

# Algorithms

Muchang Bahng

Spring 2024

## Contents

<b>1</b>	<b>Complexity</b>	<b>3</b>
1.1	Recursive Algorithms and Recurrence Relation . . . . .	4
<b>2</b>	<b>Brute Force Algorithms</b>	<b>5</b>
2.1	Basic Arithmetic . . . . .	5
2.2	Lists . . . . .	6
2.3	Stack, Queues, Heaps . . . . .	6
2.4	Cryptography . . . . .	6
2.5	Matrix Operations . . . . .	7
<b>3</b>	<b>Divide and Conquer</b>	<b>7</b>
3.1	Merge Sort and Counting Inversions . . . . .	7
3.2	Selection . . . . .	10
3.3	Multiplication and Matrix Multiplication . . . . .	10
3.3.1	Karatsuba Algorithm . . . . .	10
3.3.2	Strassen Algorithm . . . . .	10
3.4	Fast Fourier Transform . . . . .	10
<b>4</b>	<b>Hashing and Probabilistic Algorithms</b>	<b>10</b>
4.1	Modulo Operations . . . . .	10
4.2	Hashing . . . . .	10
4.3	Primality Testing . . . . .	10
<b>5</b>	<b>Dynamic Programming</b>	<b>10</b>
5.1	Longest Increasing Subsequence . . . . .	10
5.2	Knapsack . . . . .	10
<b>6</b>	<b>Graphs</b>	<b>10</b>
6.1	Representations and Properties . . . . .	10
6.2	Exploration . . . . .	12
6.3	Directed Acyclic Graphs and Topological Sorting . . . . .	15
6.4	Bipartite Graphs . . . . .	21
6.5	Strongly Connected Graphs . . . . .	22
6.6	Shortest Path . . . . .	23
6.7	Negative Weighted Graphs . . . . .	26
6.8	Minimum Spanning Trees . . . . .	29
6.8.1	Prim's Algorithm with Cuts . . . . .	30
6.8.2	Kruskal's Algorithm . . . . .	33
6.8.3	Applications . . . . .	37
<b>7</b>	<b>Greedy Algorithms</b>	<b>38</b>

<b>8</b>	<b>Linear Programming</b>	<b>43</b>
<b>9</b>	<b>Streaming Algorithms</b>	<b>43</b>

# 1 Complexity

A course on the study of algorithms.

## Definition 1.1 (Algorithm)

An **algorithm** is a procedure for solving a mathematical problem in a *finite* number of steps. It should be

1. finite,
2. correct,
3. efficient

An algorithm, with respect to some inputs  $\mathbf{n}$ , will have a runtime that is some function  $f$ . We would like a formal way to analyze the asymptotic behavior between two functions.

## Definition 1.2 (Complexity)

Given two positive functions  $f, g$ ,

1.  $f = O(g)$  if  $f/g$  is bounded.<sup>a</sup>
2.  $f = \Omega(g)$  if  $g/f$  is bounded, i.e.  $g = O(f)$ .
3.  $f = \Theta(g)$  if  $f = O(g)$  and  $g = O(f)$ .

There are two notions of complexity here. We can compare  $f$  and  $g$  with respect to the *value*  $N$  of the input, or we can compare them with respect to the *number of bits*  $n$  in the input. While we mostly use the complexity w.r.t. the value, we should be aware for certain (especially low-level operations), the bit complexity is also important.

Let's do a quick comparison of various functions. Essentially, if we want to figure out the complexity of two positive functions  $f, g$ ,<sup>1</sup> we can simply take the limit.

$$\lim_{x \rightarrow +\infty} \frac{f(x)}{g(x)} = \begin{cases} 0 & \implies f = O(g) \\ 0 < x < +\infty & \implies f = \Theta(g) \\ +\infty & \implies f = \Omega(g) \end{cases} \quad (1)$$

Most of the time, we will have to use L'Hopital's rule to derive these actual limits, but the general trend is

1.  $\log n$  is small
2.  $\text{poly}(n)$  grows faster
3.  $\exp(n)$  grows even faster
4.  $n!$  even faster
5.  $n^n$  even faster

## Theorem 1.1 (Properties)

Some basic properties, which shows very similar properties to a vector space.

1. Transitivity.

$$f = O(g), g = O(h) \implies f = O(h) \quad (2)$$

$$f = \Omega(g), g = \Omega(h) \implies f = \Omega(h) \quad (3)$$

$$f = \Theta(g), g = \Theta(h) \implies f = \Theta(h) \quad (4)$$

$$(5)$$

<sup>a</sup>Note that it is more accurate to write  $f \in O(g)$ , since we consider  $O(g)$  a class of functions for which the property holds.

<sup>1</sup>These will be positive since the runtime must be positive.

## 2. Linearity.

$$f = O(h), g = O(h) \implies f + g = O(h) \quad (6)$$

$$f = \Omega(h), g = \Omega(h) \implies f + g = \Omega(h) \quad (7)$$

$$f = \Theta(h), g = \Theta(h) \implies f + g = \Theta(h) \quad (8)$$

$$(9)$$

**Example 1.1 ()**

Compare the following functions.

1.  $f(n) = \log_{10}(n), g(n) = \log_2(n)$ . Since they are different bases, we can write  $f(n) = \log(n)/\log(10)$  and  $g(n) = \log(n)/\log(2)$ . They differ by a constant factor, so  $f = \Theta(g)$ .
2.  $f(n) = (\log n)^{20}, g(n) = n$ . We have

$$\lim_{n \rightarrow \infty} \frac{(\log n)^{20}}{n} = \lim_{n \rightarrow \infty} \frac{20 \cdot (\log n)^{19} \cdot \frac{1}{n}}{1} = \dots = \lim_{n \rightarrow \infty} \frac{20!}{n} = 0 \implies f = O(g) \quad (10)$$

3.  $f(n) = n^{100}, g(n) = 1.01^n$ . We have

$$\lim_{n \rightarrow \infty} \frac{n^{100}}{1.01^n} = \lim_{n \rightarrow \infty} \frac{100n^{99}}{1.01^n \cdot \log(1.01)} = \dots = \lim_{n \rightarrow \infty} \frac{100!}{1.01^n \cdot (\log 1.01)^{100}} = 0 \implies f = O(g) \quad (11)$$

Let's do a slightly more nontrivial example.

**Example 1.2 ()**

Given the following algorithm, what is the runtime?

```

1  for i in range(1, n+1):
2      j = 1
3      while j <= i:
4          j = 2 * j

```

Now we can see that for each  $i$ , we will double up to  $\log_2(i)$  times. Therefore summing this all over  $i$  is

$$\sum_{i=1}^n \log_2(i) = \log_2(n!) \leq \log_2(n^n) = n \log_2(n) \quad (12)$$

and so we can see that the runtime is  $O(n \log n)$ . Other ways to do this is to just replace the summation with an integral.<sup>a</sup>

$$\int_1^n \log_2(x) dx = x \log(x) - x \Big|_1^n = n \log(n) - n + 1 = O(n \log n) \quad (13)$$

**1.1 Recursive Algorithms and Recurrence Relation**

I assume that the reader is familiar with recursive algorithms. Now to evaluate the runtime of a recursive algorithm, one must implicitly solve for the runtime of its recursive calls.

<sup>a</sup>Need more justification on why this is the case. Mentioned in lecture.

## 2 Brute Force Algorithms

### 2.1 Basic Arithmetic

In here, we use basic deductions from elementary algebra to give us a starting point at which we analyze fundamental arithmetic algorithms.

#### Theorem 2.1 (Complexity of Addition)

The complexity of addition of two  $O(N)$  values with  $n$  bits is

1.  $O(n)$  bit complexity.
2.  $O(\log N)$  complexity.
3.  $O(1)$  memory complexity.

By the same logic, the complexity of subtraction is

1.  $O(n)$  bit complexity.
2.  $O(\log N)$  complexity.
3.  $O(1)$  memory complexity.

#### Proof.

To see bit complexity, we are really taking each bit of each number and adding them together, plus a potential carry operation. Therefore, we are doing a bounded number of computations per bit, which is  $O(1)$ , but we must at least read through all of the bits, making this  $O(\max\{n, m\})$ .

#### Theorem 2.2 (Complexity of Multiplication)

The complexity of multiplication of two values  $N, M$  with bits  $n, m$  is

1.  $O(n^2)$  bit complexity.<sup>a</sup>
2.  $O((\log n)^2)$  complexity.
- 3.

#### Theorem 2.3 (Complexity of Division)

The complexity of multiplication of two values  $N, M$  with bits  $n, m$  is

1.  $O(n^2)$  bit complexity.
- 2.

#### Theorem 2.4 (Complexity of Modulus)

The complexity of multiplication of two values  $N, M$  with bits  $n, m$  is

1.  $O(n^2)$  bit complexity.
- 2.

#### Theorem 2.5 (Complexity of Exponentiation)

The complexity of multiplication of two values  $N, M$  with bits  $n, m$  is

1.  $O(n^2)$  bit complexity.
- 2.

---

<sup>a</sup>It turns out we can do better, which we will learn later.

**Theorem 2.6 (Complexity of Square Root)**

The complexity of multiplication of two values  $N, M$  with bits  $n, m$  is

1.  $O(n^2)$  bit complexity.
- 2.

**Definition 2.1 (Factorial)**

## 2.2 Lists

**Definition 2.2 (Max and Min of List)****Definition 2.3 (Bubble Sort)****Definition 2.4 (Binary Search)**

## 2.3 Stack, Queues, Heaps

A heap is sort of in between a sorted array and an unsorted array.

## 2.4 Cryptography

**Example 2.1 (GCD of Two Numbers)**

Take a look at the following algorithm.

```
1 def gcd(a, b):
2     if a == b:
3         return a
4     elif a > b:
5         return gcd(a - b, b)
6     else:
7         return gcd(a, b - a)
8
9 print(gcd(63, 210))
```

**Definition 2.5 (Primality Testing)****Definition 2.6 (Integer Factorization)**

## 2.5 Matrix Operations

### Definition 2.7 (Matrix Multiplication)

### Definition 2.8 (Singular Value Decomposition)

### Definition 2.9 (QR Decomposition)

### Definition 2.10 (LU Decomposition)

### Definition 2.11 (Matrix Inversion)

## 3 Divide and Conquer

### Definition 3.1 (Divide and Conquer Algorithms)

The general idea is two steps:

1. Divide an input into smaller instances of the same problem. The simplest form is merge sort.
2. Conquer/solve these smaller instances, which takes less time.
3. Merge/Combine the smaller solutions into the original, bigger solution.

This is usually recursive, but does not need to be.

It has its applications in the most elementary operations, in sorting and multiplication. To prove correctness, we use induction by proving the correctness of the base case and then the inductive step to show that there is an invariant.

### 3.1 Merge Sort and Counting Inversions

#### Algorithm 3.1 (Merge Sort)

Merge sort is the first instance.

**Algorithm 1** Merge Sort**Require:** Array `nums`

```

function MERGESORT(nums)
  n = len(nums)
  if n < 2 then                                     ▷base case
    return nums
  end if
  mid = n // 2
  left_sorted = MergeSort(nums[:mid])                 ▷Divide into left half
  right_sorted = MergeSort(nums[mid:])                 ▷Divide into right half
  res = [0] * n
  i = j = k = 0
  while k < n do                                     ▷Merge the sorted subarrays
    if j == len(right_sorted) or left_sorted[i] < right_sorted[j] then
      res[k] = left_sorted[i]                         ▷We should add the next element from left array
      i += 1                                           ▷if it is smaller or if right array is filled already
    else
      res[k] = right_sorted[j]
      j += 1
    end if
    k += 1
  end while
  return res
end function

```

The recurrence relation for the runtime is as follows. Let  $T(n)$  represent the worst-case runtime of MergeSort of size  $n$ . Then, we have

$$T(n) = 2 \cdot T(n/2) + O(n) \quad (14)$$

Consisting of two recursive calls with input size  $n/2$  and then the merge step which is  $O(n)$ . But if we take a look at this, we have

$$T(n) = 2 \cdot T(n/2) + O(n) \quad (15)$$

$$= 2 \cdot (2 \cdot T(n/4) + O(n/2)) + O(n) \quad (16)$$

$$= 4 \cdot T(n/4) + O(n) + O(n) \quad (17)$$

and the number of times  $O(n)$  is added up is  $\log n$ , meaning that this recurrence relation turns into  $O(n \log n)$ .

**Definition 3.2 (Inversions)**

Given two lists of ranked items, say

$$\text{Alice : } a > b > c > d > e \quad (18)$$

$$\text{Bob : } b > d > a > e > c \quad (19)$$

We want to measure the dissimilarity between two rankings by counting the number of *inversions*, which are pairs of items for which one person orders the opposite of the other (e.g.  $a, b$  for above).<sup>a</sup> So how many inversions are there? We can do this in  $\Theta(n^2)$  by explicitly looking at all  $n$  pairs. Without loss of generality, we can assume that the first list is sorted simply by bijectively relabeling



these elements for both lists. Therefore, the set of inversions is defined to be

$$\{(i, j) \text{ s.t. } i < j \text{ and } A[i] > A[j]\} \quad (20)$$

### Algorithm 3.2 (Counting Inversions)

The idea is very similar. By assuming that the first list is sorted, we can simply count the number of inversions in a single list  $A$ .

```
1 A = 5 6 1 3 4 8 2 7
```

1. In the divide step, we count all inversions in  $A_l, A_r$ , which are the left and right sides of  $A$ , and we sort  $A_l, A_r$  to add additional structure.

```
1 1 3 5 6 | 2 4 7 8
```

2. In the conquer step, we merge them linearly but every time we add an element from  $A_r$  into our result, this reveals that there are  $k$  additional inversions added where  $k$  is the number of elements left in  $A_l$  to add.

---

### Algorithm 2 Counting Inversions

---

**Require:** Array `nums`

```
function INVERSIONS(nums)
  n = len(nums)
  if n < 2 then                                     ▷base case
    return nums
  end if
  mid = n // 2
  left_sorted, left_invs = Inversions(nums[:mid])    ▷Divide into left half
  right_sorted, right_invs = Inversions(nums[mid:])  ▷Divide into right half
  res = [0] * n
  i = j = k = 0                                     ▷left, right, and combined index
  inv = 0                                             ▷number of inversions
  while k < n do                                     ▷Merge the sorted subarrays
    if j == len(right_sorted) or left_sorted[i] < right_sorted[j] then
      res[k] = left_sorted[i]                       ▷We should add the next element from left array
      invs += len(left_sorted - i)                   ▷Increment inversions by # of elems in left array
      i += 1                                         ▷if it is smaller or if right array is filled already
    else
      res[k] = right_sorted[j]
      j += 1
    end if
    k += 1
  end while
  return res, inv + left_invs + right_invs
end function
```

---

The recursion relation is still

$$T(n) = 2 \cdot T(n/2) + O(1) + O(n) \implies O(n \log n) \quad (21)$$

---

<sup>a</sup>Also known as Kendall-Tau distance in statistics.

## 3.2 Selection

The next problem is a generalization of finding the median of an array, which we present can be done in  $O(n)$  time *on average*.

### Algorithm 3.3 (Select $k$ th smallest element from unsorted array)

---

**Algorithm 3** Select  $A(k, n)$ .

---

**Require:**

---

```
function SELECT(x)
end function
```

---

## 3.3 Multiplication and Matrix Multiplication

### 3.3.1 Karatsuba Algorithm

### 3.3.2 Strassen Algorithm

## 3.4 Fast Fourier Transform

## 4 Hashing and Probabilistic Algorithms

### 4.1 Modulo Operations

### 4.2 Hashing

### 4.3 Primality Testing

## 5 Dynamic Programming

### 5.1 Longest Increasing Subsequence

### 5.2 Knapsack

## 6 Graphs

A huge portion of problems can be solved by representing as a *graph* data structure. In here, we will explore various problems that can be solved through *graph algorithms*.

### 6.1 Representations and Properties

All graphs consist of a set of vertices/nodes  $V$  and edges  $E$ . This tuple is what makes up a graph. We denote  $|V| = n, |E| = m$ .

#### Definition 6.1 (Undirected Graphs)

An **undirected graph**  $G(V, E)$  is a tuple, where  $V = \{v_1, \dots, v_n\}$  is the vertex set and  $E = \{\{v_i, v_j\}\}$  is the edge set (note that it is a set of sets!).

1. The **degree**  $d_v$  of a vertex  $v$  is the number of edges incident to it.
2. A **path** is a sequence of vertices where adjacent vertices are connected by a path in  $E$ . It's **length** is the number of edges in the path.
3. A **cycle/circuit** is a path that has the same start and end.
4. A graph is **connected** if for every pair of vertices  $e_i, e_j \in E$ , there is a path from  $e_i$  to  $e_j$ .

5. A **connected component** is a maximal subset of connected vertices.

### Definition 6.2 (Directed Graph)

A **directed graph**  $G(V, E)$  is a tuple, where  $V = \{v_1, \dots, v_n\}$  is the vertex set and  $E = \{(v_i, v_j)\}$  is the edge set (note that it is a set of tuples, so  $(i, j) \neq (j, i)$ ).

1. The **in/out degree**  $d_{v,i}, d_{v,o}$  of a vertex  $v$  is the number of edges going in to or out from  $v$ .
2. A **path** is a sequence of vertices where adjacent vertices are connected by a path in  $E$ . It's **length** is the number of edges in the path.
3. A **cycle/circuit** is a path that has the same start and end.
4. A directed graph is **strongly connected** if for every pair of vertices  $e_i, e_j \in E$ , there is a path from  $e_i$  to  $e_j$ .<sup>a</sup>
5. A **strongly connected component** is a maximal subset of connected vertices.

In fact, from these definitions alone, we can solve an ancient puzzle called *the Bridges of Königsberg*. Euler, in trying to solve this problem, had invented graph theory.

### Example 6.1 (Bridges of Königsberg)

Is there a way to walk that crosses each bridge *exactly* once?

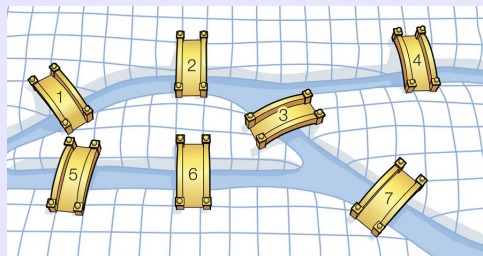


Figure 1: Bridges of Königsberg

It can be decomposed into this undirected graph.

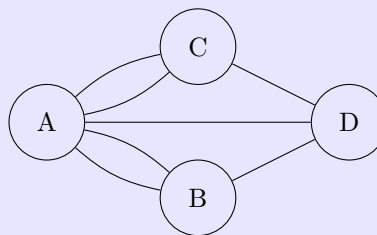


Figure 2: Graph representation.

Euler's observation is that except for start and end points, a walk leaves any vertex by different edge than the incoming edge. Therefore, the degree (number of edges incident on it) must have an even number, so all but 2 vertices must have an even degree. Since every vertex has an odd degree, there is no way of doing it.

In addition to the *adjacency list* representation, another way in which we represent a directed graph is through *adjacency matrices*.

<sup>a</sup>Obviously, a connected undirected graph is also strongly connected.

**Definition 6.3 (Adjacency Matrix)**

In a finite directed graph  $(V, E)$ , we can construct a bijection from  $V$  to the natural numbers and so we label each element in  $V$  with  $i \in \mathbb{N}$ . Then, we can construct a matrix  $A$  such that

$$A_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{if } (i, j) \notin E \end{cases} \quad (22)$$

While the adjacency matrix does have its advantages and has a cleaner form, usually in sparse graphs this is memory inefficient due to there being an overwhelming number of 0s.

**Definition 6.4 (Trees)**

An undirected graph  $G(V, E)$  is a **tree** if

1.  $G$  is connected.
2.  $G$  has no cycles.<sup>a</sup>

Removing the first requirement gives us the definition of a **forest**, which is a collection of trees. Conversely, if  $G(V, E)$  is connected and  $|E| = n - 1$ , then  $G$  is a tree.

**Theorem 6.1 (Properties of Trees)**

If  $G(V, E)$  is a tree, then

1. There exists a  $v \in V$  s.t.  $d_v = 1$ , called a **leaf node**.
2.  $|E| = |V| - 1 = n - 1$ .

**Proof.**

The outlines are quite intuitive.

1. There must be some leaf node since if there wasn't, then we would have a cycle. We can use proof by contradiction.
2. We can use proof by induction. We start off with one vertex and to construct a tree, we must add one edge and one vertex at every step, keeping this invariant.

## 6.2 Exploration

Given two  $v, s \in V$  either directed or undirected, how can we find the shortest path from  $v$  to  $s$ ? We can do with either with DFS or BFS.

**Definition 6.5 (DFS)**

The recursive algorithm is

```

1  visited = set()
2  def dfs(start):
3      if start not in visited:
4          visited.add(start)
5          # do something
6          neighbors = ...
7          for neighbor in neighbors:
8              dfs(neighbor)
```

<sup>a</sup>This makes sense, since to get back to a previous vertex you must backtrack.

The iterative algorithm uses a stack, which mirrors the function call stack.

```
1  visited = set()
2
3  def dfs(start):
4      toExplore = []
5      current = start;
6      toExplore.append(current)
7      visited.add(current)
8      while toExplore:
9          current = toExplore.pop()
10         # Do something
11         neighbors = ...
12         for neighbor in neighbors:
13             if neighbor not in visited:
14                 visited.add(neighbor)
15                 toExplore.append(neighbor)
```

### Theorem 6.2 (Runtime of DFS)

The runtime of DFS is  $O(n + m)$ .

**Proof.**

### Definition 6.6 (BFS)

The iterative version is shown.<sup>a</sup>

```
1  visited = set()
2  def bfs(start):
3      toExplore = collections.deque()
4      current = start;
5      toExplore.append(current)
6      visited.add(current)
7      while toExplore:
8          current = toExplore.popleft()
9          # Do something
10         neighbors = ...
11         for neighbor in neighbors:
12             if neighbor not in visited:
13                 visited.add(neighbor)
14                 toExplore.append(neighbor)
```

### Theorem 6.3 (Runtime of BFS)

The runtime of BFS is  $O(n + m)$ .

<sup>a</sup>The recursive version of BFS is very nontrivial.

**Proof.**

To get the running time, we know that each vertex is popped only once from the queue, giving us  $O(n)$ . For each pop, we are exploring all the neighbors of  $V$ .

$$O\left(\sum_{v \in V} |\text{neighbors of } v| + 1\right) = O\left(\sum_{v \in V} d_v + 1\right) \quad (23)$$

$$= O(2|E| + |V|) = O(m + n) \quad (24)$$

which is linear in input size!

The more straightforward application is in reachability.

**Example 6.2 (Reachability)**

Given a directed graph and a node  $v$ , find all nodes that are reachable from  $v$ .

**Exercise 6.1 ()**

Prove that in any connected undirected graph  $G = (V, E)$  there is a vertex  $v \in V$  s.t.  $G$  remains connected after removing  $v$ .

**Proof.**

Let  $u$  be such a leaf node of  $T$ , and let  $G'$  be the subgraph of  $G$  resulting by removing  $u$  and its incident edges from  $G$ . For sake of contradiction,<sup>a</sup> suppose  $G'$  has more than one connected component. Let  $C$  be a connected component in  $G'$  that does not contain  $s$ , the root of the BFS tree  $T$ . Since  $G$  was connected before the removal of  $u$ , it must be that every path from  $s$  to any vertex  $v$  in  $C$  includes  $u$  (otherwise there would remain a path from  $s$  to  $v$  in  $G'$  and  $s$  would be in  $C$ ). Then  $u$  is the only vertex not in  $S$  with edges to vertices in  $S$ , so all vertices in  $C$  must be “visited” during BFS only after visiting  $u$ . Furthermore, the vertices of  $S$  must be in the subtree of  $T$  rooted at  $u$ . But  $u$  is a leaf, which is a contradiction.

**Exercise 6.2 ()**

Two parts.

1. Give an example of a strongly connected directed graph  $G = (V, E)$  s.t. that every  $v \in V$ , removing  $v$  from  $G$  gives a directed graph that is not strongly connected.
2. In an undirected graph with exactly two connected components, it is always possible to make the graph connected by adding only one edge. Give an example of a directed graph with two strongly connected components such that no addition of one edge can make the graph strongly connected.

**Proof.**

Listed.

1. A graph whose edges form a cycle, having at least three nodes.
2. Two strongly connected components with no edges between them.

<sup>a</sup>We provide an alternative direct proof as follows: Since  $G$  is given to be connected,  $T$  contains all vertices of  $G$ . Let  $T'$  be the BFS tree minus  $u$  and its single incident edge connecting it to its parent in  $T$ . Since  $u$  is a leaf,  $T'$  remains a connected tree with all other vertices of  $G$ . The edges of  $T'$  exist in  $G'$ , so  $G'$  is connected.

### 6.3 Directed Acyclic Graphs and Topological Sorting

#### Definition 6.7 (Directed Acyclic Graph)

A DAG is a directed graph that has no cycles. Note that a DAG must have a node that has no in-edges.

To determine if a graph is a DAG, then we can brute force it by taking a node  $s \in V$ , running DFS/BFS, and if a neighbor is already in visited, return False. Then go through this for all starting nodes  $s \in V$ . This again has quadratic runtime. Can we do better? This introduces us to topological sorting.

It may be helpful to take a graph  $G(V, E)$  and induce some partial order on the set of nodes  $V$  based off of  $E$ . It turns out that we can do this for a specific type of graph.

#### Definition 6.8 (Topological Sort)

Given a directed acyclic graph (DAG), a linear ordering of vertices such that for every directed edge  $u - v$ , vertex  $u$  comes before  $v$  in the ordering is called a **topological sort**. It satisfies the facts:

1. The first vertex must have an in-degree of 0.
2. A topological sort is not unique.

#### Example 6.3 (Simple Topological Sort)

The graph below

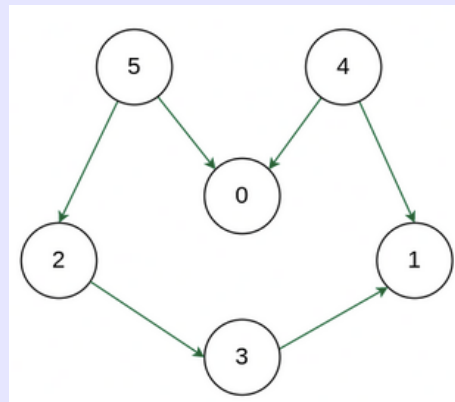


Figure 3

can have the two (not exhaustive) topological sortings.

1.  $[5, 4, 2, 3, 1, 0]$
2.  $[4, 5, 2, 3, 1, 0]$

To determine if a graph is a DAG, note the following theorem.

#### Theorem 6.4 (Topological Order and DAGs)

$G$  has a topological order if and only if it is a DAG.

**Proof.**

To prove that a DAG has a topological order, we use induction. Pick a  $v$  such that its indegree is 0. Then, delete  $v$ , and therefore  $G \setminus v$  is also a DAG with a topological order since we are only deleting edges. We keep going.

Therefore, if we can successfully topologically sort, we know it is a DAG. So we can kill two birds with one stone. Let's see how this is implemented. We can do it iteratively and recursively (the proof above should hint that this can be recursive).

**Algorithm 6.1 (Iterative Topological Sort, Determine If Graph is DAG)**

The general idea is that we first find the node that has 0 in-degree. From here, we can do DFS, and when we run out of neighbors to explore, then we push this into a queue. This is essentially a post-order traversal, where at the end are going to be the end nodes with no more neighbors, and the node we started from will be added last. Then we loop through and do this for all start nodes. We first need a slightly modified form of DFS.

---

**Require:** Nodes  $V$ , adjacency list  $E$

```

1: visited  $\leftarrow$  set()
2: res  $\leftarrow$  stack()
3: is_acyclic  $\leftarrow$  True
4: function DFS( $v \in V$ )
5:   if  $v \neq$  visited then
6:     add  $v$  to visited
7:      $N_v \leftarrow$  neighbors of  $v$ 
8:     for  $n \in N_v$  do
9:       if  $n \in$  visited then
10:        is_acyclic  $\leftarrow$  False
11:      end if
12:      DFS( $n$ )
13:    end for
14:    push  $v$  onto res
15:  end if
16: end function
17:
18: function TOPOLOGICALSORT( $V, E$ )
19:   for  $v \in V$  do
20:     DFS( $v$ )
21:   end for
22:   if ! is_acyclic then
23:     return False
24:   end if
25:   return reversed of res
26: end function

```

---

Note that this runtime is  $O(|V| + |E|)$  since we are just running DFS with a constant amount of work on top of each call.



**Algorithm 6.2 (Recursive Topological Sort)**

We want to see that while  $G$  is nonempty, we want to find the  $v \in V$  such that it has indegree 0. Then place  $v$  next in order, and then delete  $v$  and all edges out of  $v$ . The problem is finding which vertex has indegree 0 (if we brute force it by looking through all remaining nodes and edges, you have quadratic runtime). To do this fast, the idea is

1. initially scan over all edges to store the indegrees of every node to a list `indeg`.
2. store all nodes with indegree 0 to a queue.
3. Run through the queue, and during each loop, when we remove a node, we look at all of its out-nodes  $s$  and decrement `indeg[s]`. If `indeg[s] = 0`, then add it to the queue.

---

**Require:** Nodes  $V$ , Edges  $E$

```

1:  $q \leftarrow \text{queue}()$ 
2:  $\text{indeg} \leftarrow \text{list}()$ 
3:  $\text{visited} \leftarrow 0$ 
4: function RECUR( $x$ )
5:   initialize the  $\text{indeg}$  and  $q$ 
6:   while  $q$  is nonempty do
7:      $v \leftarrow \text{pop}(q)$ 
8:      $\text{visited} += 1$ 
9:     for each  $w \in E[v]$  do
10:       $\text{indeg}[w] -= 1$ 
11:      if  $\text{indeg}[w] = 0$  then
12:        push  $w$  into  $q$ 
13:      end if
14:    end for
15:  end while
16:  if  $\text{visited} \neq |V|$  then
17:    return False
18:  end if
19:  return True
20: end function

```

---

Notice that the inner for loop is  $O(d(v) + 1)$ , while we run over all  $n$ . So really, we are doing  $O(n(d(v) + 1)) = O(m + n)$ , where the plus  $n$  comes from the constant work we are doing for each node. Note that if we have a non-DAG, then at some point the queue will be empty but we haven't processed all the vertices, at which point we can declare failure.

To end this, we can make a general statement about all directed graphs.

**Theorem 6.5 ()**

Every directed graph is a DAG of strongly connected components (SCC).

This gives us a way to represent a directed graph with a collection of DAGs.<sup>2</sup> An extension of topological sort is making a *BFS tree*, which partitions a graph into layers that represent the number of steps required to go from a source vertex to a node.

---

<sup>2</sup>In fact, this Kosaraju's algorithm, can be done in linear time, though it is highly nontrivial.

**Algorithm 6.3 (BFS Tree)**

To construct a BFS tree, we just need to slightly modify the original BFS code.

---

**Require:** Nodes  $V$ , adjacency list  $E$

```

1: visited = set()
2: layers = {v : 0 | v ∈ V}
3: function BFS(start)
4:   layer ← 0
5:   toExplore ← queue()
6:   add (start, layer) to toExplore
7:   add start to visited
8:   while toExplore do
9:     curr, layer = pop from toExplore
10:    layers[curr] = layer
11:    for n ∈ neighbors of curr do
12:      if n ∉ visited then
13:        add n to visited
14:        add (n, layer + 1) to toExplore
15:      end if
16:    end for
17:  end while
18: end function

```

---

This is simply BFS with constant extra work so it is  $O(n + m)$ .

So, a BFS tree is really just another way to topologically sort. Note the following properties.

1. In a directed graph, no nodes can jump from layer  $i$  to layers  $j > i + 1$ , since if it could, then it would be in layer  $i + 1$ . However, nodes can jump from layer  $j$  back to any layer  $i < j$ , even skipping layers.
2. In a directed graph, going forward is the same as going back, so nodes can jump at most one layer forwards or backwards.

**Exercise 6.3 (DPV 3.16)**

Suppose a CS curriculum consists of  $n$  courses, all of them mandatory. The prerequisite graph  $G = (V, E)$  has a node for each course, and an edge from course  $v$  to course  $w$  if and only if  $v$  is a prerequisite for  $w$ . Note this is a directed acyclic graph (DAG). In order for a student to take a course  $w$  with prerequisite  $v$ , they must take  $v$  in an earlier semester. Find an algorithm that works directly with this graph representation, and computes the minimum number of semesters necessary to complete the curriculum, under the assumption any number of courses can be taken in one semester. The running time of your algorithm should be  $O(n + m)$ , where  $n$  and  $m$  are the numbers of vertices and edges in  $G$ , respectively.

**Proof.**

For each vertex, we want to find the longest path leading to it: if there is a path leading to a node, then all of the courses in the path should be taken sequentially. Perform a topological sort of  $G$ 's nodes and label them 1 through  $n$ . Then, we go through the nodes in the resulting topological order. For each vertex, we assign the minimum number of semesters required to take it: if there are no prerequisites, we assign 1, and if there are prerequisites, we assign 1 plus the maximum value assigned to its prerequisite nodes.

*Implementation details:* If the input is in adjacency list format, then we do not have access to the *incoming* edges to a node (its prerequisites). By exploring the entire graph with BFS calls, we can compute the list of incoming edges to every vertex in  $O(n + m)$  time. These details are not required. If the input is in adjacency matrix format, for each node it takes  $O(n)$  time to find its incoming edges, so the total running time is  $O(n^2)$ .

#### Exercise 6.4 (DPV 3.22)

Give an efficient algorithm that takes as input a directed graph  $G = (V, E)$ , and determines whether or not there is a vertex  $s \in V$  from which all other vertices are reachable.

#### Proof.

We first build the DAG representation of the SCCs of  $G$  in  $O(m + n)$  time, as described in lecture. This graph is a DAG where each SCC of  $G$  is represented by a single node. We return true if this DAG has exactly one node with no incoming edges (i.e., exactly one source node), and return false otherwise.

*Correctness.* Let  $u$  be a vertex in an SCC that is a source node in the DAG representation. If there is a path in  $G$  from a vertex  $v$  not in the SCC of  $u$  to  $u$ , then there must be an edge (corresponding to an edge in this path) into the SCC of  $u$  in the DAG, which contradicts that the node has no incoming edges. Thus, if there are two source SCCs in the DAG, no vertex of  $G$  can reach all vertices; in particular, no vertex can reach the vertices in both of the source SCCs. Thus we correctly return false if there are multiple source SCCs.

On the other hand, if there is a single source SCC in the DAG, we claim that every vertex in the SCC can reach every other vertex in  $G$ , in which case our algorithm correctly returns true. Every other SCC in the DAG is not a source, so it has an incoming edge.<sup>a</sup> Consider starting at an SCC in the DAG, picking an incoming edge to the SCC, and then repeating this process from the SCC from which the edge was leaving. This process stops when we reach an SCC without incoming edges. In this case there is exactly one source SCC, so this process will arrive at the single source SCC when starting from any SCC in the DAG. This implies there is a path from the unique source SCC to every other SCC in the DAG, and thus every vertex of the source SCC can reach all vertices in all SCCs of  $G$ ; that is, all vertices of  $G$ .

#### Exercise 6.5 (DPV 3.19)

You are given a binary tree  $T = (V, E)$  with designated root node with  $n = |V|$  vertices which are the integers from 1 to  $n$ . In addition, there is an array  $x[1..n]$  of values where  $x[u]$  is the value of vertex  $u \in V$ . Define a new array  $z[1..n]$  where, for each  $u \in V$ ,

$$z[u] = \max\{x[v] \mid v \text{ is a descendant of } u\} \quad (25)$$

That is,  $z[u]$  is the maximum  $x$ -value of all vertices in the subtree of  $T$  rooted at  $u$ . Note that, by definition, any node is a descendant of itself. Describe an  $O(n)$ -time algorithm which calculates the entire  $z$ -array.

#### Proof.

We propose the following recursive algorithm performs a *postorder* traversal of the tree and populates the values of the  $z$ -array in the process:

```
1 computeZ(u):
```

<sup>a</sup>The following argument can be made formal with induction.

```

2   maxVal = x[u]
3   if u.left is not null:
4       computeZ(u.left) # compute z for all descendants of u.left
5       maxVal = max(z[u.left], maxVal)
6   if u.right is not null:
7       computeZ(u.right) # computes z for all descendants of u.right
8       maxVal = max(z[u.right], maxVal)
9   z[u] = maxVal

```

We initially call `computeZ` on the root node of  $T$ . The algorithm takes  $O(1)$  time per node, which is  $O(n)$  overall<sup>a</sup>

### Exercise 6.6 ()

Your data center needs to run a number of jobs (compute requests) numbered  $1, 2, \dots, n$ . These are specified in a list of  $m$  tuples where  $(i, k)$  means that job  $i$  must be completed before job  $k$  can run. A given job may have multiple dependencies; for example, you might have constraints  $(1, 4), (2, 4), (3, 4)$  that all of jobs 1, 2, and 3 must be completed before 4 can run.

1. Describe an  $O(n + m)$  runtime algorithm that determines whether it is possible to execute all of the jobs, and if so, determines a valid order in which to execute the jobs one at a time. *Hint. How to relate SCCs to cycles?*
2. Suppose you have  $n$  identical servers (so that if there were no constraints you could simply run each job on a separate server). Suppose every job has the same runtime  $R$ . Describe an  $O(n + m)$  runtime algorithm to compute the total runtime that will be necessary to run all of the jobs in a valid order.

### Proof.

For this question we define a graph  $G = (V, E)$  where there is a vertex for every job  $1, \dots, n$  and an edge from  $i$  to  $k$  for every listed dependency (where  $k$  depends on  $i$ ).

1. We note that a sequence of jobs can be executed if and only if there is no circular dependency. In the language of graphs, this requires that  $G$  be free of cycles. To this end, it suffices to propose an algorithm that runs in  $\mathcal{O}(m + n)$ . We will use Kosaraju's SCC algorithm. We prove the following claim:

Any SCC with  $\geq 2$  vertices contain a cycle.

To see this, consider any distinct  $u, v$  in this SCC. Let  $p_{u,v}$  and  $p_{v,u}$  be the paths from  $u$  to  $v$  and backwards, respectively. Now concatenate the paths and get a walk that starts from  $u$  and ends at  $u$ . Note that each vertex appears at most once in  $p_{u,v}$  and in  $p_{v,u}$ , so in the combined walk, it appears at most twice. Consider the set of vertices that are revisited in this walk — clearly,  $u$  is one of them and is the latest one to be revisited. There must exist a vertex  $w$  that was the *first* to be revisited. Then, the section of the walk between the two visits of  $w$  form a cycle by definition: it starts from  $w$ , ends at  $w$ , and does not repeat any other vertices. This proves the claim. And to go back to our problem, the following are equivalent:

- (a) jobs can be executed
  - (b) no cycles in  $G$
  - (c) each SCC obtained from Kosaraju is a singleton
2. We first run the algorithm from part (a) to check if it is possible and to find a valid order of the jobs if so. Then define an array  $L$  of length  $n$ . We will compute  $L[k]$  as the length of the longest dependency chain prior to  $k$ . Loop over the  $k$  jobs in topological order. For each, compute  $L[k] = 0$  if  $k$  has no dependencies, or  $L[k] = 1 + \max_{(i,k)} L[i]$  otherwise. Finally, return

<sup>a</sup>This algorithm can also be described as a modified version of the so-called *depth-first search* (DFS) graph traversal algorithm, which is different from BFS.

$$\max_k L[k].$$

## 6.4 Bipartite Graphs

Now we shall see a further application of BFS trees.

### Definition 6.9 (Bipartite Graph)

A **bipartite graph** is an undirected graph  $G(V, E)$  where we can partition  $V = L \sqcup R$  such that for all  $e = \{u, v\} \in E$ , we have  $u \in L, v \in R$ .

We would like to devise some method to determine if an arbitrary graph is bipartite.

### Theorem 6.6 ()

$G$  is bipartite if and only if all cycles in  $G$  are even length.

### Proof.

Proving ( $\implies$ ) is quite easy since if we suppose  $G$  has an odd length cycle, then we start packing vertices of a cycle into  $L, R$ , but by the time we came back to the start, we are forced to pack it into the wrong partition!

The converse is quite hard to prove, and we'll take it at face value.

Now in practice, how would we determine if all cycles are even length? This is where BFS shines.

### Algorithm 6.4 (Determine Bipartite On All Cycles of Even Length)

The general idea is we first run BFS on the graph starting at  $s \in V$ , which divides it up into layers  $L_1, \dots, L_l$  representing the shortest path from  $s$ . Then for each layer  $L_i \subset V$ , we check if there are connections between two vertices  $x, y \in L_i$ . If there are connections, then this is not bipartite. If there are none, then this is bipartite since we can then color it.

---

```

Require: Nodes  $V$ , adjacency list  $E$ 
1: visited = set()
2: layers = { $v : 0 \mid v \in V$ }
3: function BFS(start)
4:   layer  $\leftarrow 0$ 
5:   toExplore  $\leftarrow$  queue()
6:   add (start, layer) to toExplore
7:   add start to visited
8:   while toExplore do
9:     curr, layer = pop from toExplore
10:    layers[curr] = layer
11:    for  $n \in$  neighbors of curr do
12:      if  $n \notin$  visited then
13:        add  $n$  to visited
14:        add ( $n, \text{layer} + 1$ ) to toExplore
15:      end if
16:    end for
17:  end while
18: end function
19:
20: function BIPARTITE( $V, E$ )
21:   BFS( $v$ ) for some  $v \in V$ 
22:   for  $(u, v) \in E$  do
23:     if layers[ $u$ ] == layers[ $v$ ] then
24:       return False
25:     end if
26:   end for
27:   return True
28: end function

```

---

Therefore, we run BFS, which is  $O(n + m)$ , and then to compare the edges, it is  $O(m)$ .

Bipartiteness is actually a special case of *coloring problems*. Given a graph with  $k$  colors, can I color it so that every neighbor has a different color than the original node? It may seem like at first glance that we can do the same method and look at the layers again, but it turns out that 3-coloring is hard. More specifically it is an NP-complete problem, which colloquially means that there isn't much of a better way than a brute-force solution. However, it turns out that according to the *4 color theorem*, any map can be colored with 4 colors.

## 6.5 Strongly Connected Graphs

Now how do we find out if a directed graph is strongly connected? The straightforward solution would be to take each vertex  $v \in V$ , run BFS to find the set of vertices reachable from  $v$ , and do this for every vertex. The total running time is  $O(n(n + m))$ , which is quadratic. Note that for an undirected graph this is trivial since we just run DFS/BFS once.

### Theorem 6.7 ()

$G$  is strongly connected if and only if for any  $v \in V$ ,

1. all of  $V$  is reachable from  $v$ .
2.  $v$  is reachable from any  $s \in V$

**Algorithm 6.5 (Determine if Graph is Strongly Connected)**

Using the theorem above, we can run BFS/DFS twice: one on the original graph and one on the reversed graph, consisting of all edges directed in the opposite direction.

---

**Require:** Nodes  $V$ , Adjacency list  $E$

```

1: function STRONGLYCONNECTED( $s \in V$ )
2:   visited  $\leftarrow$  set()
3:   BFS( $s$ )
4:   if visited  $\neq V$  then
5:     return False
6:   end if
7:   visited  $\leftarrow$  set()
8:   reverse all edges in  $E$ 
9:   BFS( $s$ )
10:  if visited  $\neq V$  then
11:    return False
12:  end if
13:  return True
14: end function

```

---

The running time is just running BFS twice plus the time to reverse the edges, so it is  $O(n + m)$ .

**6.6 Shortest Path**

In the shortest path, you are given a *weighted* (positive integer) directed graph and your goal is to find a path from  $s$  to  $t$  with the smallest length. This is where we use Dijkstra's. What we can do to brute force is just replace a edge with length  $k$  to  $k$  edges of length 1, and we run BFS on this. However, this is not efficient since the weights can be unbounded. This is where we introduce Dijkstra. The following is how it is introduced in class.

**Algorithm 6.6 (General Dijkstra)**

We can keep a temporarily list  $\pi$  of the shortest path we have found so far, and a permanent list **dist** keeping track of all paths we know are for sure the shortest path.

---

**Require:** Graph  $G(E, V)$

```

1:  $S \leftarrow \{s\}$                                  $\triangleright$ set of nodes that we know for sure is shortest
2:  $\text{dist}[s] = 0$  and the rest very large numbers     $\triangleright$ our final list
3:  $\pi[v] = w_{sv}$  for all  $v \in V$                      $\triangleright$ initialize list with neighboring nodes from start  $s$ 
4: function DIJKSTRA( $s$ )
5:    $u = \text{argmin}_{v \notin S} \pi[v]$                      $\triangleright$ Find node having minimum accum weight so far
6:   add  $u$  to  $S$                                      $\triangleright$ This node must be shortest so add to  $S$ 
7:    $\text{dist}[u] = \pi[u]$                                  $\triangleright$ Now we can update its shortest path in dist
8:   for  $v \notin S$  do                                 $\triangleright$ Look at all neighbors of  $u$  and update those not in
9:      $\pi[v] \leftarrow \min\{\pi[v], \text{dist}[u] + w_{uv}\}$      $\triangleright S$  since those in  $S$  are all guaranteed to be shortest
10:  end for
11: end function

```

---

The problem is line 5 above. We don't want this linear search time since it makes the whole thing quadratic, so rather than a list, we can implement a heap, resulting in the code below.

**Algorithm 6.7 (Dijkstra's Algorithm)**

The general idea is to run a graph traversal like BFS but when you reach a new vertex  $v$ , you can store the accumulated time it took to get to  $v$  and store for all neighbors the accumulated time it will take to get to each of those neighbors. If it is less than what we have currently, then we have found a shorter path and we should update this.

---

**Require:** Nodes  $V$ , Edges  $E$

---

```

1: function DIJKSTRA( $s$ )
2:    $\text{dist} \leftarrow$  list of size  $|V|$  with  $+\infty$     ▷Initialize list of big nums representing shortest distances
3:    $\text{dist}[s] \leftarrow 0$                                 ▷The starting node has dist 0
4:    $\text{predecessors} \leftarrow \{v : \text{None} \mid v \in V\}$     ▷predecessors of each node for path tracking
5:    $\text{toExplore} \leftarrow \text{minheap}()$                 ▷A priority queue of (weight, node)
6:   add  $(0, s)$  to  $\text{toExplore}$                         ▷You want to explore this first
7:   while  $\text{toExplore}$  do
8:      $(\text{curr\_dist}, \text{curr\_node}) \leftarrow \text{pop from toExplore}$     ▷pop from toExplore
9:     if  $\text{curr\_dist} > \text{dist}[\text{curr\_node}]$  then          ▷If this distance is greater than what
10:      continue                                         ▷I already have then not worth exploring
11:   end if
12:   for  $\text{neighbor}, \text{weight} \in E[\text{curr\_node}]$  do          ▷Look at each neighbor
13:      $\text{new\_dist} \leftarrow \text{curr\_dist} + \text{weight}$     ▷The distance to getting to neighbor from now
14:     if  $\text{new\_dist} < \text{dist}[\text{neighbor}]$  then          ▷If this new dist is shorter than what we have
15:        $\text{dist}[\text{neighbor}] = \text{new\_dist}$                 ▷Update best distance
16:        $\text{predecessors}[\text{neighbor}] = \text{curr\_node}$     ▷Update its predecessor
17:       push  $(\text{new\_dist}, \text{neighbor})$  onto  $\text{toExplore}$     ▷Should prob explore from here
18:   end if
19: end for
20: end while
21: return  $\text{distances}, \text{predecessors}$ 
22: end function

```

---

You essentially push  $n$  times and pop  $m$  times, and the time per push and pop is  $\log_2(n)$ . Therefore, the total time to push is  $n \log(n)$  and to pop is  $m \log(n)$ , making the total runtime  $O(\log(n)(n+m))$ .

The first example gotten in class ignores the distances and just attempts to modify the distances in the heap itself (through the decrease key operation). This takes  $2 \log_2(n)$ , but if we use a heap with  $d$  children, we can modify the runtime to  $d \log_d(n)$ . Therefore, the total runtime with tunable parameter  $d$  is

$$O((m + nd) \log_d(n)) \quad (26)$$

which can be minimized if we set  $d = m/n$ , so  $O(m \log_{m/n} n)$ , where for dense graphs  $m/n$  is large and so it can behave roughly in linear time  $\Theta(m)$ .

**Exercise 6.7 ()**

Let  $G = (V, E)$  be a weighted strongly connected directed graph with positive edge weights. Let  $v_0$  be a specific vertex. Describe an algorithm that computes the **cost** of the shortest walk between every pair of vertices of  $G$ , with the restriction that each of these walks must pass through  $v_0$  (that is, for every distinct pair  $u, v \in V$ , among all walks from  $u$  to  $v$  that pass through  $v_0$ , compute the cost of the shortest walk). Describe the algorithm, analyze its runtime complexity, and briefly explain (not a formal proof) why it is correct. Try to give an algorithm that runs in  $O(|E| \log(|V|) + |V|^2)$  time. As usual, you may use any algorithm as described in lecture without restating it or arguing for its correctness.



**Proof.**

The high level idea is to decompose any qualifying  $u \rightarrow v$  walk into the combination of two paths  $u \rightarrow v_0 \rightarrow v$ , where we try to minimize the cost of both subpaths. It's easy to compute the minimum cost of  $v_0 \rightarrow v$  for all  $v$ : running Dijkstra once over the graph suffices. The first half,  $u \rightarrow v_0$ , is the nuisance since we need to calculate this quantity for every  $u \in V$ . Solution? Observe that the destination node  $v_0$  is fixed! We flip the direction, define a “reverse graph”  $G^{-1}$  where each edge carries its original weight but points in the other direction. Then, any cheapest  $v_0 \rightarrow u$  path in  $G^{-1}$  would correspond to the cheapest  $u \rightarrow v_0$  path in  $G$ , with matching total costs.

**Exercise 6.8 ()**

Let  $G = (V, E)$  be a directed, weighted graph with  $|V| = n$  and  $|E| = O(n)$  (that is, the graph is sparse). Let  $s$  be a vertex in  $V$ . How quickly can the cost of the following shortest paths be computed under the given conditions? Just note the runtime and be prepared to explain. All of these can be solved using a single call to a shortest-path algorithm if provided the correct input graph (not necessarily the given one).

1. Compute the shortest path distance from some  $s$  to all other vertices in  $G$  under the condition that the weight of every edge is a positive integer  $\leq 10$ .
2. Compute the shortest path distance to a target  $t$  from all possible source vertices  $s$  in a graph with positive edge weights.

**Proof.**

Listed.

1. Since all weights are integer and uniformly bounded, we convert  $G$  into an unweighted graph and apply BFS. Construct unweighted  $G' = (V', E')$  as follows: for each directed edge  $(u \rightarrow v \in E)$ , put a series of dummy nodes between  $u, v$  in  $G'$  so that the distance from  $u$  to  $v$  in  $G'$  is precisely the integer weight  $w(u, v)$  of  $u \rightarrow v$  in  $E$ . Now  $G$  has at most  $10n$  nodes and  $10n$  edges. So BFS runs in  $O(|V'| + |E'|) = O(n)$ .
2. Construct the reversed graph  $G^{-1}$  and run  $\text{Dijkstra}(G^{-1}, t)$ . This finishes in  $\mathcal{O}((m+n) \log n) = O(n \log n)$  time since  $G$  is sparse.

**Exercise 6.9 ()**

Let  $G = (V, E)$  be an undirected, weighted graph with non-negative edge weights. Let vertices  $s, t \in V$  be given. Describe an algorithm that efficiently solves the following questions.

1. Find the shortest/cheapest  $s - t$  walk with an even number of edges.
2. Find the shortest/cheapest  $u - v$  walk with a number of edges of form  $6k + 1, k \in \mathbb{N}$ .

**Proof.**

Listed.

1. The key observation is that as we travel on  $G$ , the number of edges we have travelled along alternates between being odd and even. Furthermore, the very same vertex may correspond to both even and odd: for example if we walked along  $u \rightarrow v \rightarrow w \rightarrow u$ , then initially we travelled for 0 edges, but upon return we travelled a total of 3 edges. We need a way to distinguish them. The solution? Duplicate each vertex into two categories: “odd” and “even.” We construct a new graph  $G' = (V', E')$  by duplicating every vertex  $v \in V$ , labeling one of them as  $v_{\text{odd}}$  and the other  $v_{\text{even}}$ . For each edge  $(u, v) \in E$ , add two edges  $(u_{\text{odd}}, v_{\text{even}})$  and  $(u_{\text{even}}, v_{\text{odd}})$  to  $E'$ , both with the same as  $(u, v) \in E$ . Clearly,  $|V'| = 2|V|$  and  $|E'| = 2|E|$ . What would edges look like in  $G'$ ? By construction, the

two endpoints of an edge in  $G'$  have different subscripts, one with “odd,” the other “even.” This agrees with our previous observation on the original  $G$  that as we walk along the graph, the distance we have so far travelled alternates between even and odd. It follows that, starting from  $s_{\text{even}}$ , a vertex  $v_{\text{even}} \in V'$  (resp.  $v_{\text{odd}}$ ) is only reachable via even (resp. odd) number of edges. On the other hand, also notice that there is a natural correspondence between edges in  $G'$  and  $G$ :  $(u_{\text{odd}}, v_{\text{even}}) \in E'$  corresponds to  $(u, v) \in E$ . This means a *path* in  $G'$  naturally corresponds to a walk in  $G$ , e.g.:

$$u_{\text{even}} \rightarrow v_{\text{odd}} \rightarrow w_{\text{even}} \rightarrow u_{\text{odd}} \rightarrow t_{\text{even}} \quad \text{corresponds to} \quad u \rightarrow v \rightarrow w \rightarrow u \rightarrow t.$$

Combining both observations above, there exists an  $s - t$  walk in  $G$  with an even number of edges if and only if there is a path in  $G'$  from  $s_{\text{even}}$  to  $t_{\text{even}}$ . The rest is simple: run a pathfinding algorithm on  $G'$ . The weights are non-negative, so we use Dijkstra’s algorithm.

Total runtime? Time to construct  $G'$  involves  $|V'| = 2|V|$  vertices and  $|E'| = 2|E|$  edges. This is dominated by running Dijkstra on  $G'$ , which takes  $O((|V'| + |E'|) \log |V'|) = O((|V| + |E|) \log |V|)$  time. Finally, transforming the path in  $G'$  back to a walk in  $G$  takes linear time w.r.t. the path length (one step for each edge), which is bounded by  $O(|E'|)$ . So overall most work is dominated by Dijkstra’s algorithm and the overall algorithm runs in  $O((|V| + |E|) \log |V|)$ .

2. Same idea but make 6 copies of the graph.

### Exercise 6.10 ()

Suppose that in addition to having edge costs  $\{l_e : e \in E\}$ , a graph also has vertex costs  $\{c_v : v \in V\}$ . Now define the cost of a path to be the sum of its edge lengths, *plus* the costs of all vertices on the path. Give an efficient algorithm for finding the minimum cost path from  $s$  to  $t$ . You may assume edge costs and vertex costs are all nonnegative.

### Proof.

Using the generic approach, we can use  $\text{cost}_u(v) = \text{cost}(u) + w(u, v) + c_v$  to solve this problem. Alternatively, for each edge  $(u, v)$  we can update its weight to  $w(u, v) + c_v$  and run Dijkstra on this updated graph, which gives an equivalent mathematical formulation.

## 6.7 Negative Weighted Graphs

Now let’s extend this problem to find the shortest path in negative weighted graphs. Before we think of a solution, we must make sure that there is no cycle that has a negative accumulated path. Otherwise, this problem becomes ill-defined, so we first assume that such a shortest path exists.

At first glance, we may just think of adding  $\min(v)$ , the minimum value to every node so that this now just becomes a regular positive graph and run BFS on it. However, this does not work since we are not adding a constant number over all paths (it is proportional to the number of nodes in the path).

Another way we can think of is just run Dijkstra. However, if it is looking at two paths. We can have  $s \xrightarrow{2} b$  and  $s \xrightarrow{5} a \xrightarrow{-4} b$ . Dijkstra will immediately go to  $b$  thinking that it is the shortest path, since that’s how far it see. So we need to look far into the future. Therefore, after an arbitrarily long path length, you could get a negative length that just kills your accumulator.

Another way we can do is to use dynamic programming.

**Theorem 6.8 (Bellman Equations)**

We write the **Bellman equations**.

$$d[v] = \min_w \{d[w] + l_{wv}\} \quad (27)$$

with  $d[s] = 0$  for the starting vertex. The solution has a unique solution that finds the shortest path from  $s$  to any  $v \in V$ .

**Proof.**

Note that  $d[w] + l_{wv}$  is the length from some path from  $s \mapsto v$  that goes through  $w$ . The minimum of it must be the shortest path over all  $w \in V$ . Suppose the shortest path goes through fixed  $x$ . If there exists a shorter path from  $s \mapsto x$ , then replace  $d[x]$  by this shortest path. Therefore,

$$d[v] = d[x] + l_{xv} \leq d[w] + l_{wv} \implies d[v] = \min_w \{d[w] + l_{wv}\} \quad (28)$$

To prove uniqueness, suppose there are some other solutions  $\pi$  where  $\pi[v] \neq d[v]$  for some  $v$ . But this cannot be the case by definition since  $d[v] \leq \pi[v]$  for all  $v$ .

**Theorem 6.9 ()**

Given the shortest paths, we can lay out this graph like a tree where  $l_{ab} = l_{aa_1} + l_{a_1a_2} + \dots + l_{a_ib}$ .

So how do we actually implement this?

**Algorithm 6.8 (Shortest Path in Possibly Negative Weighted Graph)**


---

---

**Require:** Nodes  $V$ , Edges  $E$

```

1: function SHORTPATH( $V, E$ )
2:    $\text{res} \leftarrow \text{list}(0)$  of large numbers of size  $|V|$ .
3:    $\text{res}[s] = 0$ 
4:    $\text{predecessors} \leftarrow \{v : \text{None} \mid v \in V\}$ 
5:   while  $\exists (u, v)$  s.t.  $\text{res}[v] > \text{res}[u] + l_{uv}$  do
6:      $\text{res}[v] \leftarrow \text{res}[u] + l_{uv}$ 
7:      $\text{predecessor}[v] \leftarrow u$ 
8:   end while
9: end function
```

---

This is guaranteed to converge and stop after a finite number of steps since at every iteration, a path will either

1. get updated from infinity to a path length
2. get reduced from a path length to a shorter path length

And we will have to reach the shortest path length at which point we can't reduce it further.<sup>a</sup>

Computing the runtime is a bit tricky, since we can look at the same edge twice since minimum paths may have been updated in the middle. Therefore this list **res** may reduce very slowly. For example, let the length of each edge  $|l_e| \leq L$ . Then in the worst case,  $\text{res}[s]$  can be initialized to  $(n-1)L$  representing the max path across all nodes, and we can decrease by 1 in each step. So over all nodes, we can decrease so that each  $\text{res}[s]$  becomes  $-(n-1)L$ , meaning that we are doing on the order of  $2n^2L$  iterations. This is too slow, especially for non-distributed settings.

A better way is to not be so random about how we choose the  $(u, v)$  in the while loop. Notice how we can lay out the shortest paths like a tree, so we can work in layers. The next algorithm implements this.

### Algorithm 6.9 (Bellman-Ford Algorithm)

We think of going in rounds indexed by  $t$ , and at every round, we are iterating through all the nodes and updating the shortest path of  $v$  using the shortest path of  $w$  included in all in-neighbors of  $v$ . At most, we will need to update this at most  $n$  times, which will guarantee convergence.

---

**Require:** Nodes  $V$ , Edges  $E$

---

```

1:  $\pi \leftarrow \text{list}(0)$  of large numbers of size  $|V|$ .
2:  $\pi[s] = 0$ 
3: function BELLMANFORD( $x$ )
4:   for  $t = 1, \dots, n - 1$  do
5:     for  $v \in V$  do
6:        $\pi^{(t)}[v] \leftarrow \min_w \{ \pi^{(t-1)}[v], \pi^{(t-1)}[w] + l_{wv} \}$ 
7:     end for
8:   end for
9: end function
```

---

The runtime is easier to see.

1. The step in the inner loop looks over the set of nodes of size  $\text{indeg}(v)$ .
2. Looping over all the nodes in the inner for loop means that we are going over all edges, so  $O(m)$ .
3. The outer for loop goes through  $n - 1$  times, so the total runtime is  $O(nm)$ .

At first glance, this problem seems like it isn't too different from Dijkstra, but there is a 50-year conjecture that this cannot be improved to linear time.

### Exercise 6.11 ()

Let  $G = (V, E)$  be a directed graph with real-valued edge weights, where each vertex is colored in either **red** or **green**. Find the shortest/cheapest  $s - t$  walk such that, not counting  $s$ , the walk visits red vertices for an even number of times and green vertices at least thrice. (Duplicates allowed and will be counted more than once.)

### Proof.

Similar to the last problem in the previous recitation, the key insight lies in constructing a directed graph  $G' = (V', E')$  that captures some additional structures. Based on the constraints, as we walk along a path in  $G$ , there are two things we need to take care of:

- The number of (not necessarily distinct) red vertices we have walked past, and whether this number even or odd (this is called the *parity* of that number), and
- The number of (not necessarily distinct) distinct green vertices we have walked past.

To encode all of the information above, each vertex in  $G'$  will be represented by a "state", or a tuple  $(v, p, g)$  where

- $v \in V$  corresponds to an original vertex in  $G$ ,
- $p \in \{0, 1\}$  (or "even", "odd") represents the parity of the count of red vertices (not necessarily distinct) visited so far, and
- $g \in \{0, 1, 2, 3+\}$  represents the number of times green vertices (not necessarily distinct) have been visited.

Now we will need to consider the conditions under which each of the tuple variable updates. For

---

<sup>a</sup>This algorithm is also called *policy iteration* in reinforcement learning and is analogous to gradient descent.

example, every time we visit a red vertex, the value  $p$  should alternate, and every time we visit a green vertex, the value  $g$  should increase until it becomes  $3+$ . Formally, the state transitions (i.e. edges in  $E'$ ) can be formulated as follows. For each edge  $(u, v)$  in the original graph  $G$ , depending on the colors of  $u$  and  $v$ , we add the following edges, all with the same weight as  $(u, v)$ , to  $E'$ :

- ( $v$  red) For every state  $(u, p, g)$  [a total of 8 such states because  $p \in \{0, 1\}$  and  $g \in \{0, 1, 2, 3+\}$ ], add an edge to the corresponding state  $(v, 1 - p, g)$ . In other words, we flip the parity because we visited one more red vertex, but this does not affect the value of  $g$ .
- ( $v$  green)
- For every state  $(u, p, g)$  with  $g \in \{0, 1, 2\}$ , add a (directed) edge to  $(v, p, g + 1)$  because our green counter increases given  $v$  is green. (Define  $2 + 1$  to be “ $3+$ .”)
  - For states of form  $(u, p, 3+)$ , add a (directed) edge to  $(v, p, 3+)$  because we still fall under the “ $g \geq 3$ ” category after visiting an additional green vertex.

All of our observations on the Recitation #2 graph modeling problem still hold: if we have a path in  $G'$ , we can uniquely recover a well-defined walk in  $G$ . Initially, we want to start from state  $(s, 0, 0)$  because we start from vertex  $s \in V$  and, per the problem, the starting point does not contribute to the red and green count. Our goal is to reach the state  $(t, 0, 3+)$ , which means (i) we arrive at  $t$ , and along the course we have (ii) visited an even number of red vertices and (iii) green vertices  $\geq 3$  times. This is exactly what we want.

How about the runtime? The construction of  $G'$  involves defining  $8|V|$  vertices since  $p$  has 2 possible values and  $g$  has 4, and we need to construct one state for each pair of  $p$  and  $g$ . Similarly, for each  $(u, v)$ , regardless of the color of  $v$ , in both cases we add a total of 8 edges. Therefore  $|E'| = 8|E|$ . Since  $G, G'$  are directed graphs with real-valued weights, we need to run Bellman-Ford, which takes  $\mathcal{O}(|V||E|)$ . Like shown before, other costs (e.g. the one to recover a walk in  $G$  from a path in  $G'$ ) are linear and hence dominated by the pathfinding runtime. So the final complexity is  $\mathcal{O}(|V||E|)$ .

## 6.8 Minimum Spanning Trees

### Definition 6.10 (Spanning Tree)

Given an undirected graph  $G(V, E)$ , a **spanning tree** is a subgraph  $T(V, E' \subset E)$  that is

1. a tree, and
2. spans the graph, i.e. is connected

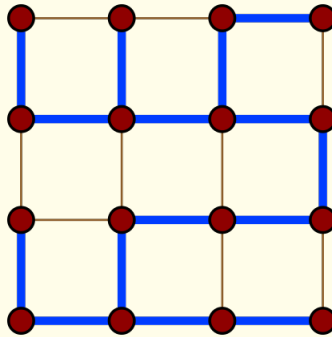


Figure 4: A spanning tree on a  $4 \times 4$  grid graph.

Note that an unconnected graph will never have a spanning tree, but what about a connected graph?

### Theorem 6.10 (Spanning Trees of Connected Graphs)

A connected graph will always have at least one spanning tree, not necessarily unique.

Given a connected undirected weighted graph, we may want to find the **minimum spanning tree (MST)**, i.e. the spanning tree with edges  $E'$  such that the sum of the weights of all  $e \in E'$  is minimized.<sup>3</sup> How do we do this? There are two well-known algorithms to solve this. Prim's and Kruskal's algorithm.

### 6.8.1 Prim's Algorithm with Cuts

Let's try to apply what we already know: Dijkstra. If we run Dijkstra on the graph starting at  $s \in V$ , we can get the shortest path from  $s$  to every other node in the graph. This will give us a tree, but it may not be minimum.

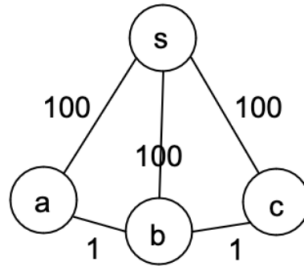


Figure 5: If we run Dijkstra on  $s$ , then our output will be a tree of cost 300, even when the actual MST can be of cost 102 starting from  $a$ .

It may seem like this is just a problem of where we start, but even this is not the case.

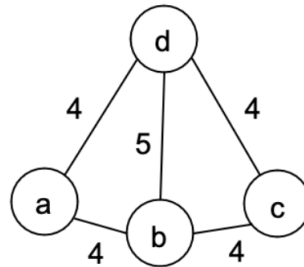


Figure 6: No matter where we start from, we will never output the MST. The MST has cost 12. If we start from  $b$  or  $d$ , we will get a tree of cost 13. If we start from  $a$  or  $c$ , we will get a tree of cost 16.

#### Definition 6.11 (Cuts)

Given graph  $G(V, E)$ , a **cut** is a partitioning of  $V$  into  $(S, V \setminus S)$ . Furthermore, let  $\text{Cut}(S)$  be the number of edges with exactly one endpoint in  $S$  and the other in  $V \setminus S$ .

#### Theorem 6.11 (Cycles and Cuts)

Given cycle  $C \subset E$  in a graph and a cut  $S \subset V$ ,

$$|C \cap \text{Cut}(S)| \quad (29)$$

is even. We can intuit this by visualizing the cycle as a long piece of looped string and a cut is a circle. The intersection between this circle and the string must be even since every time the cycle

<sup>3</sup>An application of this is when we generally want to make sparse graphs. In a datacenter, wires can be expensive, so how I can minimize the length of wires to buy to construct a spanning subgraph?

crosses through the cut, it must return back across the cut to the initial point.

Now time for a bizarre theorem.

### Theorem 6.12 (Cut Property of MSTs)

For all cuts  $S \subset V$  of an undirected graph, the minimum cost edge in  $\text{Cut}(S)$  belongs to the MST. Furthermore, the converse is true: if we take all cuts and find all their minimum cost edges, these edges are precisely the MST! Therefore, an edge  $e \in \text{MST}$  iff  $e$  is a min-cost edge for some cut.

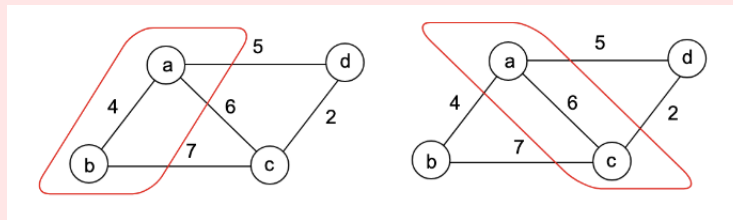


Figure 7: In the left cut, the edges are  $(a, d)$ ,  $(b, c)$ ,  $(a, c)$ . The minimum weight is 5 on the  $(a, d)$  edge, so it must be in the MST. For the right cut,  $(c, d)$  must be in the MST.

The final part is that if we have all edge costs different, then we will have a *unique* MST.

### Proof.

We use a greedy approach and prove by contradiction. Suppose that this is not true, i.e. there exists a cut  $S$  with minimum cost edge  $e$ , and  $e \notin \text{MST}$ . Then, there exists some other edge  $e' \in \text{Cut}(S)$  that is in the MST, since the MST is spanning and it must cross over to connect the whole graph. Well if we just put  $e$  in and take  $e'$  out, we will still have a spanning tree since it connects the left spanning tree to the right spanning tree, and we now have a cheaper tree. So the original cannot be the MST in the first place.

To prove the converse, consider some edge  $e$  in the MST and we must prove that it is the minimum cost edge in some cut. Note that if we take  $e$  out, then it divides the MST into two connected components, and we can just define the cut as these subsets of nodes. So this is in  $\text{Cut}(S)$  for some  $S \subset V$ . We can also prove that this is minimal since if it wasn't the minimum cost edge for some cut, we could have taken it out and inserted a cheaper edge  $e'$  to begin with, getting a cheaper spanning tree.

We can just brute force this logic into an algorithm by going through all possible cuts and adding the minimum cost edge to our MST set. It is clear that a cut is defined by a subset of  $S$ , so really the number of cuts a graph can have is  $2^{|S|-1}$ , which is exponential in  $n$ . However, the minimum spanning tree isn't exponential since it must have  $n - 1$  edges, so there must be many cuts with the same minimum edge.

One way is to start with one vertex  $a$  that contains the minimum cost edge  $(a, b)$  across all edges. This edge must be minimal and must be in the MST. Then we can look at the cut  $S = \{a, b\}$  and look at that cut. We keep doing this, keeping track of the set of edges we need to look at after adding a new node to our cut. So the number of cuts I consider is equal to the number of edges in the spanning tree.

### Algorithm 6.10 (Prim's Algorithm to Find MST)

It turns out that we can modify Dijkstra to solve it.

**Require:** Graph  $G(E, V)$

```

1: function PRIM( $s$ )
2:    $S \leftarrow \{s\}$ 
3:    $\pi[v] \leftarrow$  list of size  $|V|$  of  $+\infty$ 
4:    $\pi[v] = w_{sv}$  for all  $v \in V$ 
5:   while  $S \neq V$  do
6:      $u = \operatorname{argmin}_{v \notin S} \pi[v]$ 
7:      $S \leftarrow S \cup \{u\}$ 
8:     for  $u \notin S$  do
9:        $\pi[u] \leftarrow \min\{\pi[u], w_{wu}\}$ 
10:    end for
11:  end while
12: end function

```

▷Our initial cut  
 ▷ $\pi[u]$  is min cost of getting from  $u$  into the  $S$   
 ▷initialize list with neighboring nodes from start  $s$   
 ▷Find node having minimum cost to reach from  $S$   
 ▷Adding this node to  $S$  to expand our cut  
 ▷Since we expanded  $S$ , our min reach distances  
 ▷must be updated. It can only get shorter  
 ▷through a path from new  $u$ , so compare them

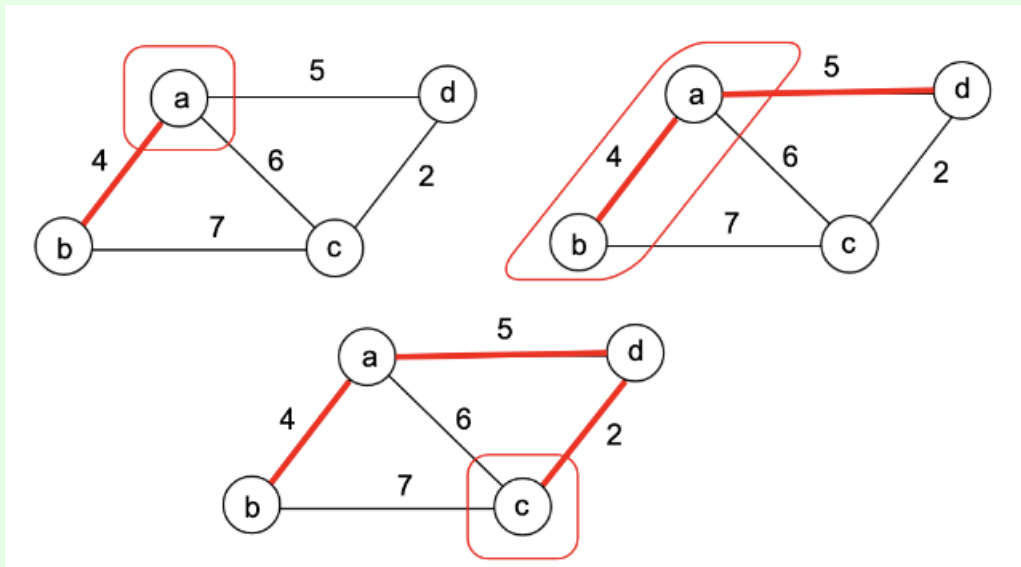


Figure 8: Step by step process of the method we mention above.

#### Algorithm 6.11 (Prim's Algorithm)

However, this is not efficient, so we can introduce a minheap to get the argmin step faster.



---

**Require:** Nodes  $V$ , Edges  $E$

---

```

1: function PRIM( $V, E$ )
2:    $mst \leftarrow []$                                 ▷Initialize mst array to return
3:    $s \leftarrow 0$                                     ▷Choose any starting node
4:    $visited \leftarrow \text{set}()$                         ▷Our expanding set of cuts.
5:    $edges \leftarrow \text{minheap}()$                     ▷The set of low-weight edges that we can explore from
6:    $\text{add}(\text{weight}, s, \text{next\_node})$  for edges in  $E$   ▷Look for edges from  $s$  to expand our cut from.
7:   while  $|visited| < |V|$  do                        ▷Until we have visited all cuts,
8:      $\text{weight}, \text{frm}, \text{to} \leftarrow \text{pop from edges}$   ▷Get the cheapest edge to explore
9:     if  $\text{to} \notin \text{visited}$  then                    ▷If this isn't already in our cut,
10:       $\text{add to} \rightarrow \text{visited}$                     ▷Add it to our cut. From cut
11:       $\text{add}(\text{frm}, \text{to}, \text{weight})$  to  $mst$             ▷property, this must be added to mst
12:      for  $\text{next\_to}, \text{next\_weight} \in E[\text{to}]$  do  ▷After expanding, add newly discovered
13:        if  $\text{next\_to} \notin \text{visited}$  then            ▷edges for future exploration
14:           $\text{push}(\text{next\_weight}, \text{to}, \text{next\_to})$  to  $edges$ 
15:        end if
16:      end for
17:    end if
18:  end while
19:  return  $mst$                                        ▷of form (from, to, weight)
20: end function

```

---

You essentially push  $n$  times and pop  $m$  times, and the time per push and pop is  $\log_2(n)$ . Therefore, the total time to push is  $n \log(n)$  and to pop is  $m \log(n)$ , making the total runtime  $O(\log(n)(n + m))$ .

### 6.8.2 Kruskal's Algorithm

If we were to try and construct this algorithm from scratch, we may take a greedy approach by incrementally adding the minimum cost edge from your cut. However, there is one thing to check: have we entered a cycle? Checking whether the next added node  $a$  completes a cycle in  $S \cup \{a\}$  is nontrivial.

#### Theorem 6.13 (Cycle Property)

For all cycles  $C$ , the max cost edge of  $C$  does not belong to MST.

Therefore, you can take  $S$  and either add to it using the cut property or delete candidates from it using the cycle property. What is the best order to do this in?

#### Algorithm 6.12 (General Kruskal's Algorithm)

This algorithm is faster and can be run faster than  $O(m \log n)$  time. The general idea is that we sort  $e \in E$  in increasing cost, and for each  $e \in E$ , we use either the cut or cycle property to decide whether  $e$  goes in or out.

**Require:** Graph  $G(V, E)$

```

1: function KRUSKAL( $V, E$ )
2:   sort  $E$  in increasing cost
3:    $T \leftarrow \{\}$ 
4:   for  $e \in E$  do
5:     if  $T \cup \{e\}$  does not have cycle then
6:        $T \leftarrow T \cup \{e\}$ 
7:     else
8:       continue
9:     end if
10:  end for
11: end function

```

▷Cut property  
 ▷Cycle property  
 ▷discard  $e$  since from sorting, this edge is heaviest in cycle

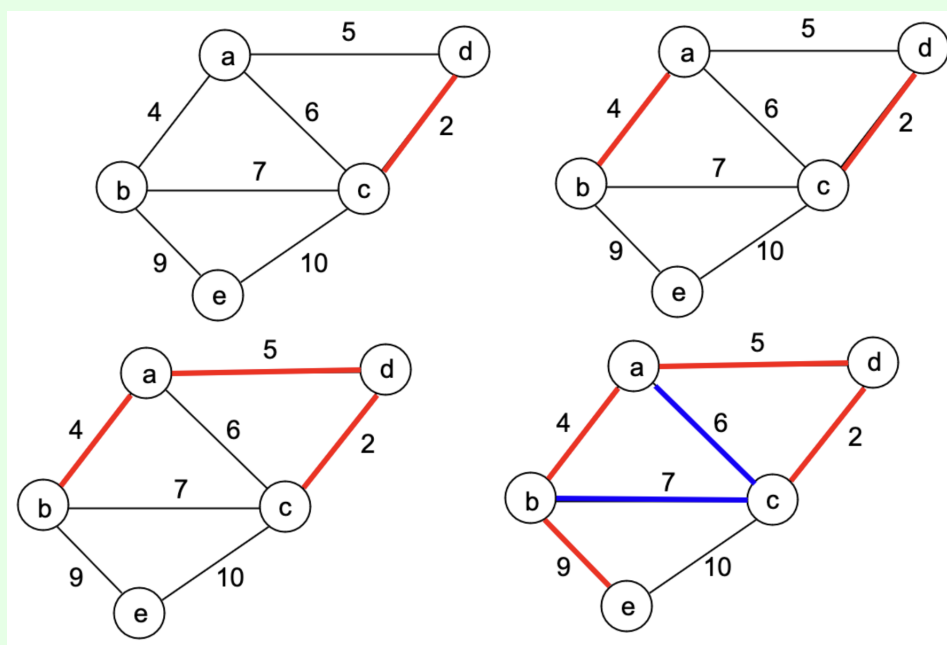


Figure 9: Kruskal's algorithm. In the last step, we see that the next minimum cost edge of 6 and 7 forms a cycle, so we add the edge of length 9.

The way to prove that this is correct is to show that every step you do is correct, known as *structural induction*, either because of one of the two properties. Say that so far, we have some edges in  $V$  which forms a partition of  $V = \sqcup_i T_i$  of disjoint trees (can be trees, one edge, or just single nodes). We are looking at the next biggest edge  $e = (a, b)$ . There are two possibilities.

1. If  $a, b$  are both in a single  $T_i$ , then this forms a cycle and can be thrown away since this is the max cost edge in the cycle by the cycle property.
2. If  $a, b$  connect  $T_i$  and  $T_j$  for  $i \neq j$ , then this edge is in  $\text{Cut}(T_i)$  and is the minimum cost edge since the rest of the edges in  $\text{Cut}(T_i)$  come next in the sorted  $E$ . Therefore this must be included by the cut property.

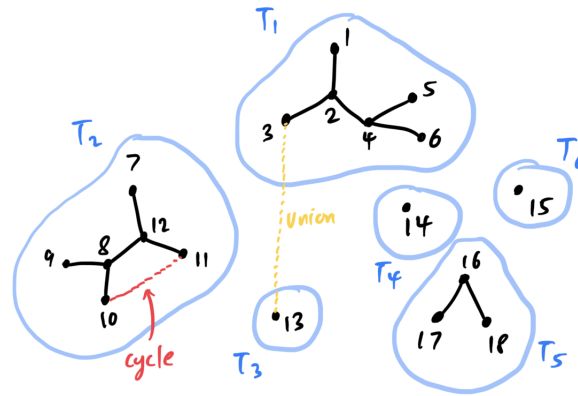


Figure 10: Each  $T_i$  is a component formed by the edges chosen so far. For example,  $T_1 = \{1, 2, 3, 4, 5, 6\}, \dots$ . We can either discard an edge (red) or include an edge (yellow).

The only bottleneck in here is line 5, where we check if  $e$  does not complete a cycle in  $T$ . It will obviously not be efficient to do BFS, construct a tree, and see if there is a loop by checking if two points in the same layer are connected. It may help to decompose this algorithm into two steps: what is being stored and what is being checked?

Note that from our visual, we are really just keeping a set of these points that each make a subtree and connecting them together. How do we efficiently search for which cluster a point is a part of and efficiently merge two clusters? We could use a hashmap but this wouldn't work. We need something like a doubly linked list.

#### Algorithm 6.13 (Kruskal's Algorithm)

The implementation uses the Union-Find data structure. For clarity, we will not elaborate it but will show the full pseudocode.

---

**Require:** Nodes  $V$ . Edges  $E = \{(u, v, w)\}$  where  $u, v \in V$  and  $w > 0$  is a weight.

---

```

function KRUSKAL( $V, E$ )
   $n \leftarrow |V|$ 
  sort  $E$  in increasing order of weights.
  parent  $\leftarrow [0, \dots, n - 1]$ 
  rank  $\leftarrow$  list of 0s of size  $n$ 
  function FIND( $x$ )
    if parent[ $x$ ]  $\neq x$  then
      parent[ $x$ ]  $\leftarrow$  Find(parent[ $x$ ])
    end if
    return parent[ $x$ ]
  end function
  function UNION( $x, y$ )
     $px, py = \text{find}(x), \text{find}(y)$ 
    if  $px = py$  then
      return False
    end if
    if rank[ $px$ ] < rank[ $py$ ] then
      parent[ $px$ ]  $\leftarrow py$ 
    else if rank[ $px$ ] > rank[ $py$ ] then
      parent[ $py$ ]  $\leftarrow px$ 
    else
      parent[ $py$ ]  $\leftarrow px$ 
      rank[ $px$ ]  $\leftarrow$  rank[ $px$ ] + 1
    end if
    return True
  end function
  mst  $\leftarrow []$ 
  for  $u, v, \text{weight} \in \text{edges}$  do
    if union( $u, v$ ) then
      add ( $u, v, \text{weight}$ ) to mst
    end if
    if len(mst) =  $n - 1$  then
      break
    end if
  end for
  return mst
end function

```

▷Needed for Kruskal  
▷Initialize the disjoint cluster each node is in  
▷Path compression

---

To analyze the runtime of this, we define the function.

**Definition 6.12 (Ackerman Function)**

The **Ackerman function** is one of the fastest growing functions known. It is practically infinity.

$$A(m, n) = \begin{cases} n + 1 & \text{if } m = 0 \\ A(m - 1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases} \quad (30)$$

The inverse Ackerman function therefore grows extremely slowly.

If we optimize the steps in Kruskal's algorithm, we can get its runtime to

$$O((m + n) \log^*(n)) \quad (31)$$

which is practically linear.

### 6.8.3 Applications

Here is a way to cluster data, a surprising way to apply MSTs. It is the most widely used application, especially in data science. The problem is that given  $n$  data points  $x_i \in \mathbb{R}^d$  and an integer  $k$ , we want to partition the points into  $k$  groups  $\mathbf{C} = (C_1, \dots, C_k)$  where  $\mathbf{x} = \sqcup_i C_i$ . You want to distances between the points within a group to be small and the distances between groups to be large. We can think of finding the objective which takes every pair of clusters and computes the minimum distance between these clusters, and we want to maximize this distance over all pairs of clusters.

#### Algorithm 6.14 (Single Linkage/Hierarchical Clustering)

The general idea is to take this dense graph, find the MST, and cut off the largest edges from this MST, which will give you  $k$  components. This is the answer. Or really, you can use Kruskal's algorithm and terminate earlier when  $T$  has  $k$  sets/components. Note that as we add edges as we construct our MST, we are merging two clusters into one. So that all you are doing is finding the next pair of closest points and merging the clusters that they are a part of.

---

**Require:** Nodes  $V = \{v_i\} \subset \mathbb{R}^n$

- 1: **function** CLUSTER( $V$ )
  - 2:     Run Kruskal and at each iteration, check if you have  $K$  clusters.
  - 3:     If so, terminate and return the **parent** list.
  - 4: **end function**
- 

#### Theorem 6.14 ()

The algorithm above minimizes the objective function.

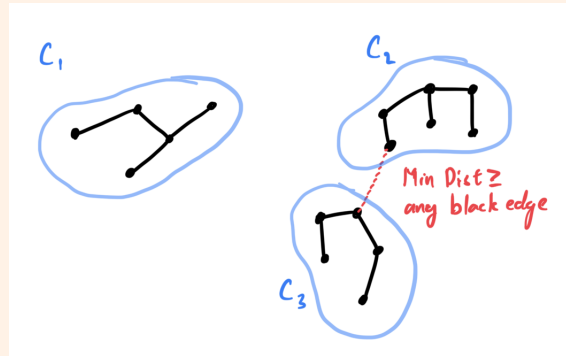
$$\operatorname{argmax}_{\mathbf{C}} \min_{p \in C_i, q \in C_j} \{d(p, q)\} \quad (32)$$

#### Proof.

Let  $\mathbf{C}^*$  be the MST clustering. We claim that for any other clustering  $\mathbf{C} = \{C_1, \dots, C_k\}$ ,

$$\operatorname{mindist}(\mathbf{C}) \leq \operatorname{mindist}(\mathbf{C}^*) \quad (33)$$

Assume that this was not the case, so  $\operatorname{mindist}(\mathbf{C}) > \operatorname{mindist}(\mathbf{C}^*)$  and therefore there exists a  $p, q \in C_i, C_j$  such that  $d(p, q) = \operatorname{mindist}(\mathbf{C}) > \operatorname{mindist}(\mathbf{C}^*)$ . Since this is a different clustering,  $p, q$  must have been in the same cluster  $C_i^*$ . But note that since Kruskal adds edges in increasing length, all edges within a cluster must have length less edges that go across two clusters.



So  $d(p, q)$  must be the less than the length of all edges within a cluster in  $\mathbf{C}^*$ . But all within-cluster edges must be smaller than  $\text{mindist}(\mathbf{C}^*)$ , meaning that  $\text{mindist}(\mathbf{C}^*) > d(p, q)$ , contradicting the fact that is is greater, and we are done.

If you define the distance between two clusters to be the distance between the centroids (mean point), then this is called *average linkage* (min avg distance). If we define the cluster distance as the maximum distance between two points, then it is called *complete linkage* (min max distance). Kruskal's algorithm only worked for the single linkage case but may not work for these additional definitions. This is why there is usually a whole suite of clustering algorithms for a particular problem and we just find out which one fits the data the best. Furthermore, we have done *bottom-up clustering*, where we took individual points to make clusters. In *top-down clustering*, we take the whole set and cut it up into clusters.

## 7 Greedy Algorithms

Greedy algorithms are easy to implement and easy to conceptualize, but they are often not correct. So the only interest in them is proving if they are correct. We have done greedy algorithms like Dijkstra and finding MSTs. Every step, greedy algorithms make an irrevocable decision of what to do and cannot undo it. That is, we cannot backtrack. In fact, some algorithms like Kruskal's algorithm for MSTs is precisely a greedy algorithm.

### Example 7.1 (Interval Scheduling)

Let there be 1 classroom and  $n$  courses, each with a start and end time  $(s_i, e_i)$ . We want to find the largest subset of courses that can be scheduled in a day such that they don't overlap. There are two things we want to do. How do we code this up? How do we show that a greedy algorithm will give a correct answer? Two greedy approaches are as follows.

1. Sort them by the start times, and add them as long as they do not overlap with what you have. This will not work.
2. Sort them by the time interval lengths and keep adding them as long as they do not overlap with what you have. This will not work either.

It turns out that if we sort based on finish time, this will work. So why is this correct? Assume that the first  $k$  decisions of this greedy algorithm are correct. Then, to find the next interval that we will include, the optimal algorithm must choose one that does not overlap with what we have. This is good since we do the same. The second part is that the next added interval must have the smallest end time  $t$  from all viable left intervals. Assume that it was not and that the optimal next interval had end time  $s > t$ . Then, the next interval must start after  $s$ , but this also starts after  $t$ , so we are sacrificing unnecessary extra space. An optimal solution with the endpoint at  $s$  can also be constructed with the same interval ending at  $t$  without anything to lose.

**Algorithm 4** Find Max Classes to Fit into 1 Room**Require:** classes  $C = \{(s_i, e_i)\}_i$  of type `List[tuple(int, int)]`**function** SCHEDULE( $C$ )     $res \leftarrow []$     sort  $C$  by increasing end time    **for**  $s, e \in C$  **do**        **if**  $s < res[-1][-1]$  **then**

continue

**end if**        add  $(s, e)$  to  $res$     **end for**    **return**  $res$ **end function**

▷If start overlaps with previous end time

▷Otherwise add class to final schedule

To get the run time, the sorting takes  $O(n \log n)$  time and iterating is  $O(n)$ , so a total of  $O(n \log n)$  time.

**Example 7.2 (Classroom Scheduling)**

A slightly more practical example is that you have  $n$  classes and you want to minimize the number of classrooms  $m$ . For this, you can really use any order. The general idea is

1. Consider intervals in increasing start time  $s_i$ .
2. If it can be scheduled in an existing classroom, then schedule it there.
3. If not, then open a new classroom.

Even then, we can conduct a line search through all the time periods, incrementing the current number of classes if an incoming time is a start time and decrementing if it is an end time. We report the max  $\Delta$  over all times.

This algorithm is correct since by definition,  $\Delta$  is the lower bound for the number of classrooms needed and the greedy algorithm attains it. The greedy algorithm reports  $\Delta$  since if it reported  $\Delta + 1$  classroom, then at some time  $t$  it opened the  $\Delta + 1$ th classroom. But this is impossible since this must mean that there are  $\Delta + 1$  classrooms concurrently in use, contradicting our assumption that  $\Delta$  is the optimal solution.

**Algorithm 5** Minimizing Number of Rooms With  $n$  Classes (Leetcode 253, Meeting Rooms II)**Require:** classes  $C = \{(s_i, e_i)\}_i$  of type `List[tuple(int, int)]`**function** MEETINGROOMS( $C$ )     $\text{start} \leftarrow$  all start times sorted increasing     $\text{end} \leftarrow$  all end times sorted increasing     $\text{res} \leftarrow 0$      $\text{count} \leftarrow 0$      $s, e \leftarrow 0$     **while**  $s < \text{len}(\text{start})$  **do**        **if**  $\text{start}[s] == \text{end}[e]$  **then**             $s += 1, e += 1$         **else if**  $\text{start}[s] > \text{end}[e]$  **then**             $e += 1, \text{count} -= 1$         **else if**  $\text{start}[s] < \text{end}[e]$  **then**             $s += 1, \text{count} += 1$              $\text{res} = \max(\text{res}, \text{count})$         **end if**    **end while**    **return**  $\text{res}$ **end function**

▷Our max classes count

▷Our curr. classes count

▷pointers for **start**, **end**▷Don't need to check for **end** since▷**start** will always finish faster.

▷One started and one ended, so don't change

▷One ended, so decrement **count**▷One started, so increment **count**▷Update **res** if we got past it

The runtime is  $O(n \log n)$ , which is to sort. However, you don't even need to sort since you can go through all intervals in any order and place it in an existing classroom or open a new classroom.

The next one isn't as trivial, but requires us to devise a custom sorting method by comparing two sequences with a swap difference.

**Example 7.3 (Quiz Problem)**

Suppose you have  $n$  questions with question  $i$  having reward  $r_i$ , but the probability that you get it correct is  $p_i$ . You keep collecting the reward until you get one question wrong, and the game terminates. What order would you answer the questions in?

Suppose we have  $q = (10, 5)$  and  $r = (0.1, 0.2)$ .

1. If you choose the first and then second, then the expected return is

$$10 \cdot 0.1 + 5 \cdot 0.1 \cdot 0.2 = 1.1 \quad (34)$$

2. If you choose the second and then first, then the expected return is

$$5 \cdot 0.2 + 10 \cdot 0.1 \cdot 0.2 = 1.2 \quad (35)$$

Clearly, finding all possibilities is too computationally expensive since it increases exponentially w.r.t.  $n$ . Intuitively, if I have a question with a high reward, I want to answer it first, but if I have a low probability of getting it correctly, then I can't answer future questions if I get it wrong. So I have to balance these two forces. So we want to sort the score of each question with some function  $f(r_i, p_i)$ , which is increasing in both  $r_i$  and  $p_i$ . It is not  $r_i p_i$ , but this is a good start.

Rather, we can take a different approach. Assume that the tuples were sorted in some order, but this was not the optimal order. Then this indicates that we can swap to adjacent elements and it will get a better order. This is really like bubble sort, and now we need to find out the conditions on which it can be improved.

1. If we answer  $q_1 \rightarrow q_2 \rightarrow q_3 \dots$ , our expected reward is

$$\mathbb{E}[R] = r_1 p_1 + r_2 p_1 p_2 + r_3 p_1 p_2 p_3 \quad (36)$$



2. If we swap  $q_2$  and  $q_3$  and answer  $q_1 \rightarrow q_3 \rightarrow q_2 \dots$ , then our expected reward is

$$\mathbb{E}[R] = r_1 p_1 + r_3 p_1 p_3 + r_2 p_1 p_2 p_3 \quad (37)$$

Note that the swap does not affect higher order terms. Doing some algebra, the swap is better iff

$$r_2 p_2 (1 - p_3) < r_3 p_3 (1 - p_2) \implies \frac{r_2 p_2}{1 - p_2} < \frac{r_3 p_3}{1 - p_3} \quad (38)$$

where the final swap saves computational time since we can compute for a single element rather than comparing pairwise elements. So, we sort it (in descending order!) according to these values to maximize this value. Note that the numerator measures your expected reward, while the denominator measures the probability of you screwing up.

Not all greedy problems admit to scoring functions however. This was just one example.

#### Example 7.4 (Minimizing Max Lateness)

Suppose there are  $n$  jobs that all come in at once at time 0, with job  $j$  having length  $p_j$  and deadline  $d_j$ . We want to schedule the order of jobs on one machine that can handle one job at a time, where you must minimize the maximum lateness, where the lateness of job  $i$  is  $\max\{f_i - d_i, 0\} = [f_i - d_i]^+$ . For example, given jobs  $(p, d) = \{(10, 9), (8, 11)\}$ , we can run it in two ways.

1. Running 1 then 2. The first job finishes at 10 and the second at 18. The lateness is  $10 - 9 = 1$  and  $18 - 11 = 7$  respectively, so the maximum lateness is 7.
2. Running 2 then 1. The first job finishes at 8 and the second at 18. The lateness is 0 and  $18 - 9 = 9$ , so the maximum lateness is 9.

Therefore,  $1 \rightarrow 2$  beats  $2 \rightarrow 1$ . Clearly, brute forcing this over  $N!$  jobs orderings is unfeasible, but there is a greedy approach to this. We can

1. schedule jobs in increasing order of deadlines.
2. try to exchange by taking a sequence, swapping it, and computing the score like we did for previous examples.

Both lead to the same score/principle that we should schedule jobs in increasing order of deadlines. Let's prove this by taking jobs  $i$  and  $j$ , where  $d_j \leq d_i$ . Then, if we schedule  $i \rightarrow j$ , then we have

$$l_i = [t + p_i - d_i]^+ \quad (39)$$

$$l_j = [t + p_i + p_j - d_j]^+ \quad (40)$$

with  $l_j$  being the greatest late time. If we did  $j \rightarrow i$ , then we have

$$\hat{l}_j = [t + p_j - d_j]^+ \quad (41)$$

$$\hat{l}_i = [t + p_i + p_j - d_i]^+ \quad (42)$$

with  $\hat{l}_i$  being the greatest late time. But this is a good sacrifice since we can see that both

$$l_j > \hat{l}_j \text{ and } l_j > \hat{l}_i \quad (43)$$

meaning that if we swap, then we will decrease the lateness of  $j$  at the smaller cost of increasing that of  $i$ . Therefore,  $j \rightarrow i$  is better. Note that even though the end times of the second job will always be the same between the two choices, starting with  $j$  will give a later deadline time. So this means that in the optimal solution, we have to place them in this order of deadlines since we can always improve them by swapping.

**Example 7.5 (Gift Wrapping a Convex Hull)**

Given  $n$  points in  $\mathbb{R}^2$ , we want to find the subset of points on the boundary of its convex hull. We can intuit this by taking the bottom most point and imagine “wrapping” a string around the entire region.<sup>a</sup> Here’s a basic fact. If we have a point  $p_1 = (x_1, y_1)$  and we’re looking at  $p_2 = (x_2, y_2)$ , and then  $p_3 = (x_3, y_3)$  shows up, then we can look at the cross product

$$P = (x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1) \quad (44)$$

and if  $P > 0$ , then  $p_3$  is counterclockwise from  $p_2$ . The algorithm is as such.

1. You choose the bottommost point and label it  $p_1$ .
2. You construct a vector pointing to the right and start turning it counterclockwise until it hits the first point. This can be done by iterating through all of the  $n$  points and computing the angle.

$$\tan \theta = \frac{y_i - y_1}{x_i - x_1} \quad (45)$$

and the points with the smallest such tangent value will be the desired point, which will be  $\Theta(n)$ . We set this point to be  $p_2$  and claim that  $p_1, p_2$  is in the convex hull. This is true since all point must lie above the line  $p_1 p_2$ . Now you do the same thing with  $p_2$  starting with a vector facing to the right and rotating it left until it hits the first peg. Once you get to the topmost point, you can start the vector from the right and rotate it counterclockwise, but this is an implementation detail.

If your convex hull has  $L$  vertices, then this algorithm will have  $O(nL)$ , which will be  $O(n^2)$  worst case but if we sample uniformly, then we have  $L \approx \log n$  and on average it is  $O(n \log n)$ .

A lot of geometric algorithms is output sensitive.

**Example 7.6 (Convex Hull with Graham Scan)**

In fact, we can take a different approach and get  $O(n \log n)$  time.

1. You still start with  $p_1$  with the minimum  $y$ -coordinate and we sort in increasing  $\theta_i = \arctan \frac{y_i - y_1}{x_i - x_1}$ .
2. Then it will walk from  $p_i$  to  $p_{i+1}$  and check at  $p_{i+1}$  if it is on the convex hull or not.
  - (a) If you turned counterclockwise to go to  $p_{i+1}$  then it’s good and you’re on the convex hull.
  - (b) If you turned clockwise it is bad, and  $p_i$  is not on the convex hull, so remove  $p_i$ . However,  $p_{i-1}$  could also be clockwise as well, so you must backtrack and keep on removing points until you find a point  $p_j$  such that the previous turn is counterclockwise.

Note that this is implemented as a stack.

This can be calculated in  $O(1)$  by the previous cross product formula.



Figure 11: Overview of the steps mentioned above.

<sup>a</sup>This is called the gift wrapping algorithm.

**Algorithm 6** Finding Convex Hull with Graham Scan**Require:**  $p_1, \dots, p_n \in \mathbb{R}^2$ 

```

function GRAHAM(p)
  s ← stack()
  push  $p_1, p_2$  onto s
  for  $i = 3, 4, \dots, n$  do
    push  $p_i$  onto stack
    while top 3 on stack are clockwise do
      pop second from top point
    end while
  end for
  return s
end function

```

Note that once we have 3 points, the first three in the stack will always be counterclockwise, so we will never need to check if there are enough points in the stack. Even in the beginning, the next point (third) added is guaranteed to be counterclockwise because of ordering. Note that even though we might do a bunch of pushes and pops, the maximum number of pushes we can do is  $n$  and pops is  $n$ , so this is  $O(n)$ . In fact, the bottleneck is the sorting, which is  $O(n \log n)$ , so the total runtime is  $O(n \log n)$ .

## 8 Linear Programming

## 9 Streaming Algorithms

**Algorithm 9.1** (Boyer-Moore Majority Vote Algorithm)**Algorithm 7****Require:**

```

function FUNC(x)
end function

```

**Algorithm 9.2** (Track Median From Data Stream)**Algorithm 8****Require:**

```

function FUNC(x)
end function

```