the root vertex of the DFS tree.

Lemma

For the root vertex v of a DFS tree, v is a cut vertex if and only if v has at least two children.

With these lemmas, we then know how to determine if a vertex a cut vertex by looking at a DFS tree.

We are ready to use the above lemmas to design a $O(n + m)$ time algorithm to report all cut vertices. To have an efficient implementation, the idea is to process the vertices of a DFS following a bottom up ordering, and keep track of how "far up" the back edges of a subtree can go (i.e., how close to the root). By the lemma, for a non-root vertex v, v, v is not a cut vertex if and only if all subtrees below v have an edge that goes above v.

What would be a good parameter to keep track of how far up we can go? The starting time would be a good measure, because an ancestor always has an earlier/smaller staring time than its descendants. (We could also do it in other ways, e.g., by recording the distance to the root instead.)

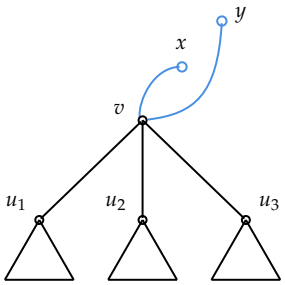
Let us define a value $\mathsf{low}[v]$ for each vertex on the DFS tree:

$$\mathsf{low}[v] := \min\left\{\mathsf{start}[v], \quad \min\left\{\mathsf{start}[w] \;\middle|\; \begin{array}{l} uw \text{ is a back edge with } u \text{ being} \\ \text{a descendant of } v \text{ or } u = v \end{array} \right\}\right\}$$

Informally, $\mathsf{low}[v]$ records how far up we can go from the subtree rooted at v. We will be done if we can prove the following two things:

1. We can compute $\mathsf{low}[v]$ for all $v \in V$ in $O(n+m)$ time.
2. We can identify all cut vertices in $O(n+m)$ time using the low array.

For 1, we compute the low values from the leaves of the DFS tree to the root of the DFS tree. The base case is when v is a left. Then we can compute $\mathsf{low}[v]$ by considering all the edges incident on v and taking the minimum of the starting time of the other endpoint. This takes $O(\deg(v))$ time. By induction, suppose the low values of all children of v are computed. Then to compute $\mathsf{low}[v]$, we just need to take the minimum of the low value of its children, as well as the start time for all back edges involving v. This takes $O(\deg(v))$ time.



In the example given in the picture, $\mathsf{low}[v] = \min\{\mathsf{low}[u_1], \mathsf{low}[u_2], \mathsf{low}[u_3], \mathsf{start}[x], \mathsf{start}[y]\}$. It should be clear that $\mathsf{low}[v]$ is computed correctly, assuming the low values of all its children are correct, and so the correctness can be established by induction. By this bottom-up ordering, every vertex on the tree is only processed once, and thus the total time complexity is $O(n + \sum_{v \in V} \deg(v)) = O(n+m)$.

For 2, to check whether a non-root vertex v is a cut vertex, we just need to check whether $\mathsf{low}[u_i] < \mathsf{start}[v]$ for all children u_i of v. If so, then v is not a cut vertex, as all subtrees rooted at u_i will be a connected component in $G - v$, and thus v is a cut vertex. These arguments are covered in the first lemma. The root vertex is handled using the other lemma.

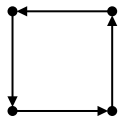
This completes the description of a linear time algorithm to identify all cut vertices given the low array.

3.5 Directed Graphs

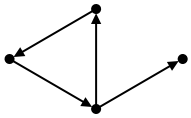
In a directed graph, each edge has direction. When we say uv is a directed edge, we mean the edge is point from u to v, and u is called the **tail** and v is called the **head** of the edge. Given a vertex v, $\text{indeg}(v)$ denotes the number of directed edges with v as the head and we call them incoming edges to v. Similarly, $\text{outdeg}(v)$ denotes the number of directed edges with directed edges with v as the tail and we call them outgoing edges of v.

Directed graphs are useful in modeling asymmetric relations (e.g., web page links, one-way streets, etc). We are interested in studying the connectivity properties of a directed graph.

We say t is **reachable** from s if there is a directed path from s to t. A directed graph is called **strongly connected** if for every pair of vertices $u, v \in V$, u is reachable from v and v is reachable from u.



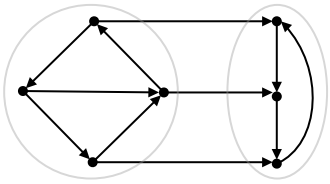
strongly connected



not strongly connected

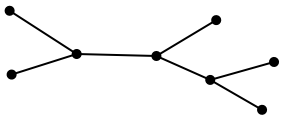
A subset $S \subseteq V$ is called strongly connected if for every pair of vertices $u, v \in S$, u is reachable from v and v is reachable from u.

A subset $S \subseteq V$ is called a **strongly connected component** if S is a maximally strongly connected subset, i.e., S is strongly connected but $S + v$ is not strongly connected for any $v \in S$.

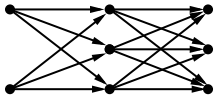


2 strongly connected components

A directed graph is a **directed acyclic graph** (DAG) if there are no directed cycles in it. Note that a directed acyclic graph, unlike its undirected counterpart, could have many edges.



undirected acyclic graph



directed acyclic graph

We are interested in designing algorithms to answer the following basic questions:

1. Is a given graph strongly connected?
2. Is a given graph directed acyclic?
3. Find all strongly connected components of a given directed graph.

As in undirected graphs, it will turn out that there are $O(n + m)$-time algorithms to solve these problems, but they are not as easy as the algorithms for undirected graphs.

3.5.1 Graph Representations

Both adjacency matrix and adjacency list can be defined for directed graphs. In the adjacency matrix A, if ij is a directed edge, then $A_{ij} = 1$; otherwise $A_{ij} = 0$. In the adjacency list, if ij is an edge, then j is on i's linked list. As in undirected graphs, we will only use adjacency list in this part of the course, as only this allows us to design $O(n + m)$-time algorithms.

3.5.2 Reachability

Before studying the above questions, we first study a simpler question of checking reachability. Given a directed graph and vertex s, both DFS and BFS can be used to find all vertices reachable from s in $O(n + m)$ time. Both BFS and DFS are defined as in for undirected graphs, except that we only explore

out-neighbors.

Algorithm 11: Depth First Search for directed graphs

Input: a directed graph $G = (V, E)$, a vertex $s \in V$
Output: all vertices reachable from s

```
1  visited[v] = False for all v ∈ V
2  time = 1
3  visited[s] = True
4  explore(s)
5
6  Function explore(u):
7      start[u] = time
8      time ← time + 1
9      foreach out-neighbor v of u do
10         if visited[v] = False then
11             visited[v] = True
12             explore(v)
13     finish[u] = time
14     time ← time + 1
```

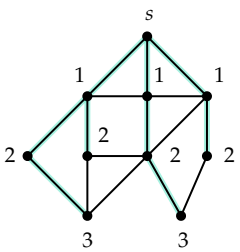
The time complexity is $O(n + m)$, and a vertex t is reachable from s if and only if visited$[t]$ = True. When we look at all vertices reachable from s, the subset form a "directed cut" with no outgoing edges (but could have incoming edges into the subset). We can define BFS for directed graphs analogously, by only exploring out-neighbors. An important property of BFS is that it computes the shortest path distances from s to all other vertices.

3.5.3 BFS/DFS Trees

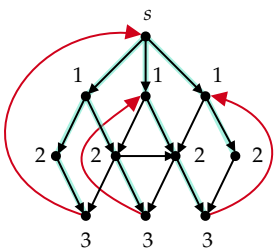
As in for undirected graphs, when a vertex v is first visited, we remember its parent as the vertex u when v is first visited from. The edges $(v, \text{parent}[v])$ form a tree, and both BFS trees and DFS trees are defined in this way.

BFS trees

By setting $\text{dist}[v] = \text{dist}[\text{parent}[v]] + 1$, we compute all shortest path distances from s. In undirected graphs, for all non-tree edges uv, $\text{dist}[v] - 1 \leq \text{dist}[u] \leq \text{dist}[v] + 1$. In directed graphs, there could be non-tree edges with large difference between $\text{dist}[u]$ and $\text{dist}[v]$, but in this case, it must be $\text{dist}[u] > \text{dist}[v]$ as they must be "backward edges".



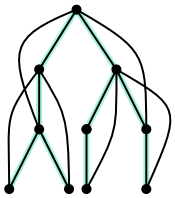
undirected BFS tree



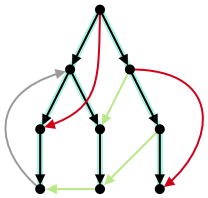
directed BFS tree

DFS trees

In undirected graphs, all non-tree edges are back edges. In directed graphs, some non-tree edges are "cross edges" and "forward edges".



undirected DFS tree



directed DFS tree

Structured in directed graphs are more complicated.

3.5.4 Strongly Connected Graphs

We are ready to consider the first problem of checking whether a directed graph is strongly connected. From the definition, we need to check $\Omega(n^2)$ pairs and see if there is a directed path between them. In undirected graphs, it is enough to pick an arbitrary vertex s, and check whether all vertices are reachable from s. So we just check reachability for $O(n)$ pairs.

What would be a corresponding "succinct" condition to check in directed graphs? It is easy to find examples for which just checking reachability from s is not enough. Checking reachability from every vertex would work, but it would take $\Omega(n(n+m))$ time, too slow. The following observation allows us to reduce the number of pairs to check to $O(n)$.

Observation G is strongly connected if and only if every vertex v is reachable from s and s is reachable from every vertex v, where s is an arbitrary vertex.

Proof:
$\Rightarrow$ is trivial by the definition for strong connectedness.

Now let's prove $\Leftarrow$. For any u, v, by combining a path from u to s and a path from s to v, we obtain a path from u to v, so G is strongly connected. □

We know how to check whether all vertices are reachable from s in $O(n+m)$ time by BFS or DFS. How do we check whether s is reachable from all vertices efficiently? There is simple trick to do it, by reversing the direction of the edges.

Claim Given G, we reverse the direction of all the edges to obtain G^R. There is a directed path from v to s in G if and only if there is a directed path from s to v in G^R. So, s is reachable from all vertices in G if and only if every vertex is reachable from s in G^R.

With this claim, we can check whether s is reachable from every vertex in G by doing one BFS/DFS in G^R from s.

To summarize, we have the following algorithm.

Algorithm 12: Strong Connectivity

1. Check whether all vertices in G are reachable from s by one BFS/DFS.
2. Reverse the direction of all edges in G to obtain G^R.
3. Check whether all vertices in G^R are reachable from s by one BFS/DFS.
4. If both yes, return "strongly connected"; otherwise return "not strongly connected".

The correctness of the algorithm follows from the observation and the claim above. The time complexity is $O(n+m)$ time. Note that we can construct G^R in linear time.

3.5.5 Direct Acyclic Graphs

Direct acyclic graphs are directed graphs without directed cycles. They are useful in modeling dependency relations (e.g., course prerequisites, software installation). In such situations, it would be useful to find an ordering of the vertices so that all the edges go forward. This is called a **topological ordering** of the vertices (e.g., an ordering to take the courses).

Proposition

A directed graph is acyclic if and only if there is a topological ordering of the vertices.

Proof:

$\Leftarrow$ Since all the directed edges go forward, there are no directed cycles.

$\Rightarrow$ We will prove that any directed acyclic graph has a vertex v of indegree zero. Since $G - v$ is also acyclic, there is a topological ordering of $G - v$ by induction on the number of vertices, and we are done.

It remains to argue that every directed acyclic graph has a vertex of zero indegree.

Suppose by contradiction that every vertex has indegree at least one. Then we start from an arbitrary vertex u, and go to an in-neighbor u_1 of u, and then go to an in-neighbor u_2 of u, and so on. It is always possible since every vertex has in-degree at least one. If some in-neighbor repeats, then we find a directed cycle, a contradiction. But it must repeat at some point, since the graph is finite. □

There are at least two approaches to find a topological ordering of a directed acyclic graph efficiently.

Approach 1 Keep finding a vertex of indegree zero in the remaining graph and put it in the beginning of the ordering. The algorithm is in $O(n + m)$ time.

Approach 2 This is perhaps less intuitive, but the ideas will be useful later. The idea is to do a DFS on the whole graph (i.e., start a DFS on an arbitrary vertex, but if not all vertices are visited, start a DFS on an unvisited vertex, and so on, until all vertices are visited, just like what we would do in finding all connected components of an undirected graph). Note that this DFS can be done in any ordering of vertices. In particular, we don't need to start at a vertex of indegree zero nor do we need any information about a topological ordering.

In the following proof, we use that the parenthesis property of starting and finishing time holds for directed graphs as well.

Lemma

If G is directed acyclic, then for any directed edge uv, $\mathsf{finish}[v] < \mathsf{finish}[u]$ for any DFS.

Proof:

We consider two cases.

Case 1 $\mathsf{start}[v] < \mathsf{start}[u]$. Since the graph is acyclic, u is not reachable from v. So u cannot be a descendant of v. By the parenthesis property, the intervals $[\mathsf{start}[v], \mathsf{finish}[v]]$ and $[\mathsf{start}[u], \mathsf{finish}[u]]$ must be disjoint. The only possibility left is $\mathsf{start}[v] < \mathsf{finish}[v] < \mathsf{start}[u] < \mathsf{finish}[u]$, proving the lemma in this case.

Case 2 $\mathsf{start}[u] < \mathsf{start}[v]$. Then, since v is unvisited when is started and uv is an edge, v will be a descendant of u in the DFS tree. This is the same as the argument used in the back edge property.

By the parenthesis property, we have $\mathsf{start}[u] < \mathsf{start}[v] < \mathsf{finish}[v] < \mathsf{finish}[u]$. □

Algorithm 13: Topological ordering / directed acyclic graphs

1 Run DFS on the whole graph.
2 Output the ordering with decreasing finishing time.
3 Check if it is a topological ordering. If not, return "not acyclic".

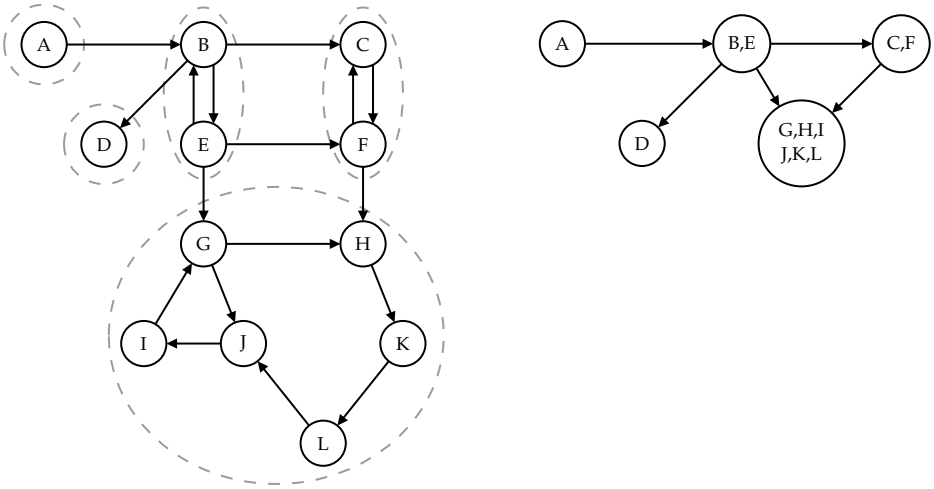
Correctness The lemma proves that if the graph is acyclic, then all edges go forward in this ordering. On the other hand, if the graph is not acyclic, then there is no topological ordering by the proposition.

Time Complexity The algorithm can be implemented in $O(n+m)$ time. Note that we don't need to do sorting for the second step. Just put a vertex in a queue when it is finished, by adding one line in the code.

3.5.6 Strongly Connected Components (SCC)

Finally, we consider the more difficult problem of finding all strongly connected components. We wil combine and extend the previous ideas to obtain an $O(n+m)$ time algorithm. First, let's get a good idea about how a general directed graph looks like.

Observation Two strongly connected components are vertex disjoint. If two strongly connected components C_1 and C_2 share a vertex, then $C_1 \cup C_2$ is also strongly connected, contradicting maximality of C_1, C_2.



In the picture (adapted from [DPV 3.4]), when every strongly connected component is "contracted" into a single vertex, then the resulting directed graph is acyclic. This is true in general. So, a general directed graph is a directed acyclic graph on its strongly connected components.

Idea 1 Suppose we start a DFS/BFS in a "sink component" C (a component with no outgoing edges), then we can identify the strongly connected component C. This is because every vertex in C is reachable from the starting vertex, but no vertices outside. So, just read off vertices with $\mathsf{visited}[v] = \mathsf{True}$ will identify C.

This suggests the following strategy.

1. FInd a vertex v in a sink component C.
2. Do a DFS/BFS to identify C.

3. Remove C from the graph and repeat.

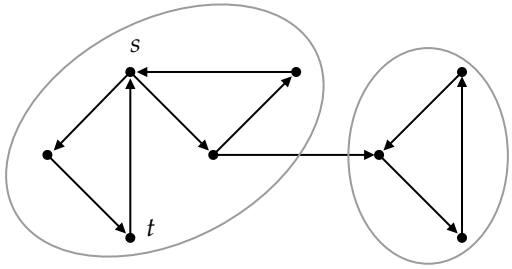
So, now, the question is how to find a vertex in a sink component efficiently? It doesn't look easy.

Idea 2 Do "topological sort". As discussed above, if each strong component is "contracted" to a single vertex, the resulting graph is acyclic. From the previous section about directed acyclic graphs, we know that if we do a DFS on the whole graph, the node with the earliest finishing time is a sink.

This suggests the following strategy.

1. Run DFS on the whole graph and obtain an ordering in increasing finishing time.
2. Use this ordering in the previous strategy in idea 1 to take out one sink component at a time.

This is a very nice strategy, but unfortunately it doesn't work. Consider a counterexample



If we start the DFS at s, then node t has the earliest finishing time, but t is not a sink component.

Idea 3 The natural strategy doesn't work, but a modification of it may still work. The observation si that the DFS ordering still gives us useful information about a topological ordering of components. In particular, although we couldn't say that a vertex with smallest finishing time is in a sink component, we can say that a vertex with the largest finishing time is in a source component. The proof of the following lemma is similar to the proof of the lemma in topological ordering.

Lemma

If C and C' are strongly components and there are edges from C to C', then the largest finishing time in C is bigger than the largest finishing time in C'

Proof:
Again, we consider two cases.

Case 1 The first vertex v visited in $C \cup C'$ is in C'. Note that vertices in C are not reachable from v but all vertices in C' are reachable from v. By the time when v is finished, all vertices in C' are finished, while all vertices in C haven't started.

Case 2 The first vertex v visited in $C \cup C'$ is in C. Since vertices in $C \cup C'$ are reachable from v, all vertices in $C \cup C'$ will be finished before v is finished, and so $v \in C$ will have the largest finishing time in $C \cup C'$. □

With this lemma, we know that if we first do a DFS adn order the vertices in decreasing order finishing time, and then do a DFS again using this ordering, then we will visit "ancestor components" before we visited "descendant components". But this not what we want, as we want to start in a sink component and cut it out first.

Idea 4 Reverse the graph so that sources become sinks!

First observe that the strong components in G are the same as the strong components in G^R. Very

important for us, source components in G become sink components in G^R and vice versa. Therefore, the ordering we have in G following a topological ordering of the components from sources to sinks becomes an ordering in G^R following a topological ordering of the components from sinks to sources. Now, we can just follow this ordering to do the DFS in G^R to cut out sink components one at a time, as we wished in idea 1 and idea 2.

Finally, we can summarize the algorithm.

Algorithm 14: Strong Components

1. Run DFS on the whole graph G using an arbitrary ordering of vertices.
2. Order the vertices in decreasing order of finishing times obtained in step 1.
3. Reverse the graph G to obtain the graph G^R.
4. Follow the ordering in step 2 to explore the graph G^R to cut out the components one at a time.

To be more precise, we expand step 4 in more details.

Algorithm 15: Strong Components (step 4)

```
1 Let i be the vertex of i-th largest finishing time in step 2.
2 Let c = 1 // It is a variable counting the number of strong connected components.
3 for 1 ≤ i ≤ n do
4     if visited[i] = False then
5         DFS(G^R, i)
6         Mark all the vertices reachable from i in G^R in this iteration to be in component C.
7         c ← c + 1
```

The proof of correctness follows from the discussion above. All the steps can be implemented in $O(n+m)$ time.

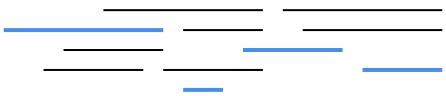
4

Greedy Algorithms

4.1 Interval Scheduling

Input: n intervals $[s_1, f_1]$, $[s_2, f_2]$, $\ldots$, $[s_n, f_n]$.
Output: a maximum set of disjoint intervals.

For example, in the picture below,



the highlighted intervals form a maximum set of disjoint intervals. There are multiple optimal solutions in this example.

For this problem, we can imagine that we have a room, and there are people who like to book our room and they tell us the time interval that they need the room, and our objective is to choose a maximum subset of activities with no time conflicts.

Generally speaking, greedy algorithms work by using simple and/or local rules to make decisions and commit on them. An analogy is to make maximum profit in short term.

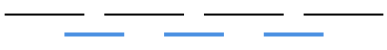
There are multiple natural greedy strategies for this problem including

- earliest starting time (choose the interval with $\min_i s_i$),
- earliest finishing time (choose the interval with $\min_i f_i$),
- shortest interval (choose the interval with $\min_i f_i - s_i$),
- minimum conflicts (choose the interval that overlaps with the minimum number of other intervals).

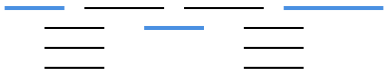
It turns out that only one of these strategies would work.

Earliest starting time is the easiest to find counterexamples, because the interval with earliest starting time could be very long, e.g.,

It is not difficult to find counterexamples for shortest intervals, e.g.,



It's a bit harder to find counterexamples for minimum conflicts, e.g.,



There are no counterexamples for earliest finishing time. The intuition is that we leave maximum space for future intervals. But how could we argue that this will always give an optimal solution (that there are no solutions better)?

Algorithm 16: Interval Scheduling

```
Sort the intervals so that f_1 ≤ f_2 ≤ ··· ≤ f_n. Current solution S = ∅.
for 1 ≤ i ≤ n do
    if interval i [s_i, f_i] has no conflicts with other intervals in S then
        S ← S ∪ {i}
return S
```

Correctness The idea is to argue that any (optimal) solution would do no worse by using the interval with earliest finishing time. Let $[s_{i_1}, f_{i_1}] < [s_{i_2}, f_{i_2}] < \cdots < [s_{i_k}, f_{i_k}]$ be the solution returned by the greedy algorithm. Let $[s_{j_1}, f_{j_1}] < [s_{j_2}, f_{j_2}] < \cdots < [s_{j_\ell}, f_{j_\ell}]$ be an optimal solution with $\ell \geq k$. Since $f_{i_1} \leq f_{j_1} < s_{j_2}$ (the first inequality is because $i_1 = 1$ and is the interval with earliest finishing time, and the second inequality is because $[s_{j_1}, f_{j_1}]$ and $[s_{j_2}, f_{j_2}]$ are disjoint), so $[s_{j_1}, f_{j_1}] < [s_{j_2}, f_{j_2}] < \cdots < [s_{j_\ell}, f_{j_\ell}]$ is still an optimal solution. Thus we have the following claim, showing that it is no worse by choosing $[s_1, f_1]$.

Claim There exists an optimal solution with $[s_1, f_1]$ chosen.

We will use this argument inductively to prove the following lemma.

Lemma

$[s_{i_1}, f_{i_1}], [s_{i_2}, f_{i_2}], \ldots, [s_{i_k}, f_{i_k}], [s_{j_{k+1}}, f_{j_{k+1}}], \ldots, [s_{j_\ell}, f_{j_\ell}]$ is an optimal solution with $f_{i_k} \leq f_{j_k}$.

Informally, the lemma says that the greedy solution always "stays ahead". Before we prove the lemma, let's see how it implies that the greedy solution is optimal. Suppose, by contradiction, that the greedy solution is not optimal, i.e., $\ell > k$. Since $f_{i_k} \leq f_{j_k} < s_{j_{k+1}}$, we see that the interval $[s_{j_{k+1}}, f_{j_{k+1}}]$ has no overlapping with the greedy solution, and it should have been added to the greedy solution by the greedy algorithm, a contradiction that the greedy algorithms only finds k disjoint intervals.

Proof of lemma:
The base case holds because of the previous claim. Assume the claim is true for $c \geq 1$, and we are to prove the inductive step. Since $[s_{i_1}, f_{i_1}], \ldots, [s_{i_c}, f_{i_c}], [s_{j_{c+1}}, f_{j_{c+1}}], \ldots, [s_{j_\ell}, f_{j_\ell}]$ is an optimal solution by the induction hypothesis with $f_{i_c} \leq f_{j_c}$, we have $f_{i_{c+1}} \leq f_{j_{c+1}}$ as the interval $[s_{j_{c+1}}, f_{j_{c+1}}]$ has no overlapping with $[s_1, f_1], \ldots, [s_{i_c}, f_{i_c}]$ and the greedy algorithm chooses the next interval with earliest finishing time. As $f_{i_{c+1}} \leq f_{j_{c+1}}$, by replacing $[s_{j_{c+1}}, f_{j_{c+1}}]$ with $[s_{i_{c+1}}, f_{i_{c+1}}]$, the resulting solution $[s_{i_1}, f_{i_1}], \ldots, [s_{i_{c+1}}, f_{i_{c+1}}], [s_{j_{c+2}}, f_{j_{c+2}}], \ldots, [s_{j_\ell}, f_{j_\ell}]$ is feasible and optimal. □

Time complexity $O(n \log n)$.

4.2 Interval Coloring

Input: n intervals $[s_1, f_1], [s_2, f_2], \ldots, [s_n, f_n]$
Output: use the minimum number of colors to color the intervals, so that each interval gets one color

and two overlapping intervals get two different colors.

We can imagine this is the problem of using the minimum number of rooms (colors) to schedule all the activities (intervals).

Our natural greedy algorithm is to use the previous algorithm to choose a maximum subset of disjoint intervals and use only one color for them, and repeat. However, this algorithm won't work. There is another greedy algorithm that will work.

Algorithm 17: Interval coloring

```
1 Sort the intervals by starting time so that s_1 < s_2 < ··· < s_n.
2 for 1 ≤ i ≤ n do
3     use the minimum available color c_i to color the interval i. // i.e., use the minimum
        number to color the interval i so that it doesn't conflict with the colors
        of the intervals are already colored.
```

Suppose the algorithm uses k colors. To prove the correctness of the algorithm, we must prove that there are no other ways to color the intervals using at most $k-1$ colors. How do we argue that?

A nice and simple way is to show that when the algorithm uses k colors, there exists a time t such that it is contained in k intervals. Since these k intervals are pair wise overlapping. We need at least k colors just to color them, and therefore a $(k-1)$ coloring doesn't exist.

Proof of correctness:
Suppose the algorithm uses k colors. Let interval ℓ be the first interval to use color k. This implies that interval ℓ overlaps with intervals with colors $1, \ldots, k-1$. Call then $[s_{i_1}, f_{i_1}], \ldots, [s_{i_{k-1}}, f_{i_{k-1}}]$. As we sort the intervals with increasing order of starting time, we have $s_{i_j} \leq s_\ell$ for all $1 \leq j \leq k-1$. Since all these intervals overlap with $[s_\ell, f_\ell]$, we also have $s_\ell \leq f_{i_j}$ for all $1 \leq j \leq k-1$. Therefore, s_ℓ is a time contained in k intervals. This implies that there is no $k-1$ coloring. □

4.3 Minimizing Total Completion Time

Input: n jobs, each requiring processing time p_i.
Output: An ordering of the jobs to finish so as to minimize the total completion time.

The completion time of a job is defined as the time when it is finished. For example, given four jobs with processing times $3, 4, 6, 7$, and if we process the jobs in this order, then the completion times are $3, 7, 13, 20$, and the total completion time is 43.

It is very intuitive that we should process the jobs in increasing order of processing time. (Imagine we are in a supermarket with four customers with $3, 4, 6, 7$ items.) But how do we are argue that there are no better solutions? Here we use an "exchange argument" to show that the greedy solution is optimal (Again, we imagine that we have 1 item but the customer in front of us has 10 items.)

Proof of correctness:
Consider a solution which is not sorted by non-decreasing processing time. This implies there is an "inversion pair": the i-th job and the j-th job with $i < j$ but $p_i > p_j$. This implies that there exists $i \leq k < j$ with $p_k > p_{k+1}$. By swapping the jobs k and $k+1$, we will prove that the total completion time is smaller. Note that all the completion times, except for k and $k+1$, are unchanged.

Let c be the completion time of job $k-1$. Before swapping, the completion times of job k and $k+1$ are $c + p_k$ and $c + p_k + p_{k+1}$ respectively, and the total of these two jobs are $2c + 2p_k + p_{k+1}$. After

swapping, the total of these two jobs are $2c + 2p_{k+1} + p_k$. Since $p_{k+1} < p_k$, it is clear that the solution after swapping is better. So, any ordering which is not sorted by non-decreasing processing times is not optimal. □

The exchange argument is quite nice and useful.

4.4 Huffman Coding

This is a well-known greedy algorithm that gives an optimal prefix code. The idea might has been covered CS 240 without motivation.

Compression

Suppose a text has 26 letters $a, b, c, \ldots, z$. A standard way to represent the letters using bits is to use $\lceil \log_2 26 \rceil = 5$ bits for each letter, e.g., $a = 00000$, $b = 00001$, $c = 00010$, $\ldots$, $z = 11010$. So we use five bits to represent each letter. In general, we cannot do much better if each letter appears equally likely.

What if we scan the text once and notice that the letters appear with quite different frequencies? E.g., "a" appears 10% of the time. Can we hope to do better by using variable-length encoding scheme? The idea is to use fewer bits for more frequent letters, and more bits for less frequent letters, so that the average number of bits used is fewer.

Prefix Coding

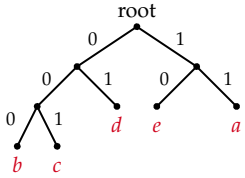
When we use fixed-length bit strings to represent the letters, it is easy to decode. Say if we use five bits to encode the 26 letters, we just need to read five bits at a time, to decode one letter at a time. It is not as clear how to decode if we use variable-length encoding.

Say suppose we encode the five letters as five letters as $a = 01$, $b = 001$, $c = 011$, $d = 110$, $e = 10$. Then, when we read a compressed text such as 00101110, it could be decode as "*bce*", but it could also be decoded as "*bad*". To avoid ambiguity in decoding, we will construct **prefix codes**, so that no encoded string is a prefix of another encoded string.

In the above example, the ambiguity arises because the string representing "a" is a prefix of the string representing "c", i.e., $a = 01$ and $c = 011$. Now, suppose we use a prefix code for the five letters, $a = 11$, $b = 000$, $c = 001$, $d = 01$, $e = 10$. Then we encode the text "*cabed*" using the string 001110001001. When the decoder reads this string, it will read from left to right and unambiguously decode the text "*cabed*". In short, prefix coding allows easy and efficiently encoding decoding.

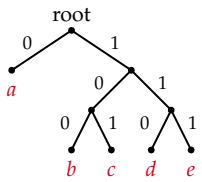
Decoding Tree

It is useful to represent a prefix code as a binary tree. In the example above, the prefix code can be represented by the tree as shown below.



To decode, we start from the root, read a bit, move to the corresponding branch, until we reach a leaf, then we return the letter associated to the leaf, and go back to the root and repeat.

As another example, $a = 0$, $b = 0$, $c = 101$, $d = 110$, $e = 111$, then the tree is



This is useful when frequency of "a" is very high.

It should be clear that each prefix code has a binary tree representation and each binary tree representation corresponds to a (unique) prefix code.

Objective

Suppose we are given the frequencies of the five letters, say $f_a = 0.8$, $f_b = f_c = f_d = f_e = 0.05$. Then the average length of a letter (in the second example) is

$$\underbrace{0.8 \times 1}_{\text{1 bit for "}a\text{"}} + \underbrace{0.05 \times 3}_{\text{3 bits for "}b\text{"}} + 0.05 \times 3 + 0.05 \times 3 + 0.05 \times 3 = 1.4$$

In the three representation, the length of an encoded string is equal to the depth of the corresponding leaf, and the objective becomes $\sum_i f_i \cdot \text{depth}_T(i)$.

Optimal Prefix Code

Input: n symbols with frequencies $f_1, \ldots, f_n$ so that $\sum_{i=1}^{n} f_i = 1$.
Output: a binary tree T with n leaves that minimizes $\sum_i f_i \cdot \text{depth}_T(i)$

This problem doesn't look so easy as the output space is quite complicated, as there are exponentially many possible binary trees. It is also not clear how the algorithm can make a decision greedily.

Let's first think about how an optimal solution should look like. We say a binary tree is **full** if every internal node has two children. We start with a simple observation.

Observation The binary tree of an optimal solution is full.

Proof:
If there is an internal node with only one child, then we can directly connect its child to its parent and decrease the depth of some leaves, getting a better solution. □

Corollary There are at least two leaves of maximum depth that are siblings (having the same parent).

Proof:
Look at a leaf of maximum depth. If it has no sibling, then the tree is not full. □

Suppose we know the shape of an optimal binary tree (which we don't know yet). Then it is not difficult to figure out how to assign symbols to the leaves. We should assign symbols with highest frequencies to the leaves with smallest depth, and assign symbols with lowest frequencies to the leaves with largest depth. Otherwise, if one symbol of higher frequency is of a larger depth than another symbol of lower frequency, then we could "exchange" the two symbols and decrease the objective value. This exchange argument leads to the following observation.

Observation There is an optimal solution in which the two symbols with lowest frequencies are assigned to leaves of maximum depth, and furthermore they are siblings.

Proof:
By the corollary, there are two leaves of maximum depth that are siblings. By the exchange argument, we can assign them with the symbols with lowest frequencies without increasing the objective value (just exchange with the symbols there. So, the new solution is still optimal and satisfies the properties as stated. □

Huffman's Algorithm

So far, we have deduced very little information about the optimal solutions. We just know that there are two leaves of maximum depth that are siblings, adn we can assign two symbols with lowest frequencies there. but we still don't know how to use the frequencies to make decisions.

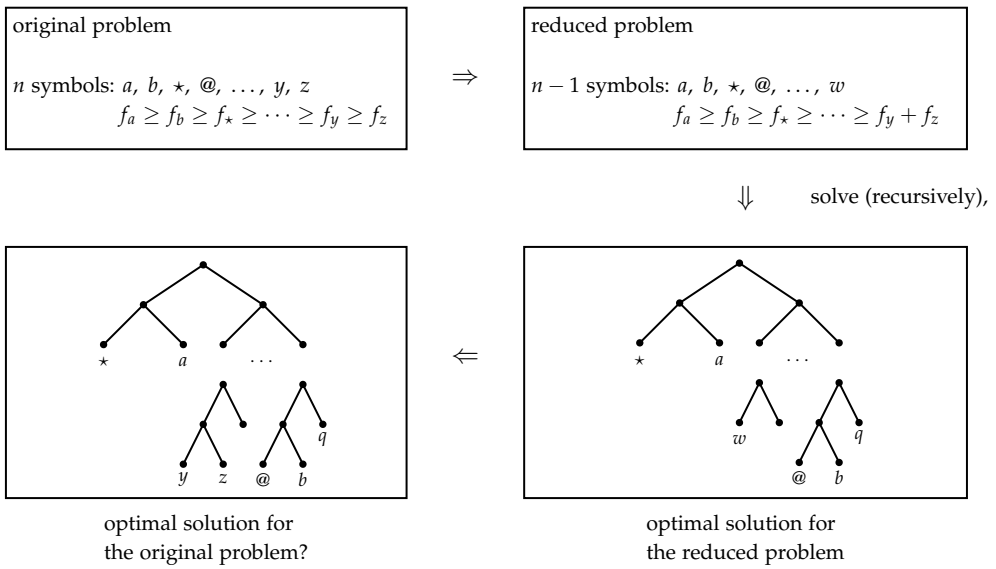
Huffman figured out that this little information is already enough to design an efficient algorithm. The idea is to "reduce" the problem size by one, by identifying/combing two symbols with lowest frequencies into one, knowing that they can be assumed to be siblings of maximum depth. How the tree should look like will become apparent when the problem size becomes small enough, and then we can construct back a bigger tree from a smaller tree one step at a time.

Algorithm 18: Huffman Code

Input: a set S of n symbols, with frequencies $f_1, \ldots, f_n$
Output: an optimal binary tree T, with leaves associated to symbols in S

if $|S| = 2$ **then**
  encode one symbol using 0 and another symbol using 1, and return the tree. `// Base case`
else
  Let y and z be two symbols with lowest frequencies, denoted by f_y and f_z.
  `// Induction step`
  Delete symbols t and z from S. Add a new symbol w with frequency $f_y + f_z$.
  Solve this new problem (with $n-1$ symbols) recursively and get an optimal tree T'.
  In T', look at the leaf associated with w, add two leaves to it (so that w becomes an internal node), and associate y and z with the two new leaves.

The scheme is summarized as follows.



Let's do some examples to be familiar with Huffman's algorithm.

Example:

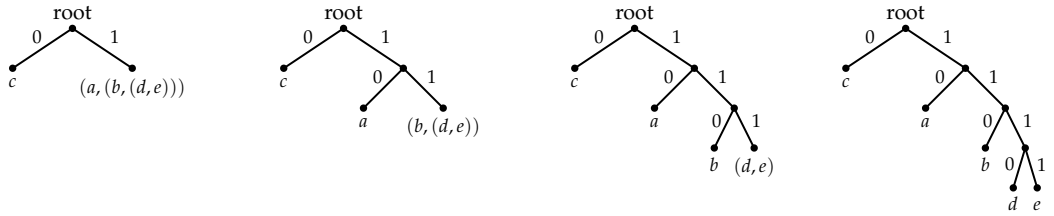
Five symbols: a, b, c, d, e, with $f_a = 0.3$, $f_b = 0.2$, $f_c = 0.4$, $f_d = 0.05$, $f_e = 0.05$

Reduce to four symbols: a, b, c, (d,e) with $f_a = 0.3$, $f_b = 0.2$, $f_c = 0.4$, $f_{(d,e)} = 0.1$

Reduce to three symbols: a, $(b,(d,e))$, c with $f_a = 0.3$, $f_{(b,(d,e))} = 0.3$, $f_c = 0.4$

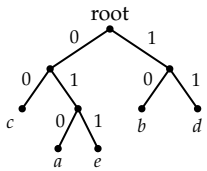
Reduce to two symbols: $(a,(b,(d,e)))$, c with $f_{(a,(b,(d,e)))} = 0.6$, $f_c = 0.4$

Now construct trees.



Example:

Five symbols: a, b, c, d, e, with $f_a = 0.18$, $f_b = 0.24$, $f_c = 0.26$, $f_d = 0.2$, $f_e = 0.12$. Then the tree is



Correctness proof The most natural way to analyze a recursive algorithm is to use induction. We will use the same terminology as described in the algorithm.

Clearly, the output of the base case when there are only two symbols is optimal. Denote the objective value of the solution returned by Hoffman's algorithm by $\mathsf{Sol}(n)$. By induction, we have computed correctly an optimal binary tree T' for the $n-1$ symbols, with y and z replaced by w, and $f_w = f_y + f_z$. Let $\mathsf{Obj}(T')$ be the objective value of T'.

What is the relationship between $\mathsf{Sol}(n)$ and $\mathsf{Obj}(T')$? We just add two leaves to T' to form T. Recall that the objective is $\sum_i f_i \cdot \mathrm{depth}_T(i)$. Every other leaf has the same contribution to the objective value of T and T'. So, just focus on the change of deleting w and then adding y and z back. Then

$$\begin{aligned}\mathsf{Sol}(n) &= \mathsf{Obj}(T') - f_w \cdot \mathrm{depth}_{T'}(w) + f_y \cdot (\mathrm{depth}_{T'}(w)+1) + f_z \cdot (\mathrm{depth}_{T'}(w)+1) \\ &= \mathsf{Obj}(T') + f_y + f_z\,,\end{aligned}$$

since $f_w = f_y + f_z$ by our construction.

Next, we would like to argue that any optimal solution must have objective value at least $\mathsf{Obj}(T') + f_y + f_z$, and this would imply that the solution returned by Hoffman's algorithm is optimal.

Let T^* be an optimal solution for the original problem. By the lemma using the exchange argument, we can assume that y and z are leaves of maximum depth in T^* and furthermore they are siblings. We define $T^{*\prime}$ as obtained from T^* by deleting y and z and define $f_w = f_y + f_z$ where w is the parent of y and z.

By the same calculation above, $\mathsf{Obj}(T^*) = \mathsf{Obj}(T^{*\prime}) + f_y + f_z$. Now, observe that $T^{*\prime}$ is a solution to the reduced problem of size $n-1$. By the induction hypothesis, T' is an optimal solution to the reduced problem, and hence it must hold that $\mathsf{Obj}(T') \le \mathsf{Obj}(T^{*\prime})$. Putting together, we have

$$\mathsf{Obj}(T^*) = \mathsf{Obj}(T^{*\prime}) + f_y + f_z \ge \mathsf{Obj}(T') + f_y + f_z = \mathsf{Obj}(T) = \mathsf{Sol}(n)\,,$$

proving that T is an optimal solution.

Implementation

In every iteration, we need to find two symbols with lowest frequencies, delete them and add a new symbol with the frequency as their sum. A straightforward Implementation takes $\Theta(n)$ time to find two symbols with lowest frequencies.

We can use a heap to do these operations in $O(\log n)$ time. Recall that a heap support the operations of `insert` and `extract-min` in $O(\log n)$ time. So, each iteration can be implemented in $O(n \log n)$ time, after initially inserting all n frequencies in the heap in $O(n \log n)$ time. Therefore, the total time complexity is $O(n \log n)$.

4.5 Single source shortest paths

This section is typically covered in graph algorithm part of other offerings, as they cover graph algorithm later than greedy algorithm.

Shortest Paths

Input: A directed graph $G = (V, E)$ with a non-negative length ℓ_e for each edge $e \in E$, and two vertices $s, t \in V$.
Output: A shortest path from s to t, where the length of a path is equal to the sum of edge lengths.

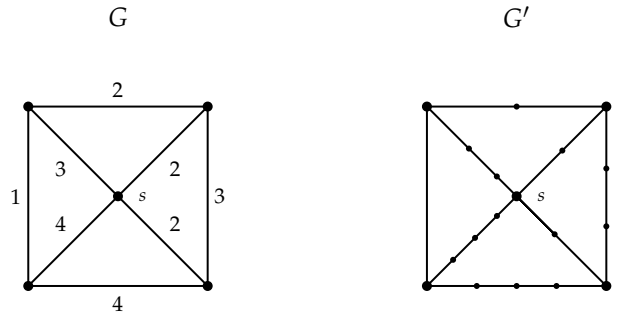
We can think of the graph as a road network, and the edge length represents the time needed to drive pass the road (may depend on traffic). Then the problem is to find a fastest way to drive from point s to point t, which is the type of queries that Google Maps answers every day. It will be more convenient to solve a more general problem.

Input: A directed graph $G = (V, E)$ with a non-negative length ℓ_e for each edge $e \in E$ and a vertex $s \in V$.
Output: A shortest path from s to v, for every vertex $v \in V$.

The problem seems to be harder, because each path could have $\Omega(n)$ edges, and so the output size could already be $\Omega(n^2)$. But it will turn out that there is a succinct representation of these paths and this single source shortest path problem can be solved in the same time complexity as the shortest s-t path problem.

Breadth First Search and Dijkstra's Algorithm

We have seen previously that BFS can be used to solve the single-source shortest paths problem, when every edge has the same length (so we count the number of edges on the paths). It is not difficult to reduce the non-negative edge length problem to this same-length special case. Let's say all edge lengths are positive integers. We can reduce our problem to the special case by replacing each edge of length ℓ_e by a path of ℓ_e edges.

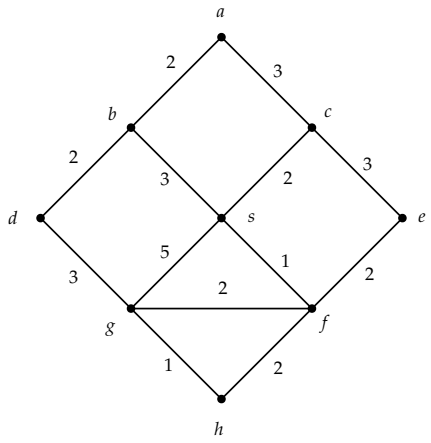


Then there is an s-t path of length k in G if and only if there is an s-t path of length k in G'. And then we can solve the problem in G' by simply doing BFS starting from s. The only problem of this reduction is that it may not be efficient as G' may have many more vertices. The number of vertices in G' is $n + \sum_{e \in E}(\ell - 1)$, so when ℓ_e is large then G' has many more vertices, and then a linear time algorithm for G' does not correspond to a linear time algorithm for G.

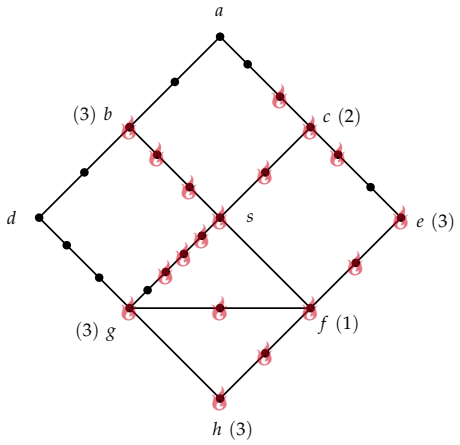
Physical Process

To design an efficient algorithm through this reduction, instead of constructing G' explicitly and run BFS on it, we just keep G' in mind and try to simulate BFS on G' efficiently. We can think of the process of doing BFS on G' as follows.

- We start a fire at vertex s at time 0. The fire wil spread out.
- It takes one unit of time to burn an edge in G'.



G



G' at time $t = 3$

To simulate this process efficiently, the idea is that we just need to be able to keep finding out what is the next vertex to be burned and when (rather than simulating it faithfully on the paths). Initially, it is the source vertex s. Then, knowing that s is burned at time 0, in the example above, we know that b, c, g, f will be burned in *at most* time $3, 2, 5, 1$ respectively. Then, the next vertex to be burned is vertex f. Knowing that f is burned at time 1, we know that g, h, e will all be burned in *at most* 2 times units (by time step 3) through the edges fg, fh, fe.

In the above example, we update the *upper bound* on g to be burned to be 3 (replacing the initial value of 5 which was set when s was burned), as vertex g will be burned earlier through the fire from g than the fire from s. Then, with this updated information after g is burned, we find out the next vertex to be burned (in this example c) and then update the upper bound on the neighbors of c as the fire can now go from c to its neighbors. Repeating this (i.e., find out next vertex, update upper bound on

neighbors) gives us an efficient simulation of the process of G', and this exactly Dijkstra's algorithm.

Algorithm 19: Dijkstra's Algorithm (or BFS simulation)

```
1  dist(v) = ∞ for every v ∈ V // the initial upper bound on the time v is burned
2  dist(s) = 0 // start a fire at vertex s at time 0
3  Q = make-priority-queue(V) // all vertices are put in the priority queue, using
       dist(v) as the key value (priority queue) of v
4  while Q is not empty do
5      u = delete-min(Q) // dequeue the vertex with minimum dist() value, i.e., find
           out the next vertex to be burned
6      foreach (out-)neighbor v of u do // it works for directed graph as well
7          if dist(u) + ℓ_uv < dist(v) then
8              dist(v) = dist(u) + ℓ_uv
9              decrease-key(Q, v)
10             parent(v) = u // note that this may be updated multiple times
```

Dijkstra's algorithm is very similar to that of the BFS algorithm, except that we use a priority queue (which can be implemented by a min-heap) to replace a queue.

Correctness It's clear that the algorithm is an efficient simulation of BFS in G'. Also we argued before that shortest paths from s in G' are shortest paths from s in G. So, the correctness of this algorithm in G just follows from the correctness of using BFS to solve shortest paths in G' (where every edge is of the same length), which we proved before. Anyway, we will do a more traditional and formal proof as well.

Time complexity Each vertex is enqueued once (in the beginning) and dequeued once (when it is burned). When a vertex is dequeued, we check every edge uv once and may use the value $\mathsf{dist}(u) + \ell_{uv}$ to update the value of $\mathsf{dist}(v)$ using a `decrease-key` operation. All the `enqueue`, `dequeue`, `decrease-key` operations can be implemented in $O(\log n)$ time by a min-heap, and thus the total time complexity is $O\Big((n + \sum_v \mathrm{outdeg}(v)) \log n\Big) = O((n+m)\log n)$ time. So, the cost of simulating BFs in the big graph is just an extra factor of $\log n$. With more advanced data structures (namely Fibonacci heaps), the runtime could be improved to $O(n \log n + m)$.

Here we present a more traditional approach to prove the correctness of Dijkstra's algorithm. Besides being more formal, the proof technique is also useful in analyzing other problems, e.g., MST. The algorithm will turn out to be the same, but the way of thinking about it is slightly different. This is also the way we think of Dijkstra's algorithm as a greedy algorithm.

The idea is to grow a subset $R \subseteq V$ so that $\mathsf{dist}(v)$ for $v \in R$ are computed correctly. Initially, $R = \{s\}$, and then at each iteration we add one more vertex to R. Which vertex to be added in an iteration? We

will add the vertex which is closest to R, so in this sense we are growing R greedily.

Algorithm 20: Dijkstra's algorithm (set version)

```
dist(v) = ∞ for all v ∈ V
dist(s) = 0
R = ∅
while R ≠ V do
    pick the vertex u ∉ R with smallest dist(u)
    R ← R ∪ {u}
    foreach edge uv ∈ E do
        if dist(u) + ℓ_uv < dist(v) then
            dist(v) = dist(u) + ℓ_uv
```

Equivalently, $u \notin R$ is the vertex with $\mathsf{dist}(u) = \min_{v \notin R, w \in R}\{\mathsf{dist}(w) + \ell_{wv}\}$. Then it's easy to see that these two versions of Dijkstra's algorithm are equivalent.

We prove the correctness by induction, maintaining the following invariant.

Invariant: For any $v \in R$, $\mathsf{dist}(v)$ is the shortest path distance from s to v.

Base case: It is true when $R = \{s\}$.

Induction step: Assume that the invariant holds for the current R. We would like to prove that the invariant remains true after a new vertex u is added to R. We need to argue that $\mathsf{dist}(u) = \min_{v \notin R, w \in R}\{\mathsf{dist}(w) + \ell_{wv}\}$ is the shortest path distance from s to u.

Let $w \in R$ be the vertex in a minimizer, i.e., $\mathsf{dist}[u] = \mathsf{dist}[w] + \ell_{wv} \leq \mathsf{dist}[a] + \ell_{ab} \quad \forall a \in R$ and $b \notin R$. Since $\mathsf{dist}[w]$ is the shortest path distance from s to w, the shortest path distance from s to u is at most $\mathsf{dist}[w] + \ell_{wu}$ (i.e., using the path from s to w and edge wu). So it remains to prove that the shortest path distance from s to u is at least $\mathsf{dist}[w] + \ell_{wu}$.

Consider any path P from s to u. Since $s \in R$ and $u \notin R$, there must be an edge xy in P such that $x \in R$ and $y \notin R$. (Note that x could be w and/or y could be u.) The length of path P is at least $\mathsf{dist}[x] + \ell_{xy}$, as $\mathsf{dist}[x]$ is computed correctly by the invariant.

Here observe that we crucially use the fact that the length of each edge is non-negative. But we know that $\mathsf{dist}[w] + \ell_{wu} \leq \mathsf{dist}[x] + \ell_{xy}$ as (w, u) is a minimizer. Therefore, we conclude that

$$\mathsf{length}(P) \geq \mathsf{dist}(x) + \ell_{xy} \geq \mathsf{dist}[w] + \ell_{wu} = \mathsf{dist}[u]\,.$$

This inequality is true for any path from s to u, and thus the shortest path distance from s to u is at least $\mathsf{dist}[u]$.

Note that the proof applies to directed graphs as is.

Shortest Path Tree

Now, we think about how to store the shortest paths from s for all $v \in V$. From the proof, when a vertex v is added to R, any vertex u that minimizes $\mathsf{dist}[u] + \ell_{uv}$ is parent on a shortest path from s to v. We keep track of this parent information throughout Dijkstra's algorithm, by always keeping a vertex that attains the minimum of $\mathsf{dist}[u] + \ell_{uv}$ for $u \in R$ as the current parent, and will update when new vertex is put in R and makes this value smaller. After the algorithm finished (i.e., $R = V$), by tracing the parents until we reach the source s, we can find a shortest path from s to v. Every vertex can do the same to find a shortest path from s.

As in BFS tree, the edges $(v, \mathsf{parent}[v])$ form a tree. So, we have a succinct way to store all the shortest

paths from s, using only $n - 1$ edges to store n paths. This shortest path tree is very useful.

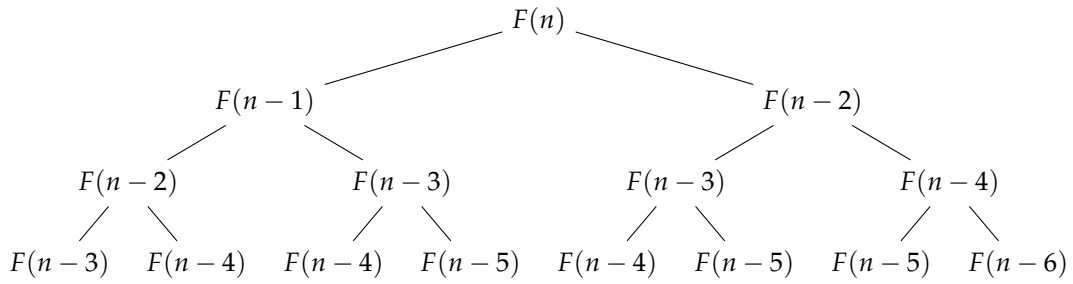
Clearly, the algorithm will fail if there are some negative edge lengths. We will come back to this more difficult setting when we study dynamic programming algorithms.

5

Dynamic Programming

On a high level, we can solve a problem problem by dynamic programming if there is a recurrence relation with only a small number of subproblems (i.e., polynomially many). This is a general and powerful technique, and also simple to use once we learnt it well.

To illustrate the idea using a toy example, consider the problem of computing the Fibonacci sequence $F(n) = F(n-1) + F(n-2); F(1) = F(2) = 1$. The function is defined recursively with the base cases given. It is then natural to compute it using recursion, but if we trace the recursion tree, we find out that it is huge.



If we solve the recurrence relation (MATH 249), then we find out the runtime of this algorithm is $\Theta(1.618^n)$. Observe that the recursion tree is highly redundant, many subproblems are computed over and over again. There are only n subproblems. Why do we waste so much time? There are two approaches to solve the problem efficiently.

Top-Down Memorization

As in BFS/DFS, we use an array visited$[i]$ to ensure that we only compute each subproblem at most once, and we use an array answer$[i]$ to store the value $F(i)$ for future lookup.

Algorithm 21: Top-Down Memorization

```
1  Function Main(n):
2      visited[i] = False for 1 ≤ i ≤ n
3      F(n)
4
5  Function F(i):
6      if visited[i] = True then
7          return answer[i]
8      if i = 1 or i = 2 then
9          return 1
10     answer[i] = F(i-1) + F(i-2)
11     visited[i] = True
12     return answer[i]
```

Time Complexity Each subproblem is only computed once, when visited$[i]$ = False. When it is computed, it looks up two values. So, the total time complexity is $O(n)$. Alternatively, we can think of a graph search on a directed acyclic graph, with only n vertices and $2(n-1)$ edges.

Bottom-up Computation

For Fibonacci sequence, there is a straightforward algorithm to solve the problem in $O(n)$ time.

Algorithm 22: Bottom-Up Computation

```
1  F(1) = F(2) = 1
2  for 3 ≤ i ≤ n do
3      F(i) = F(i − 1) + F(i − 2)
```

It is clear that this solves the problem in $O(n)$ additions.

Note that the values grow exponentially in n, so we cannot assume that each addition can be done in $O(1)$ time.

Dynamic Programming

So, basically, this is the framework of dynamic programming, to store the intermediate values so that we don't need to compute it again. From the top-down approach, it should be clear that if we write a recursion with only polynomially many subproblems with additional polynomial time processing, then the problem can be solved in polynomial time. In this viewpoint, designing an efficient dynamic programming algorithm amounts to coming up with a nice recursion. This is similar to what we have done in designing divide and conquer algorithms, coming up with a right recursion. We will see that many interesting and seemingly difficult problems have a nice recurrence relation. Of course, it requires some skills and practices to write a nice recursion to solve the problems, and this is what we will focus on in many examples to follow.

In practice, the bottom-up implementation is preferred as it is non-recursive and usually more efficient. In some cases, it will be easy to translate a top-down solution into a bottom-up solution. In some cases,

however, it requires clear thinking to find a correct ordering to compute the subproblems, especially when the recurrence relation is complicated, although in principle we just need a topological ordering. Our main focus will be to come up with the right recurrence, as that would already imply an efficient algorithm. We will also mention the bottom-up implementations as much as possible.

5.1 Weighted Interval Scheduling

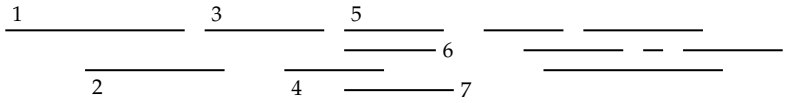
Input: n intervals $[s_1, f_1]$, $[s_2, f_2]$, $\ldots$, $[s_n, f_n]$ with weights $w_1, w_2, \ldots, w_n \in \mathbb{R}$.
Output: a subset S of disjoint intervals that maximizes $\sum_{i \in S} w_i$.

This is a generalization of the interval scheduling problem previously, when $w_i = 1$ for all i. We can think of the intervals are requests to book our room from time s_i to f_i, and the i-th request is willing to pay w_i dollars if the request is accepted. Then, our objective is to maximize our income, while ensuring that there are no conflicts for the accepted requests.

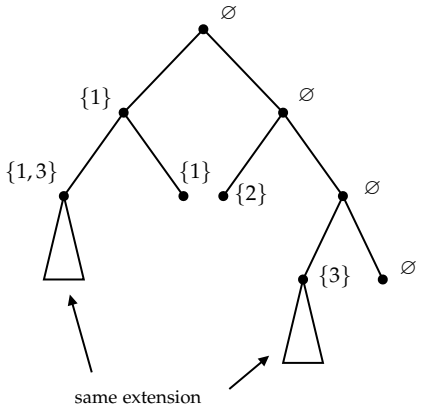
Unlike the special case when $w_i = 1$ for all i, there are no known greedy algorithms for this problem.

Exhaustive Search

To come up with a good recursion for this problem, we start by running an exhaustive search algorithm and see why it is wasteful and how to improve it.



First, we make a decision on interval 1, either choose it or not. If we choose interval 1, then we cannot choose interval 2, and we need to make a decision on interval 3. If we do not choose interval 1, then we need to make decision on interval 2. Doing this recursively, we have a recursion tree as shown.



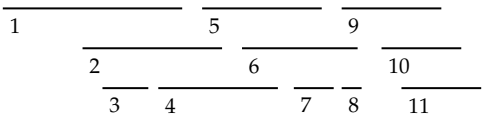
This is an exponential time algorithm, but observe a lot of redundancy. For example, in the branches $\{1,3\}$ and $\{3\}$, the subtrees extending these partial solutions are exactly the same. This is because to determine how to extend these partial solutions, what really matter is the last interval of the current partial solutions, but not anything on the left, since they won't interact with anything on the right.

So, we just need to keep track of the "boundary" of the solution. In this problem, the boundary is simply the last interval. This suggests that there should be a recursion with only one parameter!

Better Recurrence

To facilitate the algorithm description, we use a good ordering of the intervals and pre-compute useful information. We sort the intervals by starting time so that $s_1 \leq s_2 \leq \cdots \leq s_n$. For each interval i, we

use next$[i]$ to denote the smallest j such that $j > i$ and $f_i < s_j$, i.e., the first interval on the right of interval i that is not overlapping with interval i. If no such intervals exist, next$[i]$ is defined as $n + 1$, representing the end. In the example below,



next$[1] = 5$, next$[2] = 6$, next$[3] = 4$, next$[9] = 12$, etc.

Since we sort the intervals by non-decreasing starting time, for an interval i, next$[i]$ has the property that intervals $\{i, \ldots, \text{next}[i] - 1\}$ overlap with interval i while the intervals $\{\text{next}[i], \ldots, n\}$ are disjoint from interval i.

Now, we are ready to write a recursion with only one parameter, resulting in only n subproblems! Let $\texttt{opt}(i) :=$ maximum income that we can earn using the intervals in $\{i, i+1, \ldots, n\}$ only. Then $\texttt{opt}(1)$ is the optimal value that we would like to compute. To compute $\texttt{opt}(1)$, there are only two options for the solutions:

1. The solutions that choose interval 1.

   For these solutions, we earn w_1 dollars by choosing interval 1. But then we cannot choose the intervals in $\{2, \ldots, \text{next}[1] - 1\}$ since these intervals overlap with interval 1. So, to find the optimal value of choosing interval 1, we need to find an optimal way to choose from $\{\text{next}[1], \ldots, n\}$. Therefore, the optimal value for solutions choosing interval 1 is $w_1 + \texttt{opt}(\text{next}[1])$.

2. The solutions that don't choose interval 1.

   Then, by definition of $\texttt{opt}(2)$, the optimal value for solutions not choosing interval 1 is $\texttt{opt}(2)$.

Combining the two cases, we get that $\texttt{opt}(1) = \max\{w_1 + \texttt{opt}(\text{next}[1]), \texttt{opt}(2)\}$. This recurrence relation is true for every i, and so we have the following recursive algorithm to solve the problem.

Algorithm 23: top-down weighted interval scheduling

```
1  Sort the intervals by non-decreasing starting time so that s_1 ≤ s_2 ≤ ··· ≤ s_n.
2  Compute next[i] for 1 ≤ i ≤ n. Set visited[i] = False for 1 ≤ i ≤ n.
3  return opt(1)
4
5  Function opt(i):
6      if i = n + 1 then return 0
7      if visited[i] = True then return answer[i]
8      answer[i] = max{w_i + opt(next[i]), opt(i + 1)}
9      visited[i] = True
10     return answer[i]
```

Correctness The correctness of the algorithm follows from the explanation of the recurrence relation above.

Time complexity Sorting and the next[] array can be computed in $O(n \log n)$ time. After that, the top-down memorization implements the recursion in $O(n)$ time, since there are only n subproblems

and each subproblem only needs to look up two values.

Algorithm 24: Bottom-up implementation for weighted interval scheduling

```
1 opt(n + 1) = 0
2 for i ← n downto 1 do
3     opt(i) = max{w_i + opt(next[i]), opt(i)}
```

Quite surprisingly, this has the same time complexity as the greedy algorithm! That is, somehow we can implement an exhaustive search algorithm as efficient as the greedy algorithm! We can see why dynamic programming is general and powerful, because it is very systematic (so that we may not need problem specific insight) and yet we get very competitive algorithms.

5.2 Subset-Sum and Knapsack Problem

We consider two related and useful problems.

Subset-Sum

Input: n positive integers and $a_1, a_2, \ldots, a_n$, and an integer K.
Output: a subset $S \subseteq [n]$ with $\sum_{i \in S} a_i = K$, or report that no such subset exists.

For example, given $1, 3, 10, 12, 14$, is there a subset with sum 27? Yes, $\{3, 10, 14\}$. Sum 29? No.

This problem can be modified to ask for a subset S with $\sum_{i \in S} a_i \leq K$ but maximizes $\sum_{i \in S} a_i$. This is the version in [KT] and is slightly more general, but once we solve our equality version, it should be clear how to solve inequality version as well.

Knapsack

Input: n items, each of weight w_i and value v_i, and a positive integer W.
Output: a subset $S \subseteq [n]$ with $\sum_{i \in S} w_i \leq W$ that maximizes $\sum_{i \in S} v_i$.

We can think of W as the weight that the knapsack can hold. Then, the problem asks us to find a maximum value subset that we can fit in the knapsack. Alternatively, we can think of W as the total time that we have, and out objective is to choose a subset of jobs that can be finished on time while maximizing our income.

Knapsack is more general than subset-sum as there are two parameters to consider, but again it will not be difficult once we solved subset-sum. So, let's start with subset-sum.

To come up with the recurrence, we start with exhaustive search algorithm for the subset-sum problem. We will consider all possibilities. Start with the first number a_1. Then, either we choose it or not.

- If we choose a_1, then we need to choose a subset from $\{2, \ldots, n\}$ fo that the sum is $K - a_1$.
- Otherwise, if we don't choose a_1, then we need to choose a subset from $\{2, \ldots, n\}$ so that the sum is K.

A naive implementation will consider all subsets, and this gives an exponential time algorithm.

The observation is, we don't really need to keep track of which subset we have chosen so far, as long as they have the same sum. This suggests that we can just keep track of the (partial) sum in the recursion, which allows us to reduce the search space significantly.

The subproblems that we will consider are $\texttt{subsum}[i, L]$ for $1 \leq i \leq n$ and $L \leq K$ where $\texttt{subsum}[i, L]$

returns true if and only if there is a subset in $\{1, \ldots, n\}$ with sum L. Then the original problem we would like to solve is $\texttt{subsum}[1, K]$. The recurrence relation is

$$\texttt{subsum}[i, L] = (\texttt{subsum}[i+1, L-a_i] \quad \text{OR} \quad \texttt{subsum}[i+1, L])$$

The former case corresponds to choosing a_i, while the latter case corresponds to not choosing a_i, If either case returns true, then return true. Otherwise, when both return false, return false.

Algorithm 25: top-down subset-sum

```
1 Function subsum(i, L):
2     if L = 0 then
3         return True
4     if i > n or L < 0 then
5         return False // running out of numbers, or partial sum too large.
6     return (subsum(i + 1, L − a_i) OR subsum(i + 1, L))
```

Correctness follows from the recurrence relation explained above.

Time complexity There are totally nk subproblems, n choices for i and K choices for L. Each subproblem can be solved by looking up two values. Using top-down memorization, the time complexity is $O(nK)$.

Pseudo-polynomial time note that the time complexity is $O(nK)$, which depends on K. When K is small, this is fast. But K could be exponential in n (as n-bit number can be as big as 2^n), in which case this is even slower than the naive exhaustive search (and also uses much more space). We call this type of time complexity pseudo-polynomial. This is probably avoidable, as we will see that the subset-sum problem is NP-complete.

Implementations

Bottom-up computation We use a 2D-array $\text{subsum}[n][K]$ to store the values of all subproblems. We can compute these values in reverse order from n to 1.

Algorithm 26: Bottom-up computation for subsum

```
1 subsum[i][L] = False for all 1 ≤ i ≤ n and 0 ≤ L ≤ K // initialization
2 subsum[n][a_n] = subsum[n][0] = True // the YES cae for the size 1 problem where we
      only have a_n
3 subsum[i][0] = True for all 1 ≤ i ≤ n // the YES case when the target is zero
4 for i ← n downto 1 do
5     for i ← 1 to K do
6         if subsum[i + 1][L] = True then
7             subsum[i][L] = True
8         if L − a_i ≥ 0 and subsum[i + 1][L − a_i] = True then
9             subsum[i][L] = True
```

Space-Efficient Implementation With this bottom-up implementation, we see that we don't need to use an $n \times K$ array. When we are computing $\text{subsum}[i][*]$ in the outer for-loop, we just use the values of $\text{subsum}[i+1][*]$. So, we can throw away the values $\text{subsum}[\geq i+2][*]$ and only use a $2 \times K$ array, which is a significant saving.

Top-Down vs Bottom-Up The bottom-up implementation don't need recursions, and also leads to a space-efficient algorithm. But the run-time is always exactly nK, to compute all the subproblems. When there is a solution, the top-down approach may run faster, as it may be able to find a solution by solving only a few subproblems. From this perspective, it may make a difference by solving $\texttt{subsum}(i+1, L-a_i)$ before $\texttt{subsum}(i+1, L)$.

Tracing a solution By following a path of "true" from $\texttt{subsum}(1, K)$, we can find a subset of sum K.

- If $\texttt{subsum}(2, K-a_1) = \mathsf{True}$, then put a_1 in our solution and recurse.
- Otherwise, don't put a_1 in the solution, and follow a "true" path from $\texttt{subsum}(2, K)$.

Dynamic Programming and Graph Search Dynamic programming reduces the search space to polynomial size. We can draw a graph. Each vertex is a subproblem, and the edges are added according to the recurrence relation. That is, there is a direct edge from (i, L) to $(i+1, L)$ and from (i, L) to $(i+1, L-a_i)$ if $L - a_i \geq 0$. Then the problem is equivalent to determining whether there is a directed path from the starting state $(1, K)$ to a "true" "base state" (e.g., $(i, 0)$ for some i). The way the top-down memorization algorithm works is basically doing a DFS on this "subproblem graph", by recursing and marking subproblems visited.

Knapsack (Sketch)

We outline how to solve the knapsack problem, which is quite similar to solving subset-sum. From now on, we won't start from the exhaustive search again.

First approach We use a similar recurrence as in subset-sum. The subproblems are $\texttt{knapsack}(i, W, V)$, which is true if and only if there is a subset in $\{i, i+1, \ldots, n\}$ with total weight W and total value V. With this 3D-table, we should be able to write a recurrence as in subset-sum to solve the problem.

Better recurrence It is possible to just use 2 parameters as in subset-sum. Note that some subproblems dominate other subproblems, e.g., if $\texttt{knapsack}(i, W, V+1) = \mathsf{True}$, then we can ignore the subproblem $\texttt{knapsack}(i, W, V)$. So, the idea is to only have subproblems $\texttt{knapsack}(i, W)$ and keep track of the max value achievable. Define $\texttt{knapsack}(i, W)$ as the maximum value that we can earn using items in $\{i, \ldots, n\}$ with total weight $\leq W$. More precisely, let

$$\texttt{knapsack}(i, W) = \max_{S \subseteq \{i, \ldots, n\}} \left\{ \sum_{j \in S} v_j \;\middle|\; \sum_{j \in S} w_j \leq W \right\}.$$

Then the recurrence relation is

$$\texttt{knapsack}(i, W) = \max\{v_i + \texttt{knapsack}(i+1, W - w_i),\ \texttt{knapsack}(i+1, W)\}.$$

The first case corresponds to choosing item i, thus earning v_i and the maximum value of using items from $i+1$ to n when the capacity left in the knapsack is $W - w_i$. The second case corresponds to not choosing item i.

With this recurrence relation, it si not difficult to compute the algorithm and the analysis as in subset-sum. Just need to be careful in the base cases: when $W < 0$, return $-\infty$; when $i > n$, return 0. The time complexity is $O(n, W)$.

5.3 Longest Increasing Subsequence

Given n numbers $a_1, \ldots, a_n$, a subsequence is a subset of these numbers taken in order of the form $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ where $1 \leq i_1 < i_2 < \cdots < i_k \leq n$, and a subsequence is increasing if $a_{i_1} < a_{i_2} < \cdots < a_{i_k}$.

Input: n numbers $a_1, a_2, \ldots, a_n$
Output: an increasing subsequence of maximum length.

For example, given $5, 1, 9, 8, 8, 8, 4, 5, 6, 7$ (recognize this?), the longest increasing subsequence is $1, 4, 5, 6, 7$. By now, it should be relatively straightforward to solve this problem using dynamic problems.

Subproblems Let $L(i)$ be the length of a longest increasing subsequence starting at a_i and only using the numbers in $a_i, \ldots, a_n$. So, there are only n subproblems.

Final answer After we compute $L(1), L(2), \ldots, L(n)$, the final answer is $\max_{1 \le i \le n}\{L(i)\}$.

Recurrence relation Given we start at a_i, we try all possible numbers a_j with $j > i$ and $a_j > a_i$, and form an increasing subsequence starting from a_i by concatenating with a longest increasing subsequence starting at a_j. More precisely, $L(i) = 1 + \max_{i+1 \le j \le n}\{L(j) \mid a_j > a_i\}$. Note that the subsequences formed must be increasing. The correctness can be proved by induction, i.e., if $L(i+1), \ldots, L(n)$ are correct, then $L(i)$ is also correct.

Algorithm 27: Bottom-up implementation for longest increasing subsequence

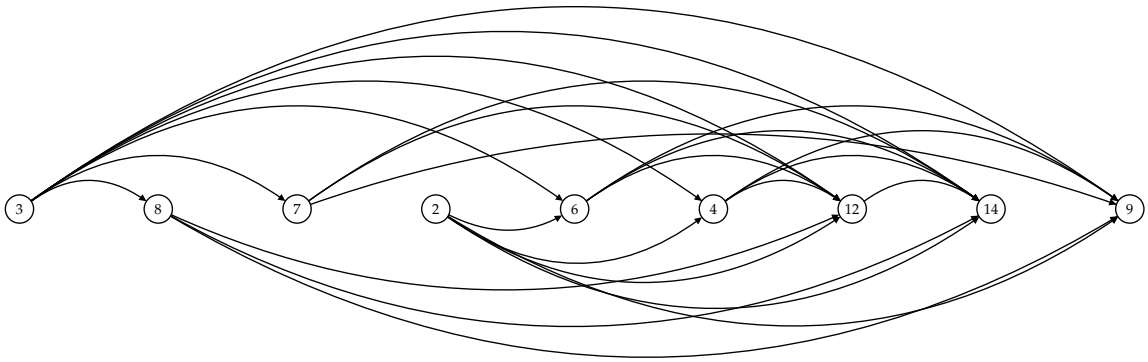
```
1 L(i) = 1 for all 1 ≤ i ≤ n // initialization
2 for i ← n downto 1 do
3     for j ← i + 1 to n do
4         if a_j > a_i and L(j) + 1 > L(i) then
5             L(i) ← L(j) + 1
```

For example, given $3, 8, 7, 2, 6, 4, 12, 14, 9$, then L-values are $4, 3, 3, 4, 3, 3, 2, 1, 1$.

Time complexity $O(n^2)$.

Printing a longest increasing subsequence One way to do it is to keep track of the next number (e.g., $\mathsf{parent}[i] = j$) when we update $L(i) \leftarrow L(j) + 1$. We can also directly trace back a longest increasing subsequence using the L-values in $O(n)$ time without using extra storage.

Longest path in DAG An alternative way to think about this problem is to find a longest path in a DAG. Given n numbers $a_1, \ldots, a_n$, we create a graph of n vertices, each corresponding to a number. There is a directed edge from i to j if $j > i$ and $a_j > a_i$. Then, an increasing subsequence corresponds to a directed path in this directed acyclic graph, and vice versa. So, a longest path in the graph gives us a longest increasing subsequence. For example, given $3, 8, 7, 2, 6, 4, 12, 14, 9$, the graph is



In general, the longest path problem in DAG can solved by dynamic programming efficiently.

A Faster Algorithm for Longest Increasing Subsequence

There is a clever algorithm to solve the problem in $O(n \log n)$ time. The observation is that we don't need to store all the subproblems, as some subproblems are "dominated" by other subproblems. For each length k, we will only store the "best" position to start an increasing subsequence of length k. Then it will turn out that these best positions satisfy a monotone property, and this allows us to use binary search to update these values in $O(\log n)$ time when we consider a new element. This is high level summary and now we discuss the details.

For a given length k, consider the indices $i < i_1 < i_2 < \cdots < i_\ell$ so that $L(i_1) = L(i_2) = \cdots = L(i_\ell) = k$. What is the best subproblem to keep for future computations of $L(i), L(i-1), \ldots, L(1)$? Since we are extending these increasing subsequences using elements in $\{1, \ldots, i\}$, the starting positions $i_1, i_2, \ldots, i_\ell$ are not important. What is important is the starting value.

If $L(i_1) = L(i_2) = k$ and $a_{i_1} > a_{i_2}$, then the subproblem $L(i_1)$ dominates $L(i_2)$, because any increasing subsequence using numbers in $\{a_1, \ldots, a_i\}$ that can be extended by an increasing subsequence of length k, starting at a_{i_2} can also be extended by an increasing subsequence of length k starting at a_{i_1}. That is, among the increasing subsequences of length k, the one with largest starting value is easiest to be extended. Therefore, we define $\mathsf{pos}[k] = \text{argmax}_{j>i}\{a_j \mid L(j) = k\}$, when $L(i)$ is the current subproblem to be computed.

Intuitively, $\mathsf{pos}[k]$ is the best position to start an increasing subsequence of length k after the current index i. Let $m = \max_{i<j\le n}\{L(j)\}$ be the length of a longest increasing subsequence we have computed so far. By the reasoning above, when we compute $L(i)$, we just need to consider $L(\mathsf{pos}[1]), \ldots, L(\mathsf{pos}[m])$, as the other subproblems are dominated by these subproblems. For example, given the sequence $2, 7, 6, 1, 4, 8, 5, 3$, when we compute $L(2)$, we have $L(3) = L(5) = 2, L(4) = 3$, and $L(6) = L(7) = L(8) = 1$, then we only keep $\mathsf{pos}[1] = 6$ with $a_6 = 8$, $\mathsf{pos}[2] = 3$ with $a_3 = 6$, $\mathsf{pos}[3] = 4$ with $a_4 = 1$.

Once we only keep the best subproblems, we have the following important monotone property.

Claim $a[\mathsf{pos}[1]] > a[\mathsf{pos}[2]] > \cdots > a[\mathsf{pos}[m]]$ where $m = \max_{i<j\le n}\{L(j)\}$ and $L(i)$ is the current subproblem.

(Intuition: A longer subsequence should be more difficult to be extended, i.e., its starting value is smaller.)

Proof:
Suppose, by contradiction, that there exists j such that $a[\mathsf{pos}[j]] \ge a[\mathsf{pos}[j-1]]$. Let an optimal increasing subsequence of length j be $a_{p_1} < a_{p_2} < \cdots < a_{p_j}$ where $p_1 = \mathsf{pos}[j]$. Then $a_{p_2} < \cdots < a_{p_j}$ is an increasing subsequence of length $j-1$ with $a_{p_2} > a_{p_1} = a[\mathsf{pos}[j]] \ge a[\mathsf{pos}[j-1]]$, contradicting that $\mathsf{pos}[j-1]$ is the best position to start an increasing subsequence of length $j-1$. □

Suppose we have found the best subproblems $\mathsf{pos}[m], \mathsf{pos}[m-1], \ldots, \mathsf{pos}[1]$ after processing the numbers $a_n, \ldots, a_{i+1}$. Now, we process the number a_i, and would like to update the best subproblem for future computations. We consider three cases.

1. When $a_i < a[\mathsf{pos}[m]]$.

   This is the good case, as we can extend the longest increasing subsequence so far by one, by adding a_i in front of the increasing subsequence of length m starting at $\mathsf{pos}[m]$. So, we can increase m by 1, and set $\mathsf{pos}[m] = i$.

2. When $a[\mathsf{pos}[j]] \le a_i < a[\mathsf{pos}[j-1]]$.

Since $a_i > a[\text{pos}[j]]$, we cannot use a_i to form an increasing subsequence of length $j+1$. But we can use a_i to form an increasing subsequence of length j, by adding a_i in front of the increasing subsequence of length $j-1$ starting at $\text{pos}[j-1]$.

Furthermore, this increasing subsequence of length j is better than the one starting at $\text{pos}[j]$, as $a_i > a[\text{pos}[j]]$. So, in this case, we update $\text{pos}[j] = i$.

3. When $a[\text{pos}[1]] \leq a_i$.

   In this case, we cannot use a_i to extend any increasing subsequence because it is larger than all the starting values, but we can use it to update $\text{pos}[1] = i$.

Note that since $a[\text{pos}[m]] < a[\text{pos}[m-1]] < \cdots < a[\text{pos}[1]]$, we can use binary search to find the smallest j so that $a[\text{pos}[j]] \leq a_i$, and then we update by above rules.

Algorithm 28: Faster algorithm for longest increasing subsequence

```
1 m = 1, pos[1] = n // base case
2 for i ← n − 1 downto 1 do
3     if a_i < a[pos[m]] then
4         m ← m + 1
5         pos[m] ← i // longer increasing subsequence
6     else
7         use binary search to find the smallest j so that a[pos[j]] ≤ a_i
8         pos[j] ← i
9 return m
```

Time complexity $O(n \log n)$.

5.4 Longest Common Subsequence

Input: Two strings $a_1, \ldots, a_n$ and $b_1, \ldots, b_m$, where each a_i, b_j is a symbol.
Output: The largest k such that there exist $i_1 < i_2 < \cdots < i_k$ and $j_1 < j_2 < \cdots < j_k$ such that $a_{i_\ell} = b_{j_\ell}$ for $1 \leq \ell \leq k$.

One example is that we are given two DNA sequences and want to identify common structures.

$$\begin{aligned} S_1 &= AAACCGTGAGTTATTCGTTCTAGAA \\ S_2 &= CACCCCTAAGGTACCTTTGGTTC \end{aligned} \quad \Longrightarrow \quad ACCTAGTACTTTG$$

The longest subsequence (LIS) is a special case of the longest common subsequence problem (LCS).

$$3,8,7,2,6,4,12,14,9 \quad \text{(for LIS)} \quad \xrightarrow{\text{reduces to}} \quad \begin{aligned} S_1 &= 3,8,7,2,6,4,12,14,9 \\ S_2 &= 2,3,4,6,7,8,9,12,14 \end{aligned} \quad \text{(for LCS)}$$

Since the second sequence is sorted, it forces the solution of LCS to be an increasing subsequence.

Let $C(i,j)$ be the length of a longest common subsequence of $a_i, \ldots, a_n$ and $b_j, \ldots, b_m$. Then the answer we are looking for is $C(1,1)$. The base cases are $C(n+1,j) = 0$ for all $1 \leq j \leq m$, and $C(1,m+1) = 0$ for all $1 \leq i \leq n$. To compute $C(i,j)$, there are three cases, depending on whether a_i and b_j are used or not.

1. (Use both a_i and b_j) If $a_i = b_j$, then we can put a_i and b_j in the beginning of a common subsequence, then the remaining subproblem is to find a longest common subsequence for $a_{i+1}, \ldots, a_n$ and $b_{j+1}, \ldots, b_m$. So, let $\text{SOL}_1 = 1 + C(i+1, j+1)$ if $a_i = b_j$. Otherwise $\text{SOL}_1 = 0$.

2. (Not use a_i) Then we find a longest common subsequence for $a_{i+1}, \ldots, a_n$ and $b_j, \ldots, b_m$. So, let $\mathsf{SOL}_2 = C(i+1, j)$.
3. (Not use b_j) Then we find a longest common subsequence for $a_i, \ldots, a_n$ and $b_{j+1}, \ldots, b_m$. So, let $\mathsf{SOL}_3 = C(i, j+1)$.

Then we take the best out of these three possibilities. That is, $C(i,j) = \max\{\mathsf{SOL}_1, \mathsf{SOL}_2, \mathsf{SOL}_3\}$.

Correctness All solutions for $C(i,j)$ fall into at least one of the above three cases. We can then prove correctness by induction.

Time complexity There are $n \cdot m$ subproblems. Each subproblem looks up three values. Using top-down memorization, the total time complexity is $O(m \cdot n)$.

Tracing out solution We can either record some "parent" information when computing $C(i,j)$. We can also compute it directly using $C(i,j)$ only, by recursively going to a subproblem that gives the maximum value for $C(i,j)$.

Algorithm 29: Bottom-up implementation for longest common subsequence

```
1 C(i, m+1) ← 0 for all 1 ≤ i ≤ n
2 C(n+1, j) ← 0 for all 1 ≤ j ≤ m
3 for i ← n downto 1 do
4     for j ← m downto 1 do
5         if a_i = a_j then
6             SOL ← 1 + C(i+1, j+1)
7         else
8             SOL ← 0
9         C(i, j) ← max{SOL, C(i+1, j), C(i, j+1)}
```

5.5 Edit Distance

Input: Two strings $a_1, \ldots, a_n$ and $b_1, \ldots, b_m$, where each a_i, b_j is a symbol.
Output: The minimum k so that we can do k add/delete/change operations to transform $a_1, \ldots, a_n$ into $b_1, \ldots, b_m$.

For example, if two input strings are `SNOWY` and `SUNNY`, the following are two ways:

```
S - N O W Y          - S N O W - Y
S U N N - Y          S U N N - - Y
   Cost: 3              Cost: 5
```

In the first way, we match `S`, add `U`, match `N`, change `O` to `N`, delete `W`, and match `Y`. This takes three add/delete/change operations to transform `SNOWY` and `SUNNY`. The second way requires five add/delete/change operations to transform `SNOWY` and `SUNNY`. We call the minimum number of operations to transform one string to anther string the "edit distance" between two strings. It is a useful measure of the similarity of two strings, e.g., in a word processor.

The recurrence is similar to that in LCS. Let $D(i,j)$ be the edit distance of the strings $a_i, \ldots, a_n$ and $b_j, \ldots, b_m$. The answer we want is $D(1,1)$. The base case is $D(n+1, m+1) = 0$. To compute $D(i,j)$, there are four possible operations to perform.

Add We add b_j to the current string, when $j \leq m$. For example,

$$\begin{array}{ll|l} \cdots & & abc \\ \cdots & & def \end{array} \quad \Longrightarrow \quad \begin{array}{lll|l} \cdots & - & & abc \\ \cdots & d & & ef \end{array}$$

Then we match one more symbol to the target string and move on. More precisely,

$$\mathsf{SOL}_1 = \begin{cases} 1 + D(i, j+1) & \text{if } j \leq m \\ \infty & \text{otherwise.} \end{cases}$$

Delete We delete a_i from the current string, when $i \leq n$. For example,

$$\begin{array}{ll|l} \cdots & & abc \\ \cdots & & def \end{array} \quad \Longrightarrow \quad \begin{array}{lll|l} \cdots & a & & bc \\ \cdots & - & & def \end{array}$$

Then we move one symbol forward in the current string. More precisely,

$$\mathsf{SOL}_2 = \begin{cases} 1 + D(i+1, j) & \text{if } i \leq n \\ \infty & \text{otherwise} \end{cases}$$

Change We change a_i to b_j when $i \leq n$ and $j \leq m$. Then we move one symbol forward in both strings. For example,

$$\begin{array}{ll|l} \cdots & & abc \\ \cdots & & def \end{array} \quad \Longrightarrow \quad \begin{array}{lll|l} \cdots & a & & bc \\ \cdots & d & & ef \end{array}$$

More precisely,

$$\mathsf{SOL}_3 = \begin{cases} 1 + D(i+1, j+1) & \text{if } i \leq n \text{ and } j \leq m \\ \infty & \text{otherwise} \end{cases}$$

Match If $i \leq n$ and $j \leq m$ and $a_i = b_j$, then we match and move one symbol forward in both strings. For example,

$$\begin{array}{ll|l} \cdots & & abc \\ \cdots & & aac \end{array} \quad \Longrightarrow \quad \begin{array}{lll|l} \cdots & a & & bc \\ \cdots & a & & ac \end{array}$$

More precisely,

$$\mathsf{SOL}_4 = \begin{cases} D(i+1, j+1) & \text{if } i \leq n \text{ and } j \leq m \\ \infty & \text{otherwise} \end{cases}$$

Finally, we set $D(i, j) = \min\{\mathsf{SOL}_1, \mathsf{SOL}_2, \mathsf{SOL}_3, \mathsf{SOL}_4\}$.

Correctness Follows from the base case and an inductive argument. In the inductive step as discussed above, we have considered all the possibilities to transform one string to another string.

Time complexity There are $m \cdot n$ subproblems, each requiring a constant number of operations. Using top-down memorization, the time complexity is $O(n \cdot m)$.

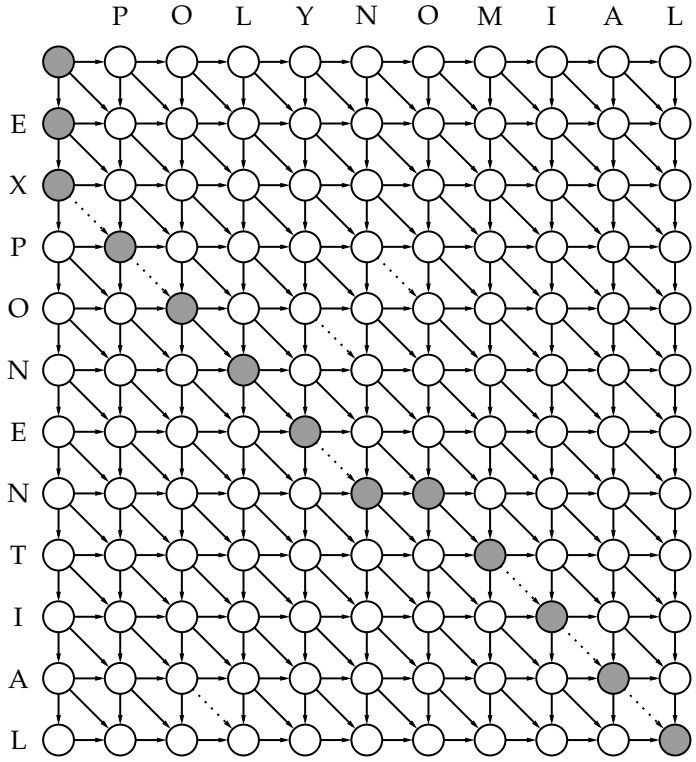
Algorithm 30: Edit distance

```
1  D(i,0) ← i for all 0 ≤ i ≤ m
2  D(0,j) ← j for all 1 ≤ j ≤ n
3  for i ← 1 to m do
4      for j ← 1 to n do
5          if a_i = b_j then
6              diff(i,j) ← 0
7          else
8              diff(i,j) ← 1
9          D(i,j) = min{D(i−1,j)+1, D(i,j−1)+1, D(i−1,j−1)+diff(i,j)}
10 return D(m,n)
```

The algorithm presented here is from [DPV 6.3], which is a bit different from the discussion here (bottom-up implementation).

Graph searching Once again, we would like to point out that dynamic programming can be thought of as finding a (shortest) path from the starting state to the target state in the state graph. This connection is even more transparent when we are using the state table to trace out a solution.

In the graph[1] below, all edge lengths are 1, except for $\{(i-1, j-1) \to (i, j) : x[i] = y[j]\}$ (shown dotted in the figure), whose length is 0. The final answer is the simply the distance between nodes $s(0, 0)$ and $t(m, n)$.

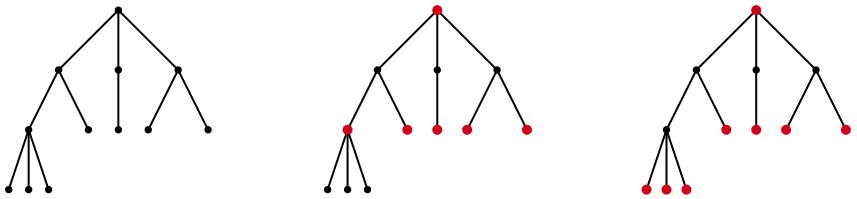


[1] adapted from [DPV 6.3]

5.6 Independent Sets on Trees

Given a graph $G = (V, E)$, a subset of vertices $S \subseteq V$ is called an independent set if $ij \notin E$ for all $i, j \in S$. We will see that the problem of finding a maximum cardinality independent st is NP-complete, but we can use dynamic programming to solve the problem in polynomial time on trees.

Input: A tree $T = (V, E)$
Output: An independent set $S \subseteq V$ of maximum cardinality.



T 7 vertices 9 vertices (optimal)

Having a tree structure suggests a natural way to use dynamic programming. We will define a subproblem for each subtree rooted at a vertex v. They key point is that since there are no edges between different subtrees, we can solve the problem on each subtree separately and thus reduce to smaller subproblems. Then we can write a recurrence relation between a parent and its children.

Actually, we have used this idea before. When we compute the low[] array to compute all cut vertices, we have already used dynamic programming on trees.

Let $I(v)$ be the size of a maximum independent set in the subtree rooted at vertex v. The answer we want is $I(\text{root})$. Bases cases are $I(\text{leaf}) = 1$ for all leaves in the tree. To compute $I(v)$, we consider two possibilities:

1. v is in the independent set.

   Then all its children cannot be included in the independent set. The optimal way to extend the current partial solution is to take a maximum independent set in each subtree rooted at its grand children. So in this case, the maximum size is $1 + \sum_{w:w \text{ grandchild of } v} I(w)$.

2. v is not in the independent set.

   The optimal way to extend the current partial solution is to take a maximum independent set in each subtree rooted at its children. So, in this case, the maximum size is $\sum_{w:w \text{ child of } v} I(w)$.

Therefore,

$$I(v) = \max\left\{1 + \sum_{w:w \text{ grandchild of } v} I(w), \sum_{w:w \text{ child of } v} I(w)\right\}$$

Correctness It follows from the explanation of the recurrence relation and induction.

Time complexity There are n subproblems. Each subproblem requires (# children + # grandchildren) lookups. Note that

$$\sum_{v \in V} (\#\text{ children} + \#\text{ grandchildren}) = \sum_{v \in V} (\#\text{ parent} + \#\text{ grandparent}) \leq \sum_{v \in V} 2 = 2n\,,$$

by counting the sum in a different way (i.e., counting upward instead of counting downward). So using top-down memorization, the total time complexity is $O(n)$.

We can also write a recurrence relation involving children only. The idea is to use two subproblems on a vertex v. Let $I^+(v)$ be the size of a maximum independent set with v included, and $I^-(v)$ be

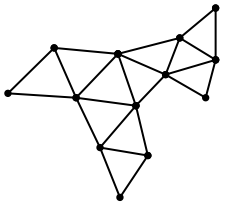
the size of a maximum independent set with v excluded. Then the answer is $\max\{I^+(\mathsf{root}), I^-(\mathsf{root})\}$. The base cases are $I^+(\mathsf{leaf}) = 1$ and $I^-(\mathsf{leaf}) = 0$ for all leaves. The recurrence are

$$I^+(v) = 1 + \sum_{w:w \text{ child of } v} I^-(w) \quad \text{and} \quad I^-(v) = \sum_{w:w \text{ child of } v} \max\{I^+(w), I^-(w)\}$$

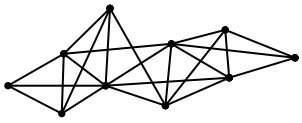
Note that we can extend the algorithm to solve the maximum weighted independent set problem on trees.

5.7 Dynamic Programming on "Tree-like" Graphs

Many optimization problems are hard on graphs but easy on trees using dynamic programming. Generalizing the ideas using dynamic programming on trees, it is possible to show that dynamic programming also works on "tree-like" graphs. For example,



"treewidth" 2



"treewidth" 3

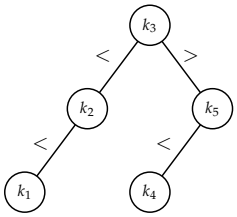
There is a way to define the "tree-width" of a graph so as to measure how "close" a graph is to a tree. If the tree-width is small, then dynamic programming works faster, usually with runtime $\sim O(n^{\text{treewidth}})$. This has become an important paradigm to deal with hard programs on graphs, at least in research papers. Read [KT 10.4, 10.5] for an introduction to this approach. Read CO 442 for an introduction to tree-width.

5.8 Optimal Binary Search Tree

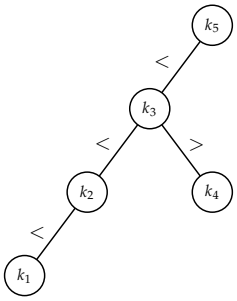
This problem is a bit similar to the Huffman coding problem, also about finding an optimal binary tree. Imagine the scenario where there are n commonly searched strings, e.g., some French vocabularies for English meanings. We would like to build a good data structure to support these queries efficiently. And somehow we have decided to use a binary search tree (say instead of using hashing). As there are n strings, we could build a balanced binary search tree to answer the queries in $O(\log_2 n)$ time. As in Huffman coding, suppose we know the frequencies of the searched strings, can we use this information to design a better binary search tree so as to minimize the average query time?

Input: n keys $k_1 < k_2 < \cdots < k_n$, frequencies $f_1, f_2, \ldots, f_n$ with $\sum_{i=1}^n f_i = 1$.
Output: a binary search tree T that minimizes the objective value $\sum_{i=1}^n f_i \cdot \mathsf{depth}_T(k_i)$.

For example, given $f_1 = 0.1,\ f_2 = 0.2,\ f_3 = 0.25,\ f_4 = 0.05,\ f_5 = 0.4$,



objective value
$= 0.25 \times 1 + 0.2 \times 2$
$+ 0.4 \times 2 + 0.1 \times 3$
$+ 0.05 \times 3$
$= 1.9$



objective value
$= 0.4 \times 1 + 0.25 \times 2$
$+ 0.2 \times 3 + 0.05 \times 3$
$+ 0.1 \times 4$
$= 2.05$

In the above example, even though k_5 has the highest frequency, it is not necessarily optimal to put k_5 at the root for it to have minimum depth. So, this is unlike the prefix coding problems, in which

keys with higher frequencies will have smaller depth. This is because we have to maintain the binary search tree structure, such that smaller keys have to be put on the left subtree while larger keys have to be put on the right subtree. This restriction turns out to be useful in setting up the recurrence relation.

The subproblem structure is slightly different from those that we have seen before. In the following, we let $F_{i,j} = \sum_{\ell=i}^{j} f_\ell$. To handle boundary cases, we let $F_{i,j} = 0$ for $i > j$. Let $C(i,j)$ be the objective value of an optimal binary search tree for keys $k_i < \cdots < k_j$. The desired answer is $C(1,n)$. Base cases are $C(i,i) = f_i$ for $1 \le i \le n$. To handle boundary cases, we also set $C(i,i-1) = 0$ for all i.

To compute $C(i,j)$, we try all the possible root of the binary search tree. For $1 \le \ell \le j$, if we set k_ℓ to be the root, then the keys $k_i, \ldots, k_{\ell-1}$ must be on the left subtree of the root, while keys $k_{\ell+1}, \ldots, k_j$ must be on the right subtree of the root. The two subtrees can be computed independently of each other, as there are no more constraints between the two subtrees. So, the best way is to find an optimal binary search tree for $k_i, \ldots, k_{\ell-1}$ on the left, and an optimal binary search tree for $k_{\ell+1}, \ldots, k_j$ on the right. Therefore,

$$\begin{aligned} C(i,j) &= \min_{1 \le \ell \le j} \left\{ \underbrace{f_\ell}_{\text{root}} + \underbrace{F_{i,\ell-1} + C(i,\ell-1)}_{\text{left subtree}} + \underbrace{F_{i+1,j} + C(\ell+1,j)}_{\text{right subtree}} \right\} \\ &= \min_{i \le \ell \le j} \left\{ F_{i,j} + C(i,\ell-1) + C(\ell+1,j) \right\} \end{aligned}$$

Note that the terms $F_{i,\ell-1}$ and $F_{\ell+1,j}$ ar added because the keys $k_i, \ldots, k_{\ell-1}$ and the keys $k_{\ell+1}, \ldots, k_j$ are put one level lower, and so the two terms account for the increase in the objective value.

Correctness follows from the justification of the recurrence formula and by induction.

Time complexity There are no more than n^2 subproblems. Each subproblem looks up no more than n values. Using top-down memorization, the total time complexity is $O(n^3)$.

To bottom-up implement this problem, it requires more care to write it correctly. We will solve the subproblems with $j - i = 1$ first, and then $j - i = 2$, and so on.

Algorithm 31: Bottom-up implementation for optimal binary search tree

```
1 C(i, i-1) = 0 for 1 ≤ i ≤ n
2 Compute F_{i,j} for all 1 ≤ i, j ≤ n // can be done in O(n^2) time using partial sums
3 for 1 ≤ width ≤ n - 1 do
4     for i ← 1 to (n - width) do
5         j ← i + width
6         C(i, j) ← ∞
7         for ℓ ← i to j do
8             C(i, j) ← min{C(i, j), F_{i,j} + C(i, ℓ-1) + C(ℓ+1, j)}
```

It is clear the time complexity is $O(n^3)$ as there are three for-loops.

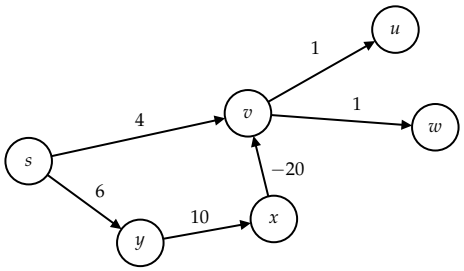
With additional observations, Knuth used the same subproblems but showed how to solve the problem in $O(n^2)$ time!

5.9 Single-Source Shortest Paths with Arbitrary Edge Lengths

Input: A directed graph $G = (V, E)$, an edge length ℓ_e for each edge $e \in E$, and a vertex $s \in V$.
Output: The shortest path distances from s to every vertex $v \in V$.

This problem is solved using Dijkstra's algorithm in the special case where the edge lengths are non-negative, which runs in near-linear time. It turns out that allowing negative edge lengths makes the problem considerably harder. Let's first see why Dijkstra's algorithm does not work in this more general setting.

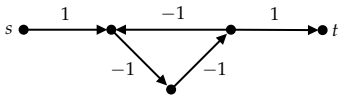
In Dijkstra's algorithm, we maintain a set $R \subseteq V$ so that $\mathsf{dist}[v]$ is computed correctly for all $v \in R$, and then we grow R greedily by adding a vertex $v \notin R$ closest to R into R. With the presence of the negative edge lengths, however, this greedy algorithm does not maintain this invariant anymore.



In the example, the vertex v is closest to s and is added to R first, but $\mathsf{dist}[v] \neq 4$ as the path s, y, x, v is of length -4 because of the negative edge xv. The correctness of Dijkstra's algorithm crucially uses that the length of a prefix of a path cannot be shorter than the length of the path, and this doesn't hold anymore with the presence of negative edges. With this wrong start, Dijkstra's algorithm will add u and w to R. Only after vertex x is added and explore later, we realize that there is a shorter path from s to v via x. Then we know that the distances to u and w are not computed correctly.

To fix it, we need to use the new distance to v to update the distance to u and w, but then we can no longer say that each vertex is only explored once. We can extend this example so that this update process needs to be done many times. This is where we could not maintain the near-linear time complexity for solving this more general problem.

Another issue of having negative edges is that there may exist negative cycles.



In the example, from s to t, we can go around the negative cycle as many times as we want, and so the shortest path distance is not even well-defined. In the following, we will study the algorithms solve the following problems:

1. If G has no negative cycles, solve the single-source shortest paths problem.
2. Given a directed graph G, check if there exists a negative cycle C, i.e., $\sum_{e \in C} \ell_e < 0$.

5.9.1 Bellman-Ford Algorithm

Although Dijkstra's algorithm may not compute all distances correctly in one pass, it will compute the distances to some vertices correctly, e.g., the first vertex on a shortest path.

In the example above, $\mathsf{dist}[y]$ will eb computed correctly. Then, if we do the update on every edge again, then we would get $\mathsf{dist}[x]$ right for sure. Then, with one more update pm every edge, then we would get $\mathsf{dist}[v]$ correct and so on. How many times we need to do? If the graph has no negative cycles, then any shortest walk must be a simple path, which has at most $n-1$ edges. So, by repeating the updating phases at most $n-1$ times, we should have computed all shortest path distances correctly, with time complexity about $O(nm)$. This is basically the Bellman-Ford algorithm.

To formalize the above idea, we design an algorithm using dynamic programming to compute the

shortest path distance from s to every vertex $v \in V$ using at most i edges, from $i = 1$ (base case) to $i = n-1$. Then we will show that this is equivalent to the Bellman-Ford algorithm we see in textbooks.

Let $D(v,i)$ be the shortest path distance from s to v using at most i edges. For each $v \in V$, $D(v,n-1)$ is the shortest path distance from s to v when the graph has no negative cycles. For the base cases, we let $D(s,0) = 0$ and $D(v,0) = \infty$ for all $v \in V - s$.

To compute $D(v,i+1)$, note that a path with at most $i+1$ edges from s to v must be coming from a path using at most i edges from s to u for an in-neighbor u of v. Since we are to compute the shortest path distance from s to v using at most $i+1$ edges, we should use a shortest path using at most i edges from s to u. We try all possibilities of u and get the recurrence relation

$$D(v,i+1) = \min\left\{D(v,i), \quad \min_{u:uv\in E}\{D(u,i)+\ell_{uv}\}\right\}.$$

Time complexity Given $D(v,i)$ for all $v \in V$, it takes in-deg(w) time to compute $D(w,i+1)$. So, the time to compute $D(w,i+1)$ for all $w \in V$ is $O(\sum_{w\in V}\text{in-deg}(w)) = O(m)$. We do this for $1 \le i \le n-1$, thus the total time complexity is $O(nm)$.

Space complexity A direct implementation requires $\Theta(n^2)$ space to store all the values $D(v,i)$. Note that to compute $D(w,i+1)$ for all $w \in V$, we just need the values $D(v,i)$ for all $v \in V$ but don't need $D(v,j)$ for $j \le i-1$, and so we can throw these away and only use $O(n)$ space.

The algorithm can be made even simpler, matching the intuition that we mentioned in the beginning.

Algorithm 32: Bellman-Ford algorithm

```
1 dist[s] = 0
2 dist[v] = ∞ for all v ∈ V − s
3 for i ← 1 to n − 1 do
4     foreach edge uv ∈ E do
5         if dist[u] + ℓ_uv < dist[v] then
6             dist[v] = dist[u] + ℓ_uv and parent[v] = u
```

This is the Bellman-Ford algorithm. The simplification is that we don't need to use two arrays. To see why we don't need two arrays, note that using one array could only have the intermediate distances smaller, and they remain to be upper bounds on the true distances, so using tighter upper bounds would not hurt (and may speed up in practice).

5.9.2 Shortest Path Tree

As in Dijkstra's algorithm, we would like to return a shortest path from s to v by following the edges $(\text{parent}[v], v)$. In Bellman-Ford, there are many iterations in the outerloop, and it is not clear whether these edges still form a tree. Actually, it is possible to have a directed cycle in the edges $(\text{parent}[v], v)$, but the following lemma shows that these directed cycles must be negative cycles.

Lemma

If there is a directed cycle C in the edges $(\text{parent}[v], v)$, then the cycle C must be a negative cycle, i.e., $\sum_{e\in C}\ell_e < 0$.

Proof:
Let the directed cycle C be $v_1, v_2, \ldots, v_k$, with $v_iv_{i+1} \in E$ for all $1 \le i \le k-1$ and $v_kv_1 \in E$. Assume that v_kv_1 is the last edge in the cycle C formed in the algorithm, i.e., the cycle C formed

when v_k becomes the parent of v_1, while $\mathsf{parent}[v_i] = v_{i-1}$ already for $2 \le i \le k$. Consider the values $\mathsf{dist}[v_i]$ right before v_k becomes the parent fo v_i. Since v_{i-1} is the parent of v_i, we have $\mathsf{dist}[v_i] \ge \mathsf{dist}[v_{i-1}] + \ell_{v_{i-1}v_i}$ for $2 \le i \le k$.

Note that at the time when we set $\mathsf{parent}[v_i] = v_{i-1}$, the inequality holds as an equality, but later $\mathsf{dist}[v_{i-1}]$ could decrease and it may become an inequality. Note also that we cannot have $\mathsf{dist}[v_i] < \mathsf{dist}[v_{i-1}] + \ell_{v_{i-1}v_i}$ as otherwise $\mathsf{parent}[v_i]$ would be updated.

Now when we set $\mathsf{parent}[v_1] = v_k$, it must be because $\mathsf{dist}[v_1] > \mathsf{dist}[v_k] + \ell_{v_k v_1}$ at that time. Adding all these k inequalities, we have

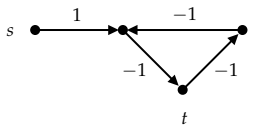
$$\sum_{i=1}^{k} \mathsf{dist}[v_i] > \sum_{i=1}^{k} \mathsf{dist}[v_i] + \sum_{e \in C} \ell_e \,,$$

which implies that $\sum_{e \in C} \ell_e < 0$. □

The lemma implies that if there are no negative cycles, then there are no directed cycles in the edges $(\mathsf{parent}[v], v)$. Assuming that every vertex can be reached from vertex s, then every vertex has exactly one incoming edge in $(\mathsf{parent}[v], v)$, and there are no directed edges by the lemma. So, the edges $(\mathsf{parent}[v], v)$ must form a directed tree, i.e., a tree with edges pointing away from s. To conclude, when there are no negative edges, the edges $(\mathsf{parent}[v], v)$ form a shortest path tree from s.

5.9.3 Negative Cycles

We can extend the dynamic programming algorithm to identify a negative cycle if it exists. Even with negative cycles, after k iterations of the dynamic programming, the same recurrence relation proves that we compute the shortest path fro ms to v using at most k edges for all $v \in V$. To solve the single-source shortest paths problem, we only used the assumption that there are no negative cycles to prove that the algorithm can stop after $n-1$ iterations and conclude that distances are computed correctly. If there are negative cycles, then we would expect that $D(v,k) \to -\infty$ as $k \to \infty$ for some $v \in V$. For example, in the graph



we have $D(t,3) = -1$, $D(t,6) = -4$, $D(t,9) = -7$, and so on. On the other hand, if there are no negative cycles, then we expect that $D(v,n) = D(v,n-1)$ for all $v \in V$, and this implies that $D(v,\infty) \not\to -\infty$ as $k \to \infty$ for all $v \in V$. So, intuitively, by checking if $D(v,n) = D(v,n-1)$ for all $v \in V$, we can determine if there is a negative cycle or not.

In the following, we assume that every vertex can be reached from s. For the problem of finding a negative cycle, this is without loss of generality since we can restrict our attention to strongly connected components and we learnt previously how to identify all SCCs in linear time.

We make the above ideas precise by the following claims (proofs see L14.pdf).

Claim 1 If the graph has a negative cycle, then $D(v,k) = -\infty$ as $k \to \infty$ for some $v \in V$.

Claim 2 If the graph has no negative cycles, then $D(v,n) = D(v,n-1)$ for all $v \in V$.

Claim 3 If $D(v,n) = D(v,n-1)$ for all $v \in V$, then the graph has no negative cycles.

Similarly, one can show from the proof of Claim 3: as long as $D(v,k+1) = D(v,k)$ for all $v \in V$, then $D(v,\ell) = D(v,k)$ for all $\ell > k$, and so we can stop in that iteration with all distances computed

correctly. This provides an early termination rule that is useful in practice (when shortest paths have few edges).

Claim 2 and 3 together imply that a graph has no negative cycles if and only if $D(v, n-1) = D(v, n)$ for all $v \in V$. Since we can compute $D(v, n)$ and $D(v, n-1)$ for all $v \in V$ in $O(mn)$ time, this implies an $O(mn)$ time algorithm for checking.

The next question is: if $D(w, n) < D(w, n-1)$ for some w, how do we find a negative cycle? Here we assume that we use $\Theta(n^2)$-space dynamic programming algorithm for computing $D(w, n)$, and that we have stored $\mathsf{parent}[w, n] = u$ if $D(w, n) = D(u, n-1) + \ell_{uw}$.

First, since $D(w, n) < D(w, n-1)$, we know that the path P from s to w with total length $D(w, n)$ and at most n edges must have exactly n edges, as otherwise $D(w, n-1) = D(w, n)$. A path of length n must have a repeated vertex, and thus a cycle C.

We claim C must a negative cycle. Suppose not, then we can skip the cycle C to get a path P' with fewer edges than that in P and $\text{length}(P') \leq \text{length}(P)$ since C is a non-negative cycle. But that would imply that $D(w, n-1) \leq \text{length}(P')$ since P' has at most $n-1$ edges, and thus $D(w, n-1) \leq \text{length}(P') \leq \text{length}(P) = D(w, n)$, a contradiction. So, the cycle C must be a negative cycle.

By tracing out the parents using the stored information, we can find P and thus the cycle C. This gives an $O(mn)$-time algorithm to find a negative cycle, using $\Theta(n^2)$ space.

There is also an $O(mn)$-time algorithm using only $O(n)$ space. Details in [KT 6.10].

5.10 All-Pairs Shortest Paths

Input: A directed graph $G = (V, E)$, an edge length ℓ_e for $e \in E$.
Output: The shortest path length from s to t, for all $s, t \in V$.

We can solve this by running Bellman-Ford for each $s \in V$. This would take $O(n^2 m)$ time, which could be $\Theta(n^4)$ when $m = \Theta(n^2)$. It is possible to solve the all-pairs shortest paths problem in $O(n^3)$, using a different recurrence. Here we present the Floyd-Warshall algorithm.

In the Floyd-Warshall algorithm, more subproblems are used to store information for each pair of vertices.

Let the vertex set $V = \{1, 2, \ldots, n\}$. Let $D(i, j, k)$ be the length of a shortest path from vertex i to vertex j using only vertices $\{1, \ldots, k\}$ as intermediate vertices in the path. (Another perhaps more natural choice is $D'(i, j, k)$ which denotes the length of a shortest path from i to j using at most k edges similar to that in the Bellman-Ford algorithm. The question was left such that $D'(i, j, k)$ doesn't work as well as the Floyd-Warshall subproblems). And the answer we want is $D(i, j, n)$ for all $i, j \in V$.

Base cases are $D(i, j, 0) = \ell_{ij}$ if $ij \in E$ and $D(i, j, 0) = \infty$ if $ij \notin E$. This is because $D(i, j, 0)$ is asking for the shortest path length from i to j without using intermediate vertices.

Assume $D(i, j, k)$ are computed correctly for all $i, j \in V$ for some k. We would like to compute $D(i, j, k+1)$ for all $i, j \in V$. The only difference between $D(i, j, k+1)$ and $D(i, j, k)$ is that $D(i, j, k+1)$ is allowed to use vertex $k+1$ as an intermediate vertex, while $D(i, j, k)$ is not allowed to do so. To use vertex $k+1$ as an intermediate vertex for a path between i and j, the path has to go from i to $k+1$ and then from $k+1$ to j. The optimal way to do this using only vertices $[k+1]$ as intermediate vertices is to use a shortest path from i to $k+1$ using $[k]$ as intermediate vertices, and a shortest path from $k+1$ to j using $[k]$ as intermediate vertices. Note that we don't need to use vertex $k+1$ more than once, as there are no negative cycles. Therefore,

$$D(i, j, k+1) = \min\{D(i, j, k), \quad D(i, k+1, k) + D(k+1, j, k)\},$$

where the first term considers the paths not going through $k+1$, while the second term consider paths that use vertex $k+1$.

Algorithm 33: Floyd-Warshall algorithm

```
D(i, j, 0) = ∞ for ij ∉ E
D(i, j, 0) = ℓ_ij for ij ∈ E.
for k ← 0 to n − 1 do
    for i ← 1 to n do
        for j ← 1 to n do
            D(i, j, k + 1) = min{D(i, j, k),   D(i, k + 1, k) + D(k + 1, j, k)}
```

Runtime is $O(n^3)$. It has been a long standing open problem whether there exists a truly sub-cubic time algorithm for computing all-pairs shortest paths, e.g., $O(n^{2.99})$.

5.11 Traveling Salesman Problem

Input: A directed graph $G = (V, E)$, with an edge length ℓ_{ij} for all $i, j \in V$.
Output: A cycle C that visits every vertex exactly once and minimizes $\sum_{e \in C} \ell_e$.

It is one of the famous problem in combinatorial optimization. As we will show in the last part of the course, this problem is NP-hard.

There is a naive algorithm for this problem, by enumerating all possible orderings to visit the vertices. This will take $\Theta(n! \cdot n)$ time. It becomes too slow when $n = 13$. We present a dynamic programing solution that can probably work up to $n = 30$.

The difficulty of the problem is that it is not enough to remember only the shortest paths, but also what vertices that we have visited so that what other vertices yet to visit. The recurrence is unlike everything that we have seen so far, as it has exponentially many subproblems!

Let $C(i, S)$ be the length of a shortest path to go from 1 to i, with vertices in S on the path. Note that we don't care about the ordering of vertices in S in the path, and this is where the speedup over the naive algorithm is coming from.

The answers we want are $\min_{i \in V}\{C(i, V) + \ell_{i1}\}$, from 1 to i using all vertices in V once, then come back to 1. Base cases are $C(i, \{1, i\}) = \ell_{1i}$ for all $i \in V$.

Suppose we have computed $C(i, S)$ for all subsets S of size k. We would like to use these to compute $C(i, S)$ for all subsets S of size $k+1$. To compute $C(i, S)$ for S of size $k+1$, we try all possibilities of the second last vertex on the path. Note that the second last vertex must be from S, and of course the best way to reach the second last vertex j is to use a shortest path from 1 to j that reaches every vertex in $S - \{i\}$ exactly once. Therefore, we have

$$C(i, S) = \min_{j \in S - \{i\}} \{C(j, S - \{i\}) + \ell_{ji}\}.$$

There are $O(n \cdot 2^n)$ subproblems, each requiring $O(n)$ time to compute. So, the total time complexity is $O(n^2 \cdot 2^n)$. The main drawback of this algorithm is that the space complexity is $\Theta(n \cdot 2^n)$.

6

Bipartite Graphs

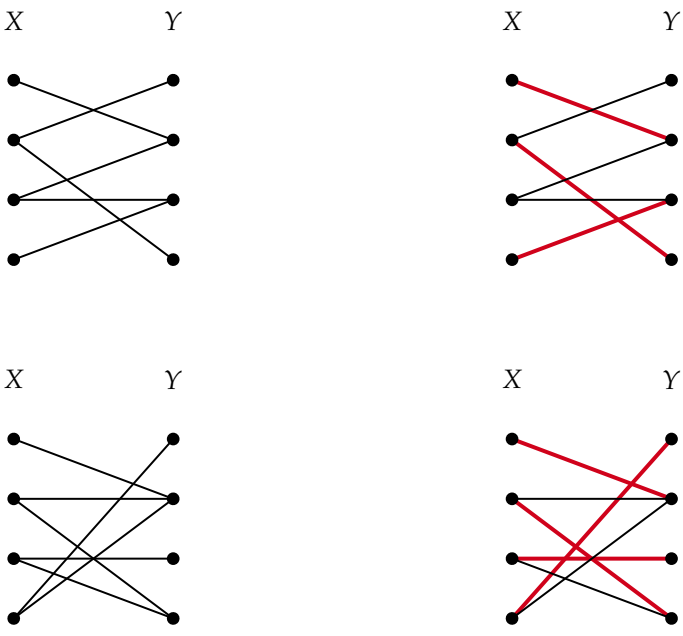
6.1 Bipartite Matching

The bipartite matching problem is an important problem both in theory and in practice. In this chapter, we will study efficient algorithms for solving bipartite matching and its "dual" problem minimum vertex cover, and see some interesting and non-trivial applications of these problems. We will learn a new technique called "the augmenting path" method to solve the bipartite matching problem. This is an important technique that underlies many algorithms for combinatorial optimization problems, including the network flow problem in CO 351. More generally, the augmenting path method can be understood as one way of solving combinatorial optimization problems using the general framework of linear programming.

The problem statement is as follows:

Input: A bipartite graph $G = (X, Y; E)$.
Output: A maximum cardinality subset of edges that are vertex disjoint.

Here consider two inputs, and maximum matchings are highlighted in the output. Note that the second example is a "perfect" matching.



A subset of edges $M \subseteq E$ is called a **matching** if edges in M are pairwise vertex disjoint, or in other words, no two edges in M share a vertex. Given a matching $M \subseteq E$, we say a vertex v is matched if v is an endpoint of some edge $e \in M$; otherwise we say v is **unmatched** or **free** or unsaturated. We are interested in finding a **maximum matching**, a matching with the maximum number of edges. A matching is called a **perfect matching** if every vertex in the graph is matched.

Obviously, a perfect matching is the best that we can hope for the maximum matching problem. As shown in the above example, not all bipartite graphs have a perfect matching. We will see a nice characterization of graphs that don't have a perfect matching.

As we will see later, there are many interesting and non-trivial applications of bipartite matching, as we will see later. Here we first see a standard and useful application, which also explains why the bipartite matching is sometimes called the **assignment problem**.

We are given n jobs, and m people/machines. Each person/machine is only capable of doing a subset of jobs. Our task is to assign all the jobs to people/machines, without assigning more than one job to a person/machine. We can model this as a bipartite matching problem. We create one vertex for each job, and one vertex for each person/machine. We add an edge between job vertex j and a person vertex p if and only if person p is capable of doing job j. So, the graph is bipartite. By construction, a matching corresponds to an assignment of jobs to people such that no one is assigned more than one job. Let $|J|$ be the number of jobs. Then the assignment problem is possible if and only if there is a matching of $|J|$ edges.

We study a new algorithmic approach to solve the bipartite matching problem. A natural first approach is to go greedy: whenever there is an edge $uv \in E$ with both endpoints free, then we add the edge uv to our partial solution and repeat until there are no such edges. This may not find a maximum matching.

How do we know what edges are in an optimal solution? Actually, we don't know of an easy way to tell whether an edge belongs to some maximum matching or not. This is unlike in the shortest path problem or in the minimum spanning tree problem, where we can prove that a shortest edge or a minimum weight edge can always be extended to an optimal solution. Instead of making a greedy decision and commit to it, the new approach is to find efficient ways to *improve* the current solution if it is not optimal. So, the augmenting path method can be understood as a *local search* algorithm. There are no known dynamic programming algorithms for bipartite matching.

A path $v_1, v_2, \ldots, v_{2k}$ is an **augmenting path** of a matching M if

1. v_1 and v_{2k} are free/unmatched,
2. $v_{2i-1}v_{2i} \notin M$ for all $1 \leq i \leq k$,
3. $v_{2i}v_{2i+1} \in M$ for all $1 \leq i \leq k-1$.

If we have found an augmenting path, we can use it to improve the matching size by one, by removing the even edges from M and add the odd edges to M. Simply by "switching" edges. Note that an edge with both endpoints free is an augmenting path of length one.

By trying some examples, surprisingly, it turns out that finding an augmenting path is all one need to do.

Proposition

M is a maximum matching if and only if there is no augmenting path of M.

One direction is easy. The other direction involves discussion by cases. See MATH 249 for details. The

proposition suggests the following "local search" algorithm for finding a maximum matching.

Algorithm 34: Bipartite matching

$M \leftarrow \varnothing$
while *there is an augmenting path* $P = v_1 v_2 \dots v_{2k}$ *of* M **do**
  $M \leftarrow M - \{v_{2i}v_{2i+1} \mid 1 \le i \le k-1\} + \{v_{2i-1}v_{2i} \mid 1 \le i \le k\}$
return M

Correctness follows from the proposition. The time complexity is $O(n \cdot T(n,m))$ where $T(m,n)$ is the time complexity to find an augmenting path of M if it exists, or report that no such paths exist, in a graph with n vertices and m edges.

There is an algorithm by Edmonds and Karp which solves the problem in $O(m\sqrt{n})$ time.

Then to complete the bipartite matching algorithm, we need to develop an algorithm to find an augmenting path. Thanks to the structure of the bipartite graph, we can design a simple algorithm for this. First we need to find a free vertex v_1 on the left. If there is a neighbor v_2 of v_1 which is unmatched, then we have found an augmenting path v_1v_2 of length one. Otherwise, v_2 is matched, and to extend it to an augmenting path, we have no choice but to follow the matching edge of v_2 to go back to the left, call the matching edge v_2v_3. Then, we repeat the above step: if v_3 has an unmatched neighbor v_4, then we found an augmenting path; otherwise we follow the matching edge v_4v_5 to go back to the left.

Observe that every time we go to the right, either we are done or we follow a matching edge that takes us back to the left, because of the bipartite graph structure. THe important idea is to encode the color information on the edges by directions, such that each unmatched edge points from the left to right and each matched edge points from right to left.

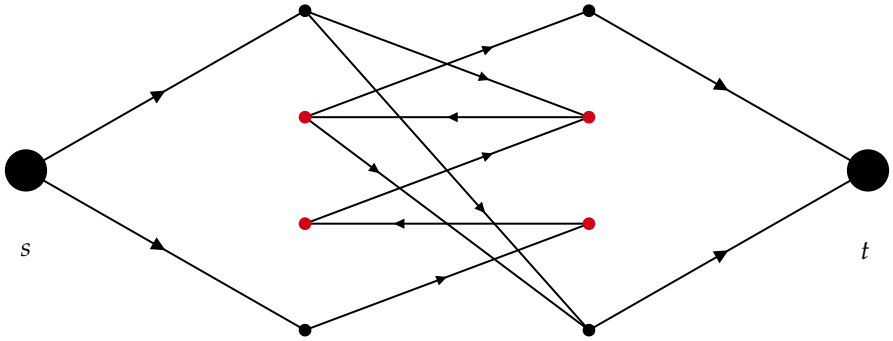
G $\qquad$ $\vec{G}_M$

augmenting path of M in G $\qquad$ directed path in $\vec{G}_M$

Given a bipartite graph $G = (X, Y; E)$ and a matching $M \subseteq E$, we construct a directed graph $\vec{G}_M$ on the same vertex set and the same edge set, while the direction of every edge in M is from X to Y and the direction of every edge in $E - M$ is from Y to X. By construction, we have the following correspondence between augmenting paths in G and directed paths in $\vec{G}_M$.

Claim There is an augmenting path of M in $G = (X, Y; E)$ if and only if there is a directed path in $\vec{G}_M$ from a free/unmatched vertex in X (on the left) to a free/unmatched vertex in Y (one the right).

The claim allows us to reduce the problem of finding an augmenting path of M in G to a "reachability" problem in $\vec{G}_M$, for which we know how to solve by BFS/DFS. A direct implementation is to start a BFS/DFS on each free vertex on the left and check if can reach a free vertex on the right, but this could take $\Omega(n(n+m))$ time. There is a simple trick to solve this "multiple-source multiple-sink" problem in $O(n+m)$ time. We just add a super-source vertex on the left, with directed edges to the free vertices on the left, and we add a super-sink vertex on the right, with directed edges from the free vertices on the right.



$\vec{G}'_M$

By construction, we reduce the augmenting path problem in G to the s-t reachability problem in $\vec{G}'_M$.

Claim There is a directed path from a free vertex on the left to a free vertex on the right in $\vec{G}_M$ if and only if there is a directed path from s and t in $\vec{G}'_M$.

Corollary There is an augmenting path of M in G if and only if there is a directed path from s to t in $\vec{G}'_M$.

Algorithm 35: Augmenting path

Input: a bipartite graph $G = (X, Y; E)$, and a matching $M \subseteq E$.
Output: an augmenting path P of M in G, or report that no such paths exist.

```
1 Construct the directed graph G'_M as described above.
2 Use BFS/DFS to determine if there exists a directed path P in G'_M from s to t.
3 if yes then
4     return P // forget the directions and also remove the edges from s and the
          edges to t
5 else
6     return "No"
```

The time complexity is $O(m+n)$. In actual implementation, we can work directly on the directed graph and it is easy to update, so that we don't need to construct $\vec{G}'_M$ in each iteration.

The bipartite matching algorithm fails in general (non-bipartite) graphs. Edmonds and Tutte have some work in this area. See CO 342 for details.

6.2 Bipartite Vertex Cover

Given a graph $G = (V, E)$, a subset of vertices $S \subseteq V$ is called a **vertex cover** if for every edge $uv \in E$, $\{u, v\} \cap S \neq \varnothing$. In words, S is a vertex cover if S intersects every edge.

Input: A bipartite graph $G = (V, E)$
Output: A vertex cover of minimum cardinality.

We could imagine that this problem (in general graphs) is useful in computer networks, say by using the minimum number of computers to monitor all the links in the network.

Actually, vertex cover problem is the dual problem of maximum matching problem, in a precise and meaningful way. First we observe that if there is a matching with k edges, and any vertex cover must have at least k vertices. The reason is simple: since the k matching edges are vertex disjoint, we must use k distinct vertices just to cover these k edges, and so any vertex cover must have at least k vertices. Therefore, the size of a maximum matching is a lower bound on the size of a minimum vertex cover. Surprisingly, this is always a tight lower bound on bipartite graphs. In other words, having a large matching is the only reason that we need a large vertex cover.

Theorem (König)

In a bipartite graph, the maximum size of a matching is equal to the minimum size of a vertex cover.

Proof:
It can be proven using algorithmic approach, or strong duality theorem. □

Algorithm 36: Bipartite vertex cover

1. Use an efficient algorithm to find a maximum matching M (doesn't need to be augmenting path algorithms).
2. Construct the directed graph $\vec{G}'_M$ as described in the previous section.
3. Do a BFS/DFS on $\vec{G}'_M$ to identify all vertices $S \subseteq V$ reachable from S.
4. Return $(Y \cap S) \cup (X - S)$ as the vertex cover.

The time complexity is dominated by the first step, as the remaining steps take only $O(n + m)$. Using the augmenting path algorithm before gives $O(mn)$-time algorithm for bipartite vertex cover. Using the $O(m\sqrt{n})$-time algorithm by Edmonds and Karp gives $O(m\sqrt{n})$-time algorithm.

König's theorem is a nice example of a graph-theoretic characterization. To see this, imagine that you work for a company and your boss asks you to find a maximum matching say to assign some jobs to the employees. If your algorithm finds a perfect matching, then your boss would be happy. But suppose there is no perfect matching in the graph, how could you convince your boss about the non-existence? Your boss may just think that you are incompetent and other smarter people could find a better assignment. With the augmenting path algorithm, maybe you could explain that there is no augmenting path for your matching and so it is maximum, but this may not be so easy to explain to your boss. A more convincing way is to show that a vertex cover of size $< \frac{n}{2}$, and explain that if there is a perfect matching such a vertex cover could not exist.

These min-max theorems are some of the most beautiful results in combinatorial optimization, providing succinct "proofs" for both YES-case (a large matching) and the NO-case (a small vertex cover). They show the non-existence of a solution by the existence of a simple obstruction. This is the same idea when studying the duality in the context of programming and optimization.

To put the above discussion into perspective, consider the dynamic programming algorithms. Even

though we could solve the problems in polynomial time, we don't have such a succinct characterization for the NO-cases. If your boss asks why there is no better solution, it would be quite difficult to explain. The "proof" is still succinct in the sense that it is polynomial sized table, but it is not nearly as elegant as the proofs provided by the min-max theorems.

Theorem (Hall)

A bipartite graph $G = (X, Y; E)$ with $|X| = |Y|$ has a perfect matching if and only if for every subset $S \subseteq X$m it holds that $|N(S)| \geq |S|$ where $N(S)$ is the neighbor set of S in Y.

Corollary Every d-regular bipartite graph $G = (X, Y; E)$ has a perfect matching.

7

Undecidability and Intractability

7.1 Polynomial Time Reductions

Once we have learned more and more algorithms, they become our building blocks and we may not need to design algorithms for new problems from scratch every time. So it becomes more and more important to be able to use existing algorithms to solve new problems. We have already seen some reductions. For instances, we have reduced subset-sum to knapsack, longest increasing subsequence to longest common subsequence, etc. In general, if there is an efficient reduction from problem A to problem B and there is an efficient algorithm to solve problem B, then we have an efficient algorithm to solve problem A.

7.1.1 Decision Problems

To formalize the notion of a reduction, it is more convenient to restrict our attention to decision problems, for which the output is just YES or NO, so that every problem has the same output format. For example, instead of finding a maximum matching, we consider the decision version of the problem "Does G has a matching of size at least k?" As we will discuss later, for all the problems that we will consider, if we know how to solve the decision version of our problem in polynomial time, then we can use the decision algorithm as a blackbox/subroutine to solve the search version of our problem in polynomial time.

polynomial time reductions

We say a decision problem A is polynomial time reducible to a decision problem B if there exists a polynomial time algorithm F that maps/transforms any instance I_A of A into an instance I_B of B (that is, $F(I_A) = I_B$) such that I_A is a YES instance of problem A if and only if I_B is a YES instance of problem B.

We use the notation $A \leq_P B$ to denote that such a reduction exists, intuitively saying that problem A is not more difficult than B in terms of polynomial time solvability.

Now suppose we have such a polynomial time reduction algorithm F and a polynomial time algorithm

ALG_B to solve problem B, then we have the following polynomial time algorithm to solve problem A.

Algorithm 37: Solving problem A by reduction

Input: an instance I_A of problem
Output: whether I_A is a YES-instance

1 Use the reduction algorithm F to map/transform I_A into $I_B = F(I_A)$ of problem B.
2 Return $\mathsf{ALG}_B(I_B)$.

Correctness follows from the property of reduction algorithm F that I_A is a YES-instance of problem A if and only if I_B is a YES-instance of problem B, and the correctness of ALG_B to solve problem B.

Time complexity Suppose F has time complexity $p(n)$ for an instance I_A of size n where $p(n)$ is a polynomial in n, and ALG_B has time complexity $q(m)$ for an instance I_B of size m where $q(m)$ is a polynomial in m. Then the above algorithm has time complexity $q(p(n))$, a polynomial in the input size n.

So far it is all familiar: We reduce problem A to problem B efficiently, and use an efficient algorithm for problem B to obtain an efficient algorithm to solve problem A. Now we explore the other implication of the inequality $A \leq_P B$. Suppose problem A is known to be impossible to be solved in polynomial time. Then $A \leq_P B$ implies that B cannot be solved in polynomial time either, as otherwise we can solve problem A in polynomial time using the reduction algorithm proven above. Therefore, if A is computationally hard and $A \leq_P B$, then B is also computationally hard.

By our current knowledge, however, we know almost nothing about proving a problem cannot be solved in polynomial time, and so we could not draw such a strong conclusion from $A \leq_P B$. But suppose there is a problem, say the traveling salesman problem, which is very famous and has attracted many brilliant researches to solve it in polynomial time but without any success. Now our boss gives us a problem C, and we couldn't solve it in polynomial time, it would be much more convincing if we could prove that $\mathsf{TSP} \leq_P C$. This is what we will be doing in this chapter!

7.1.2 Simple Reductions

We will show that the following three problems are equivalent in terms of polynomial-time solvability. Either way they can be solved in polynomial time or they all cannot be solved in polynomial time.

Maximum Clique (Clique)

A subset of vertices $S \subseteq V$ is a clique if $uv \in E$ for all $u, v \in S$.

Input: Graph $G = (V, E)$, an integer k.
Output: Is there a clique in G with at least k vertices?

Maximum Independent Set (IS)

A subset of vertices $S \subseteq V$ is an independent set if $uv \notin E$ for all $u, v \in S$.

Input: Graph $G = (V, E)$, an integer k.
Output: Is there an independent set in G with at least k vertices?

Minimum Vertex Cover (VC)

A subset of vertices $S \subseteq V$ is a vertex cover if $\{u, v\} \cap S \neq 0$ for all $uv \in E$.

Input: Graph $G = (V, E)$, an integer k.
Output: Is there a vertex cover in G with at most k vertices?

Proposition Clique $\leq_P$ IS and IS $\leq_P$ Clique.

To see the connection between independent sets and vertex covers, we need the following observation.

Observation In $G = (V, E)$, $S \subseteq V$ is a vertex cover of G if and only if $V - S$ is an independent set in G.

With this observation, we can prove the following proposition.

Proposition VC $\leq_P$ IS and IS $\leq_P$ VC.

Note that polynomial time reductions are transitive. Therefore, Clique, IS and VC are equivalent in polynomial time solvability.