Algorithms

Muchang Bahng

Spring 2024

Contents

1 Complexity **2**
1.1 Recursive Algorithms and Recurrence Relation . . . 3

2 Brute Force Algorithms **4**
2.1 Basic Arithmetic . . . 4
2.2 Lists . . . 5
2.3 Stack, Queues, Heaps . . . 5
2.4 Cryptography . . . 5
2.5 Matrix Operations . . . 6

3 Divide and Conquer **6**
3.1 Karatsuba Algorithm for Multiplication . . . 6
3.2 Merge Sort . . . 6
3.3 Strassen Algorithm . . . 6
3.4 Fast Fourier Transform . . . 6

4 Hashing and Probabilistic Algorithms **6**
4.1 Modulo Operations . . . 6
4.2 Hashing . . . 6
4.3 Primality Testing . . . 6

5 Dynamic Programming **6**
5.1 Longest Increasing Subsequence . . . 6
5.2 Knapsack . . . 6

6 Graphs **6**
6.1 Representations and Properties . . . 7
6.2 Exploration . . . 9
6.3 Directed Acyclic Graphs and Topological Sorting . . . 11
6.4 Bipartite Graphs . . . 14
6.5 Strongly Connected Graphs . . . 16
6.6 Shortest Path . . . 16
6.7 Negative Weighted Graphs . . . 17

7 Approximation Algorithms **19**
7.1 Greedy Algorithms . . . 19

8 Linear Programming **19**

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.[a]
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,[1] we can simply take the limit.

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

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.

$$f = O(g), g = O(h) \implies f = O(h) \tag{2}$$
$$f = \Omega(g), g = \Omega(h) \implies f = \Omega(h) \tag{3}$$
$$f = \Theta(g), g = \Theta(h) \implies f = \Theta(h) \tag{4}$$
$$\tag{5}$$

[a] 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.
[1] These will be positive since the runtime must be positive.

2. Linearity.

$$f = O(h), g = O(h) \implies f + g = O(h) \tag{6}$$
$$f = \Omega(h), g = \Omega(h) \implies f + g = \Omega(h) \tag{7}$$
$$f = \Theta(h), g = \Theta(h) \implies f + g = \Theta(h) \tag{8}$$
$$\tag{9}$$

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

$$\lim_{n\to\infty} \frac{(\log n)^2 0}{n} = \lim_{n\to\infty} \frac{20 \cdot (\log n)^{19} \cdot \frac{1}{n}}{1} = \ldots = \lim_{n\to\infty} \frac{20!}{n} = 0 \implies f = O(g) \tag{10}$$

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

$$\lim_{n\to\infty} \frac{n^{100}}{1.01^n} = \lim_{n\to\infty} \frac{100n^{99}}{1.01^n \cdot \log(1.01)} = \ldots = \lim_{n\to\infty} \frac{100!}{1.01^n \cdot (\log 1.01)^1 00} = 0 \implies f = O(g) \tag{11}$$

Let's do a slightly more nontrivial example.

Example 1.2 ()

Given the following algorithm, what is the runtime?

```
for i in range(1, n+1):
  j = 1
  while j <= i:
    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

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

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.[a]

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

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.

[a] 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.[a]
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.

[a] 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.

```
def gcd(a, b):
  if a == b:
    return a
  elif a > b:
    return gcd(a - b, b)
  else:
    return gcd(a, b - a)

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)

3.1 Karatsuba Algorithm for Multiplication

3.2 Merge Sort

3.3 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** 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** 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.[a]
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 Konigsberg*. Euler, in trying to solve this problem, had invented graph theory.

Example 6.1 (Bridges of Konigsberg)

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

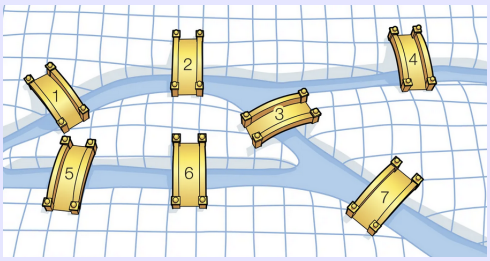

Figure 1: Bridges of Konigsberg

It can be decomposed into this undirected graph.

[a] Obviously, a connected undirected graph is also strongly connected.

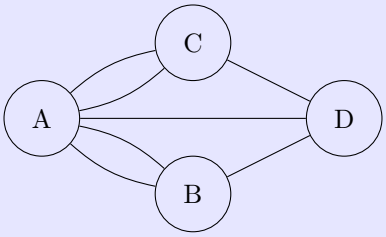

Figure 2: Graph representation.

Euler's observation is that except for start and end points, a talk leaves any vertex by different edge that 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*.

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

$$A_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{if } (i,j) \notin E \end{cases} \tag{14}$$

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.[a]

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.

[a] This makes sense, since to get back to a previous vertex you must backtrack.

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

```
visited = set()
def dfs(start):
  if start not in visited:
    visited.add(start)
    # do something
    neighbors = ...
    for neighbor in neighbors:
      dfs(neighbor)
```

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

```
visited = set()

def dfs(start):
  toExplore = []
  current = start;
  toExplore.append(current)
  visited.add(current)
  while toExplore:
    current = toExplore.pop()
    # Do something
    neighbors = ...
    for neighbor in neighbors:
      if neighbor not in visited:
        visited.add(neighbor)
        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.[a]

```
visited = set()
def bfs(start):
  toExplore = collections.deque()
  current = start;
  toExplore.append(current)
  visited.add(current)
  while toExplore:
    current = toExplore.popleft()
```

```
    # Do something
    neighbors = ...
    for neighbor in neighbors:
      if neighbor not in visited:
        visited.add(neighbor)
        toExplore.append(neighbor)
```

Theorem 6.3 (Runtime of BFS)

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

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.

$$O\left(\sum_{v\in V} |\text{neighbors of } v| + 1\right) = O\left(\sum_{v\in V} d_v + 1\right) \tag{15}$$

$$= O(2|E| + |V|) = O(m+n) \tag{16}$$

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.

You can make a BFS tree from this graph, and there has to be a last layer. Just remove the node from this layer.

Exercise 6.2 ()

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.

Exercise 6.3 (DPV 3.16)

[a] The recursive version of BFS is very nontrivial.

Proof.

Do BFS but now recursively store the maximum of all the prerequisite courses plus 1 for the current course.

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

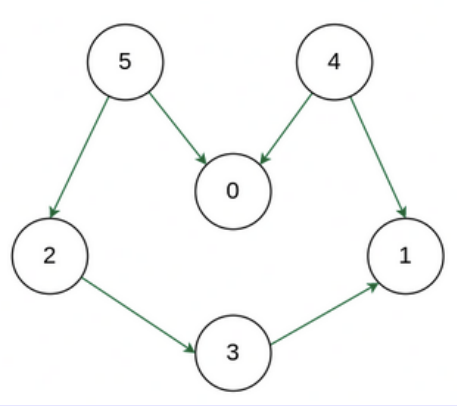

Figure 3

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 as 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
  visited ← set()
  res ← stack()
  is_acyclic ← True
  function DFS(v ∈ V)
      if v ≠ visited then
          add v to visited
          N_v ← neighbors of v
          for n ∈ N_v do
              if n ∈ visited then
                  is_acyclic ← False
              end if
              DFS(n)
          end for
          push v onto res
      end if
  end function

  function TopologicalSort(V, E)
      for v ∈ V do
          DFS(v)
      end for
      if ! is_acyclic then
          return False
      end if
      return reversed of res
  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
  q ← queue()
  indeg ← list()
  visited ← 0
  function RECUR(x)
      initialize the indeg and q
      while q is nonempty do
          v ← pop(q)
          visited += 1
          for each w ∈ E[v] do
              indeg[w] -= 1
              if indeg[w] = 0 then
                  push w into q
              end if
          end for
      end while
      if visited != |V| then
          return False
      end if
      return True
  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.[2] 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.

[2]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
  visited = set()
  layers = {v : 0 | v ∈ V}
  function BFS(start)
      layer ← 0
      toExplore ← queue()
      add (start, layer) to toExplore
      add start to visited
      while toExplore do
          curr, layer = pop from toExplore
          layers[curr] = layer
          for n ∈ neighbors of curr do
              if n ∉ visited then
                  add n to visited
                  add (n, layer + 1) to visited
              end if
          end for
      end while
  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.

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 $G(V, L)$ 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
  visited = set()
  layers = {v : 0 | v ∈ V}
  function BFS(start)
      layer ← 0
      toExplore ← queue()
      add (start, layer) to toExplore
      add start to visited
      while toExplore do
          curr, layer = pop from toExplore
          layers[curr] = layer
          for n ∈ neighbors of curr do
              if n ∉ visited then
                  add n to visited
                  add (n, layer + 1) to visited
              end if
          end for
      end while
  end function

  function BIPARTITE(V, E)
      BFS(v) for some v ∈ V
      for (u, v) ∈ E do
          if layers[u] == layers[v] then
              return False
          end if
      end for
      return True
  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
  function StronglyConnected(s ∈ V)
      visited ← set()
      BFS(s)
      if visited != V then
          return False
      end if
      visited ← set()
      reverse all edges in E
      BFS(s)
      if visited != V then
          return False
      end if
      return True
  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.

Algorithm 6.6 (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
  function DIJKSTRA(s)
      dist ← list() of very large initial numbers
      dist[s] ← 0
      predecessors ← {v : None | v ∈ V}
      toExplore ← minheap() // priority queue
      add (0, s) to toExplore
      while toExplore do
          (curr_dist, curr_node) ← pop from toExplore
          if curr_dist > dist[curr_node] then
              continue
          end if
          for neighbor, weight ∈ E[curr] do
              new_dist ← curr_dist + weight
              if new_dist < dist[neighbor] then
                  dist[neighbor] = new_dist
                  predecessors[neighbor] = curr_node
                  push (new_dist, neighbor) onto toExplore
              end if
          end for
      end while
      return distances, predecessors
  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

$$O\big((m+nd)\log_d(n)\big) \tag{17}$$

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 $\Theta(m)$.

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**.

$$d[v] = \min_w \{d[w] + l_{wv}\} \tag{18}$$

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,

$$d[v] = d[x] + l_{xv} \leq d[w] + l_{wv} \implies d[v] = \min_w \{d[w] + l_{wv}\} \tag{19}$$

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

Theorem 6.9 ()

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

So how do we actually implement this?

Algorithm 6.7 (Shortest Path in Possibly Negative Weighted Graph)

Require: Nodes V, Edges E
  function ShortPath(V, E)
    res $\leftarrow$ list(0) of large numbers of size $|V|$.
    res[s] = 0
    predecessors $\leftarrow \{v : None | v \in V\}$
    while $\exists (u, v)$ s.t. res[v] > res[u] + l_{uv} **do**
      res[v] $\leftarrow$ res[u] + l_{uv}
      predecessor[v] $\leftarrow$ u
    end while
  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.[a]
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, 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 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 6.8 (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
  $\pi \leftarrow$ list(0) of large numbers of size $|V|$.
  $\pi[s] = 0$
  function BellmanFord(x)
    for $t = 1, \dots, n-1$ **do**
      for $v \in V$ **do**
        $\pi^{(t)}[v] \leftarrow \min_w\{\pi^{(k-1)}[v], \pi^{(k-1)}[w] + l_{wv}\}$
      end for
    end for
  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.

7 Approximation Algorithms

7.1 Greedy Algorithms

8 Linear Programming

[a] This algorithm is also called *policy iteration* in reinforcement learning and is analogous to gradient descent.