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

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

Function `Sort`$(A[1,n])$**:**
  if $n = 1$ **then**
    return
  `Sort`$\left(A\left[1, \lceil \frac{n}{2} \rceil\right]\right)$
  `Sort`$\left(A\left[\lceil \frac{n}{2} \rceil + 1, n\right]\right)$
  merge$\left(A\left[1, \lceil \frac{n}{2} \rceil\right], A\left[\lceil \frac{n}{2} \rceil + 1, n\right]\right)$

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 $O(2^n \cdot n)$.

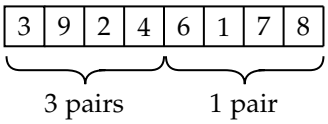
2.3 Counting Inversions

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.

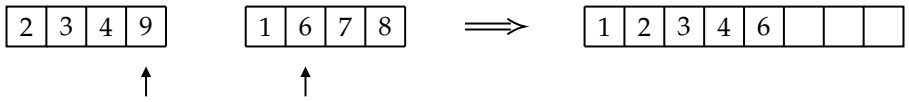
Now, for each number c in the second half, the number of "cross" inversion pairs involving c is precisely the number of numbers in the first half that is larger than c. In this example, 1 is involved in 4 cross pairs, $6, 7, 8$ are all involved in 1 cross pair (with 9), and so the number of "cross" inversion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

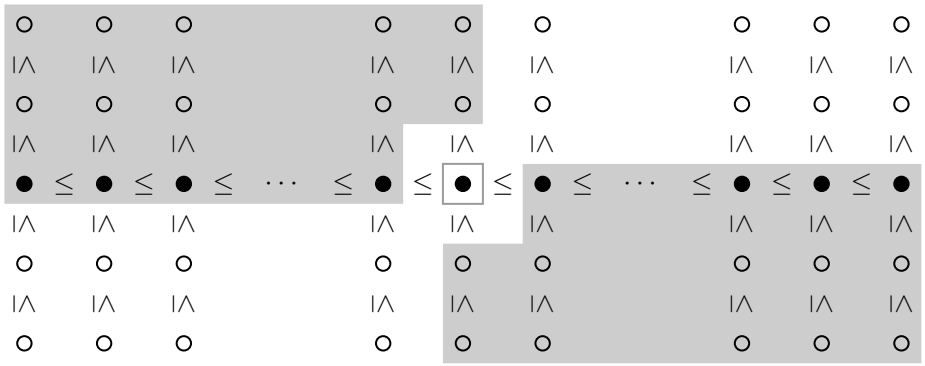
Finding a good pivot The idea of the algorithm is to find the median of medians.

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,

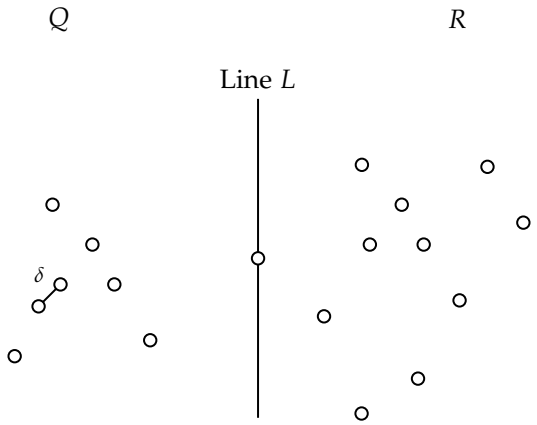
$$T(n) \leq T\left(\frac{7n}{10}\right) + P(n) + c_2 n = T\left(\frac{7n}{10}\right) + T\left(\frac{n}{5}\right) + (c_1 + c_2)n$$

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

2.6 Closest Pair

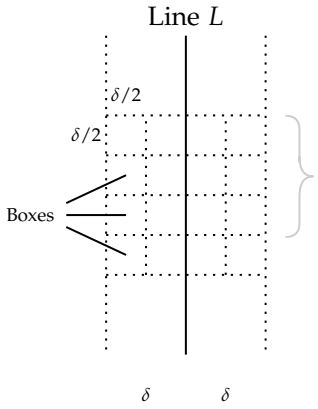
Input: n points (x_1, y_1), (x_2, y_2), ..., (x_n, y_n) on the 2D-plane.
Output: $1 \leq i < j \leq n$ that minimizes the Euclidean distance $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$

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

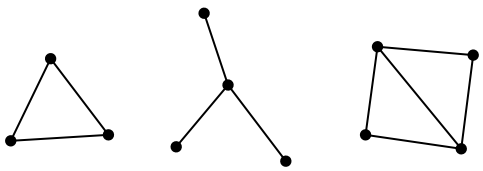
$$A[i, j] = \begin{cases} 1 & \text{if } ij \in E \\ 0 & \text{if } ij \notin E \end{cases}.$$

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 $\text{visited}[v] = \text{True}$.

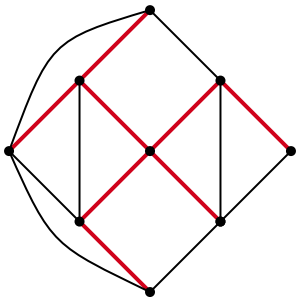
Then we can find all connected components of the graph in $O(n+m)$ time.

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
```

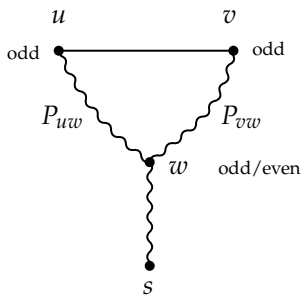
Note that we assume the graph is connected, as otherwise we can solve the problem in each component. The time complexity is $O(m + n)$ as we just do a BFS and then check every edge once.

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).

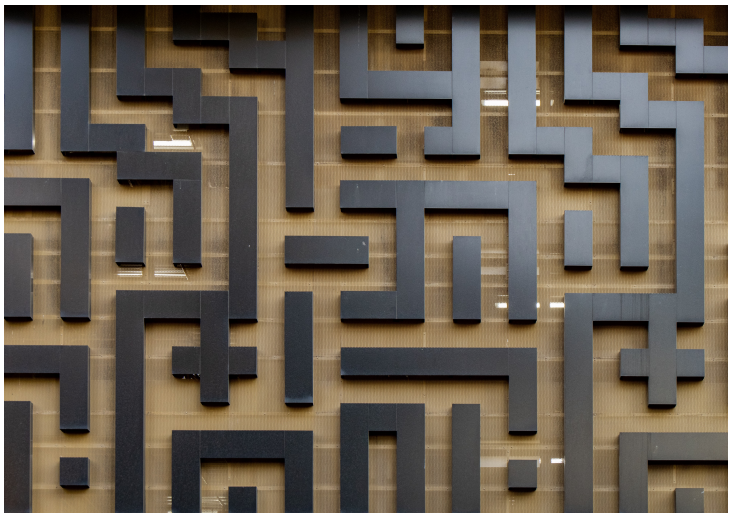


Photo by Mitchell Luo on Unsplash

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

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

Algorithm 10: `explore` in DFS

```
1 Function explore(u):
2     foreach neighbor v of u do
3         if visited[v] = False then
4             visited[v] = True
5             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.

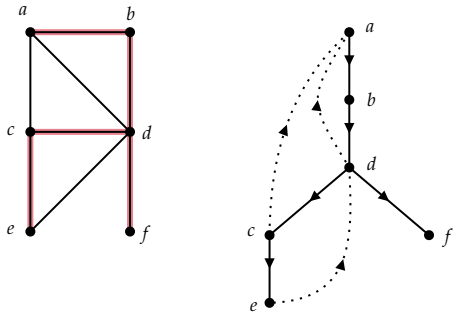
The lemma shows that DFS can also be used to check $s - t$ connectivity, to find the connected component containing s, and to check graph connectivity, all in $O(m + n)$ time. As we can also find all connected components in $O(m + n)$ time using DFS.

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 11: 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)
```

Algorithm 12: `explore` in DFS (with timer)

```
1 Function explore(u):
2     start[u] = time
3     time ← time + 1
4     foreach neighbor v of u do
5         if visited[v] = False then
6             visited[v] = True
7             explore(v)
8     finish[u] = time
9     time ← time + 1
```

Property (parenthesis)

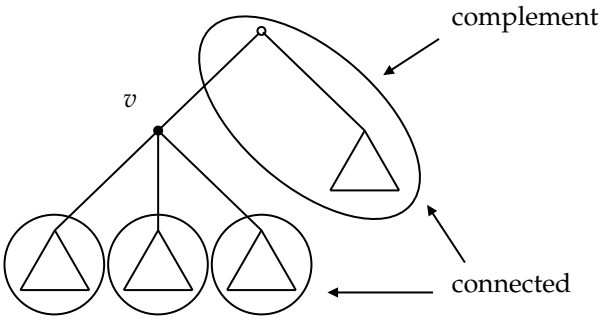
The intervals $[\text{start}(u), \text{finish}(u)]$ and $[\text{start}(v), \text{finish}(v)]$ for two vertices u and v are either disjoint or one is contained in another.

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 $G - v$. □

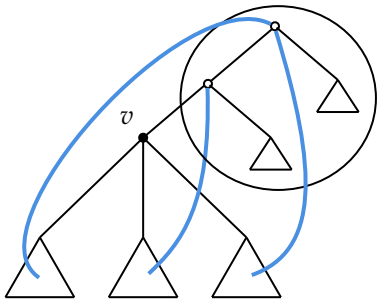
The same argument applies to each subtree below v gives the following characterization of a cut vertex.

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:

⇒ 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.

⇐ 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.)

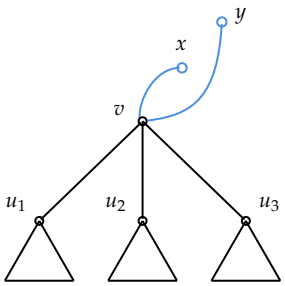
Let us define a value $\mathsf{low}[v]$ for each vertex on the DFS tree:

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

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.

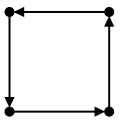
This completes the description of a linear time algorithm to identify all cut vertices given the low array.

3.5 Directed Graphs

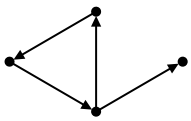
In a directed graph, each edge has direction. When we say uv is a directed edge, we mean the edge is point from u to v, and u is called the **tail** and v is called the **head** of the edge. Given a vertex v, $\text{indeg}(v)$ denotes the number of directed edges with v as the head and we call them incoming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



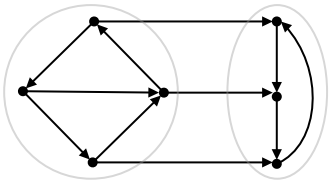
strongly connected



not strongly connected

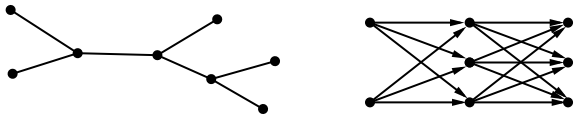
A subset $S \subseteq V$ is called strongly connected if for every pair of vertices $u, v \in S$, u is reachable from v and v is reachable from u.

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 13: 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)
```

Algorithm 14: `explore` in DFS for directed graphs

```
1 Function explore(u):
2     start[u] = time
3     time ← time + 1
4     foreach out-neighbor v of u do
5         if visited[v] = False then
6             visited[v] = True
7             explore(v)
8     finish[u] = time
9     time ← time + 1
```

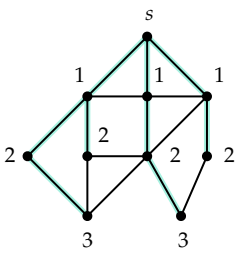
The time complexity is $O(n+m)$, and a vertex t is reachable from s if and only if $\mathsf{visited}[t] = \mathsf{True}$. When we look at all vertices reachable from s, the subset form a "directed cut" with no outgoing edges (but could have incoming edges into the subset). We can define BFS for directed 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

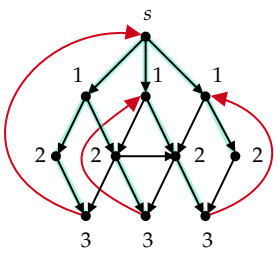
As in for undirected graphs, when a vertex v is first visited, we remember its parent as the vertex u when v is first visited from. The edges $(v, \mathsf{parent}[v])$ form a tree, and both BFS trees and DFS trees are defined in this way.

BFS trees

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



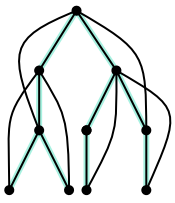
undirected BFS tree



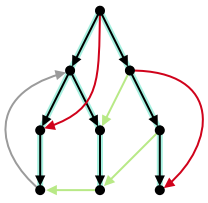
directed BFS tree

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

back edge
forward edge
cross edge

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 15: 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 16: 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".

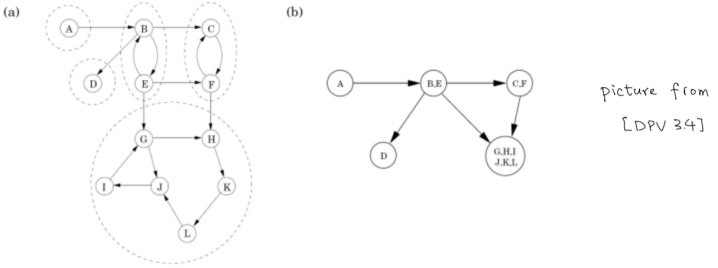
Correctness The lemma proves that if the graph is acyclic, then all edges go forward in this ordering. On the other hand, if the graph is not acyclic, then there is no topological ordering by the proposition.

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, 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.

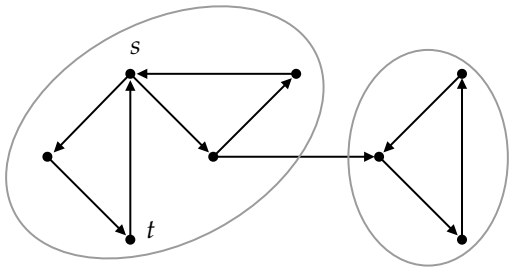
So, now, the question is how to find a vertex in a sink component efficiently? It doesn't look easy.

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 17: 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 18: 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