# Algorithms

Muchang Bahng

Spring 2024

## Contents

# 1 Graph Algorithms

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*.

## 1.1 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 1.1 (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 1.2 (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 $(i, j) \neq (j, i)$).

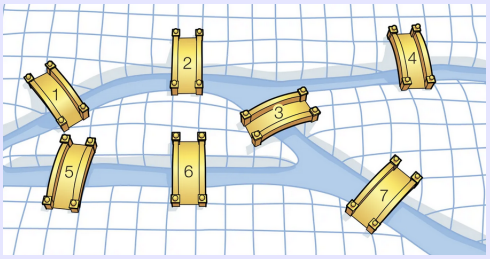
1. The **in/out degree** $d_{v,i}, d_{v,o}$ of a vertex $v$ is the number of edges going in to or out from $v$.
2. A **path** is a sequence of vertices where adjacent vertices are connected by a path in $E$. It's **length** is the number of edges in the path.
3. A **cycle/circuit** is a path that has 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 $e_j$.[a]
5. A **strongly connected component** is a maximal subset of connected vertices.

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

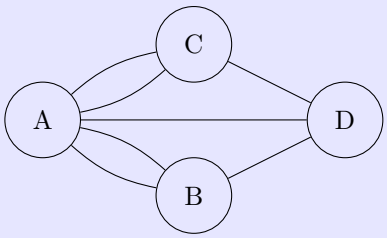
In fact, from these definitions alone, we can solve an ancient puzzle called *the Bridges of Konigsberg*. Euler, in trying to solve this problem, had invented graph theory.

**Example 1.1 (Bridges of Konigsberg)**

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




(a) Figure of the bridges of Konigsberg.




(b) Graph representation.

Figure 1: It can be decomposed into this undirected graph.

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*.

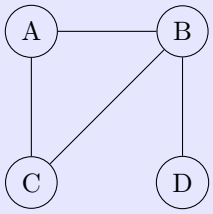
**Definition 1.3 (Adjacency Matrix)**

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

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

**Example 1.2 (Adjacency List vs Matrix)**

Given a graph, we can completely represent it with a list of adjacent vertices for each vertex or an adjacency matrix.




An adjacency list would look something like this

$$\begin{aligned} &A : B, C \\ &B : A, C, D \\ &C : A, B \\ &D : B \end{aligned}$$

and the adjacency matrix looks like this:

| | $A$ | $B$ | $C$ | $D$ |
|---|---|---|---|---|
| $A$ | 0 | 1 | 1 | 0 |
| $B$ | 1 | 0 | 1 | 1 |
| $C$ | 1 | 1 | 0 | 0 |
| $D$ | 0 | 1 | 0 | 0 |

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 1.4 (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.

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

**Theorem 1.1 (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.

## 1.2 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.

Now, in order to traverse this graph, we basically want to make an algorithm that starts at a node, prints it value, and then goes to all of its neighbors (which we can access through the adjacency list) to print them out. Thus, this is by nature recursive. We don't want the algorithm to loop around printing nodes infinitely often, so we must create a base case that tells the algorithm to not print out a node. It makes sense to create a set of visited nodes, which we can add to whenever we reach a new node. So, if we ever come onto a node that we have visited, we can just tell the function to do nothing.

**Algorithm 1.1 (Recursive Depth-First Search)**

The recursive implementation of Depth-First Search explores a graph by recursively visiting each unvisited neighbor, going as deep as possible along each branch before backtracking.

```
Require: Graph G(V, E), start vertex s
function DFS-Recursive(G, s)
    visited ← ∅                          ▷Initialize empty set of visited vertices
    DFS-Visit(G, s, visited)
end function
function DFS-Visit(G, u, visited)
    Add u to visited
    /* Process vertex u here */
    for each neighbor v of u in G do
        if v ∉ visited then
            DFS-Visit(G, v, visited)
        end if
    end for
end function
```

Though recursion really makes this simple, we can construct an iterative approach that uses stacks. Note

that in recursion, we are really making a call stack of different functions. We can be explicit about this by actually implementing a stack, which would store all the nodes that we have discovered, but not yet explored from )i.e. all the current nodes). At each iteration, we would pick a node to continue exploring, and since this is a DFS, we would want to implement a LIFO stack so that the last element we input in is the first thing that we should explore from, i.e. we always explore from the last node discovered.

**Algorithm 1.2 (Iterative Depth-First Search)**

The iterative implementation of Depth-First Search uses a stack to mimic the function call stack of the recursive implementation. It explores vertices in the same order as the recursive version.

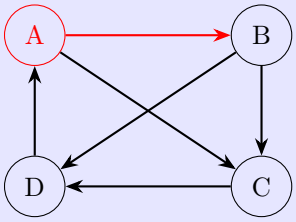
---

**Require:** Graph $G(V, E)$, start vertex $s$

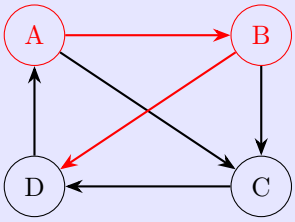
```
function DFS-ITERATIVE(G, s)
    visited ← ∅                          ▷Initialize empty set of visited vertices
    stack ← empty stack
    Push s onto stack
    Add s to visited
    while stack is not empty do
        u ← Pop from stack
        /* Process vertex u here */
        for each neighbor v of u in G do
            if v ∉ visited then
                Add v to visited
                Push v onto stack
            end if
        end for
    end while
end function
```

**Example 1.3 (DFS Walkthrough)**

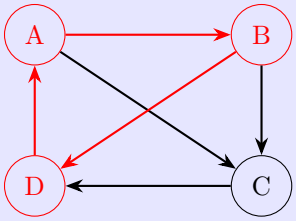
Let's conduct DFS on the following graph.



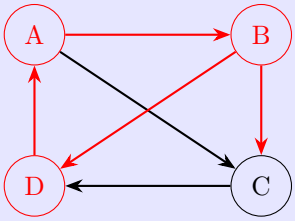

(a) You start at $A$ and you add it to your visited set $V = \{A\}$. You look at your neighbors and find that you can explore $B$ or $C$. Let's randomly choose to explore $B$ but keep in mind that you haven't finished exploring from $B$ since there's $C$ left.



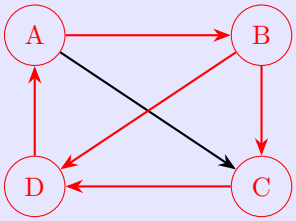

(b) You add $B$ to visited $V = \{A, B\}$ and look at your neighbors $C$ and $D$. Let's just choose $D$ randomly but keep in mind that you have $C$ to explore later, so you haven't finished exploring from $B$ either. Therefore both $A$ and $B$ are both still on your exploration stack.



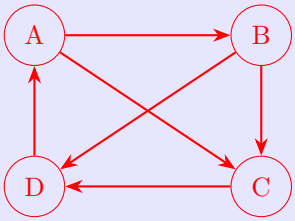

(c) You add $D$ to visited $V = \{A, B, D\}$. You look at your neighbors and see that you can explore $A$. However it is already in your visited set. Since there are no more neighbors to explore to, you are done exploring from $D$ and don't need to look at it anymore.




(d) Therefore you look back to where you came from: $B$, and continue exploring from $B$. You go back to $B$ and look at the other nodes you must explore. You've already done $D$ and the only remaining is $C$, which isn't in your visited.




(e) Add $C$ to visited $V = \{A, B, C, D\}$. You look at your neighbors and see $D$ as the only neighbor to explore. However, you have already explored $D$ and so there are no more neighbors left. Therefore you go back to the node you came from: $B$.




(f) You continue exploring from $B$. There are no more nodes to explore, so you go back to the node you came from $A$. You see that the only remaining neighbor to explore is $C$, but it is already in visited, so you don't need to explore it. You are done.

Figure 2: Walkthrough of DFS on a small graph. The iterative and recursive approaches are identical.

**Theorem 1.2 (Runtime of DFS)**

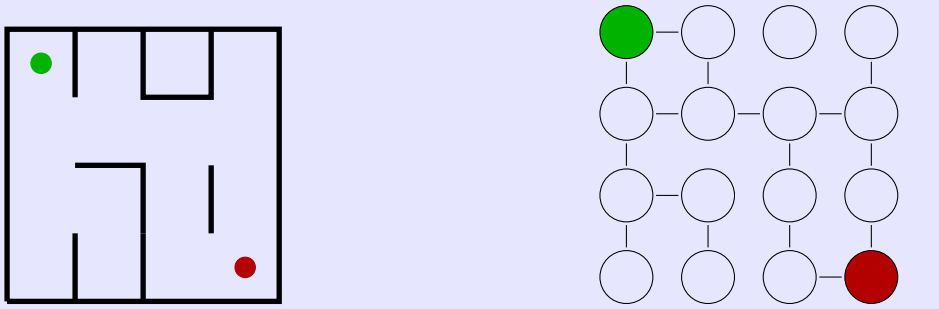
The runtime of DFS is $O(n + m)$.

**Proof.**

The runtime complexity of this search is $O(N + M)$ because first, the while loop loops at most over the $N$ nodes. The for loop may loop over $M$ edges, but this is a bit pessemistic in bound. Rather, we can view it as looping over neighbors of each node at most exactly once, and so it considers every edge twice, meaning that the for loop will get called $2M$ times in the entire algorithm. So $N + 2M = O(N + M)$.

**Example 1.4 (DFS in a Maze)**

We can represent a grid graph, like a maze, with a two dimensional array that stores whether it is connected north, east, south, and west, where boolean of true represents that there is a wall, and false means there isn't a wall (so connected).



(a) 4×4 maze representation

(b) Graph representation of the maze.

Figure 3: A 4×4 maze (left) and its corresponding graph representation (right). Start and end points are shown in green and red respectively.

But remember that in a tree traversal, we recursively searched down and down until we hit a null node, in which case we backtrack up to look in another branch. For graphs, this is a bit more complicated, since we could go in loops. Therefore, we want to keep track of all the visited nodes to avoid infinite recursion. We have three base cases:

1. If we search off the grid, then this is not a valid path
2. If we already explored here, then we don't want to repeat it
3. If we reached the goal of the maze, then we output the length of the path.

The recursive case would take each node and recurse on its 4 adjacent neighbors, if they are connected. Note that this algorithm recurses on each of the $N$ nodes 4 times (for each direction, and each recursive call is $O(1)$), so the complexity is $O(N)$.

Note that the main idea of DFS is to always explore a new adjacent vertex if possible, and if not, then backtrack to the most recent vertex adjacent to an unvisited vertex and continue. On the contrary, the main idea of BFS is to explore *all* your neighbors before you visit any of your neighbors' neighbors. It exhaustively searches for the closest regions of your search space before you look any further. Unlike DFS, which finds the some arbitrary path to a node, BFS finds the shortest (perhaps non-unique) path to a node. This can be simply done with a queue.

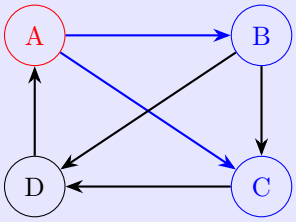
**Algorithm 1.3 (Iterative Breadth-First Search)**

Breadth-First Search (BFS) explores a graph by visiting all neighbors at the current depth before moving to nodes at the next depth level. It uses a queue to process nodes in the order they are discovered.

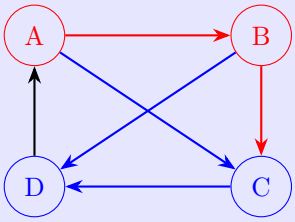
**Require:** Graph $G(V, E)$, start vertex $s$
**function** BFS($G, s$)
    visited $\leftarrow \emptyset$ ▷Initialize empty set of visited vertices
    queue $\leftarrow$ empty queue
    Enqueue $s$ onto queue
    Add $s$ to visited
    **while** queue is not empty **do**
        $u \leftarrow$ Dequeue from queue ▷Get the next vertex to process
        /* *Process vertex* $u$ *here* */
        **for** each neighbor $v$ of $u$ in $G$ **do**
            **if** $v \notin$ visited **then**
                Add $v$ to visited
                Enqueue $v$ onto queue
            **end if**
        **end for**
    **end while**
**end function**

**Example 1.5 (BFS Walkthrough)**

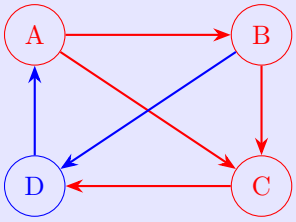
Let's conduct BFS on the following graph.



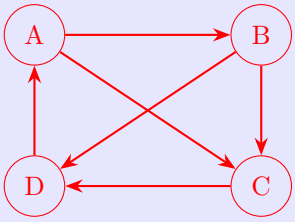

(a) You start at $A$ and you add it to your visited set $V = \{A\}$ and to your queue $[A]$. You immediately dequeue $A$ from the queue and look at all neighbors of $A$, which are $B$ and $C$. You add both to your queue in some random order, say $B$ first.




(b) You deque $B$ and see that the adjacent nodes are $C$ and $D$. $C$ is already in your queue (check with visited set, which is $O(1)$), so you do not put it into your queue and can mark the corresponding edges as explored. You do put $D$ into the queue.




(c) Next you dequeue $C$ since this is first seen from $A$. You explore from here and look at the neighbors, which is $D$. However, $D$ is already in your queue from $B$, so you skip it. There are no more neighbors to explore from $C$, so we look at the queue again.




(d) You dequeue $D$ and look at the neighbors. The only neighbor is $A$ and it is already in visited, so we can skip this and mark the corresponding edge as explored. Since there are no more nodes to explore in our queue, this concludes BFS.

Figure 4: Walkthrough of BFS on a small graph. Blue represents the nodes/edges that are in our queue and red represents the nodes/edges that we have finished exploring.

**Theorem 1.3 (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 $V$.

$$O\Big(\sum_{v\in V} |\text{neighbors of } v| + 1\Big) = O\Big(\sum_{v\in V} d_v + 1\Big) \tag{2}$$

$$= O(2|E| + |V|) = O(m+n) \tag{3}$$

which is linear in input size!

The more straightforward application is in reachability.

**Example 1.6 (Reachability)**

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

**Exercise 1.1 ()**

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,[a] 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.

[a] 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.

**Exercise 1.2 ()**

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.

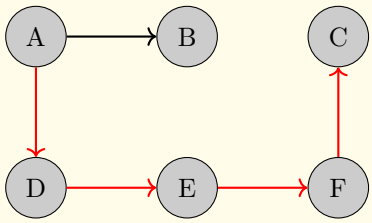
**Proof.**

Listed.

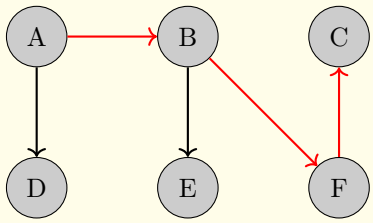
1. A graph whose edges form a cycle, having at least three nodes.
2. Two strongly connected components with no edges between them.

**Definition 1.5 (Search Trees)**

Once we have traversed a graph using BFS or DFS, we can label the directed path that this traversal algorithm takes into a **search tree**.




(a) After DFS traversal, we can store the previous nodes in a hashmap $\{B \mapsto A, D \mapsto A, E \mapsto D, F \mapsto E, C \mapsto F\}$. From this we can see the path to get to $C$ is $A \mapsto D \mapsto E \mapsto F \mapsto C$ of length 4.




(b) After BFS traversal, we can store the previous nodes in a hashmap $\{B \mapsto A, D \mapsto A, E \mapsto B, F \mapsto B, C \mapsto F\}$. From this we can see the path to get to $C$ is $A \mapsto B \mapsto F \mapsto C$ of length 3.

Figure 5: Comparison of two directed graphs with different path lengths from A to C.

By construction, we can see that the path from A to C is always shorter for BFS than for DFS.

## 1.3 Directed Acyclic Graphs and Topological Sorting

**Definition 1.6 (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.

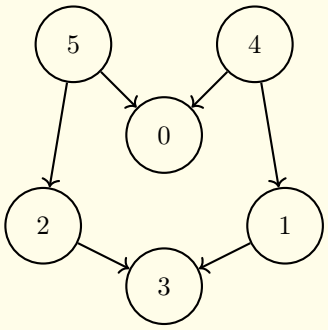
To determine if a graph is a DAG, then we can brute force it by taking a node $s \in V$, running DFS/BFS, and if a neighbor is already in visited, return False. Then go through this for all starting nodes $s \in V$. This again has quadratic runtime. Can we do better? This introduces us to topological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

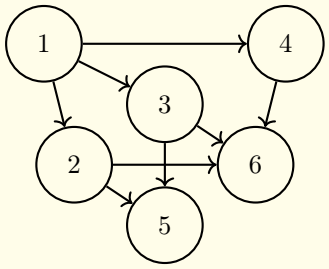
**Definition 1.7 (Topological Sort)**

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.




(a) This graph can have the two (not exhaustive) topological sortings $[5, 4, 2, 3, 1, 0]$ and $[4, 5, 2, 3, 1, 0]$.




(b) This graph can have the two (not exhaustive) topological sortings $[1, 4, 2, 3, 6, 5]$ and $[1, 4, 3, 2, 6, 5]$.

Figure 6: Examples of topological sortings of different graphs.

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

**Theorem 1.4 (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 1.4 (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 1.5 (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 1.5 ()**

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.[1] 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.

**Algorithm 1.6 (BFS Tree)**

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

[1] In fact, this Kosaraju's algorithm, can be done in linear time, though it is highly nontrivial.

```
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.

**Exercise 1.3 (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 1.4 (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.[a] 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 $G$.

[a] The following argument can be made formal with induction.

**Exercise 1.5 (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 $u \in V$,

$$z[u] = \max\{x[v] \mid v \text{ is a descendant of } u\} \tag{4}$$

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:

```
computeZ(u):
  maxVal = x[u]
  if u.left is not null:
    computeZ(u.left) # compute z for all descendants of u.left
```

```
    maxVal = max(z[u.left], maxVal)
  if u.right is not null:
    computeZ(u.right) # computes z for all descendants of u.right
    maxVal = max(z[u.right], maxVal)
  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[a]

[a]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.

**Exercise 1.6 ()**

Your data center needs to run a number of jobs (compute requests) numbered $1, 2, \ldots, n$. 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 $1, \ldots, n$ 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:

   $$\text{Any SCC with } \geqslant 2 \text{ vertices contain a cycle.}$$

   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 $\max_k L[k]$.

## 1.4 Bipartite Graphs

Now we shall see a further application of BFS trees.

**Definition 1.8 (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 1.6 ()**

$G$ 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 1.7 (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.

## 1.5 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 1.7 ()**

$G$ 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 1.8 (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 != $V$ **then**
5: **return** False
6: **end if**
7: visited $\leftarrow$ set()
8: reverse all edges in $E$
9: BFS(s)
10: **if** visited != $V$ **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 $O(n+m)$.

## 1.6 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 1.9 (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: $S \leftarrow \{s\}$ ▷set of nodes that we know for sure is shortest
2: dist[s] = 0 and the rest very large numbers ▷our final list
3: $\pi[v] = w_{sv}$ for all $v \in V$ ▷initialize list with neighboring nodes from start $s$
4: **function** Dijkstra(s)
5: $u = \text{argmin}_{v \notin S} \pi[v]$ ▷Find node having minimum accum weight so far
6: add $u$ to $S$ ▷This node must be shortest so add to $S$
7: dist$[u] = \pi[u]$ ▷Now we can update its shortest path in dist
8: **for** $v \notin S$ **do** ▷Look at all neighbors of $u$ and update those not in
9: $\pi[v] \leftarrow \min\{\pi[v], d[u] + w_{uv}\}$ ▷$S$ 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 1.10 (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 size |V| with +∞      ▷Initialize list of big nums representing shortest distances
    dist[s] ← 0                          ▷The starting node has dist 0
    predecessors ← {v : None | v ∈ V}    ▷predecessors of each node for path tracking
    toExplore ← minheap()                ▷A priority queue of (weight, node)
    add (0, s) to toExplore              ▷You want to explore this first
    while toExplore do
        (curr_dist, curr_node) ← pop from toExplore    ▷pop from toExplore
        if curr_dist > dist[curr_node] then            ▷If this distance is greater than what
            continue                                   ▷I already have then not worth exploring
        end if
        for neighbor, weight ∈ E[curr_node] do         ▷Look at each neighbor
            new_dist ← curr_dist + weight              ▷The distance to getting to neighbor from now
            if new_dist < dist[neighbor] then          ▷If this new dist is shorter than what we have
                dist[neighbor] = new_dist              ▷Update best distance
                predecessors[neighbor] = curr_node     ▷Update its predecessor
                push (new_dist, neighbor) onto toExplore    ▷Should prob explore from here
            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

$$O\big((m+nd)\log_d(n)\big) \tag{5}$$

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 1.7 ()**

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 $u \to v$ walk into the combination of two paths $u \to v_0 \to v$, where we try to minimize the cost of both subpaths. It's easy to compute the minimum cost of $v_0 \to v$ for all $v$: running Dijkstra once over the graph suffices. The first half, $u \to v_0$, 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 $v_0 \to u$ path in $G^{-1}$ would correspond to the cheapest $u \to v_0$ path in $G$, with matching total costs.

**Exercise 1.8 ()**

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 $(u \to v \in E$, 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 $u \to v$ 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 Dijkstra$(G^{-1}, t)$. This finishes in $\mathcal{O}((m+n) \log n) = O(n \log n)$ time since $G$ is sparse.

**Exercise 1.9 ()**

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 $u \to v \to w \to u$, 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.:

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

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 1.10 ()**

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.

## 1.7 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.

We use the Bellman equations, which could be solved using dynamic programming like we've seen before.

**Theorem 1.8 (Bellman Equations)**

We write the **Bellman equations**.

$$d[v] = \min_w \{d[w] + l_{wv}\} \tag{6}$$

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,

$$d[v] = d[x] + l_{xv} \leq d[w] + l_{wv} \implies d[v] = \min_w \{d[w] + l_{wv}\} \tag{7}$$

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 $d[v] <= \pi[v]$ for all $v$.

**Theorem 1.9 ()**

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 1.11 (Shortest Path in Possibly Negative Weighted Graph)**

**Require:** Nodes $V$, Edges $E$
1: **function** ShortPath(V, E)
2: res $\leftarrow$ list(0) of large numbers of size $|V|$.
3: res[$s$] = 0
4: predecessors $\leftarrow \{v : None | v \in V\}$
5: **while** $\exists (u, v)$ s.t. res[v] > res[u] + $l_{uv}$ **do**
6: res[$v$] $\leftarrow$ res[u] + $l_{uv}$
7: 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.[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] This algorithm is also called *policy iteration* in reinforcement learning and is analogous to gradient descent.

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 1.12 (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 $E$

```
π ← list(0) of large numbers of size |V|.
π[s] = 0
function BellmanFord(x)
    for t = 1, ..., n − 1 do
        for v ∈ V do
            π^(t)[v] ← min_w{π^(k−1)[v], π^(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.

**Exercise 1.11 ()**

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 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 thates 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 "$g \geqslant 3$" 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 $\geqslant 3$ 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 $\mathcal{O}(|V||E|)$.

## 1.8 All Pairs Shortest Paths

Now what if we want to find the minimum distance between all $u, v \in V$? We can just use $|V|$ Dijkstras or Bellman-Fords to get the appropriate runtimes of $O(EV + V^2 \log V)$ or $O(EV^2)$, respectively, but for negative weighted graphs, there is a way to do this in $O(V^3)$.[2]

## 1.9 Minimum Spanning Trees

**Definition 1.9 (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

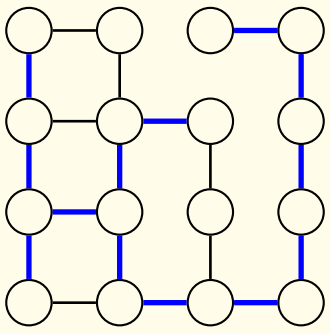


Figure 7: A spanning tree on a $4 \times 4$ grid graph.

[2]Note that if the graph is sparse, then $|E| < |V|$ and just running $|V|$ Bellman Fords may be optimal.

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

**Theorem 1.10 (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.[3] How do we do this? There are two well-known algorithms to solve this. Prim's and Kruskal's algorithm.

### 1.9.1 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.

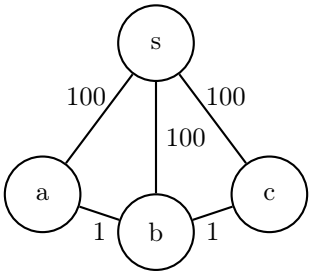



Figure 8: If we run Dijkstra on $s$, then our output will be a tree of cost 300, even when the actual MST can be of cost 102 starting from $a$.

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

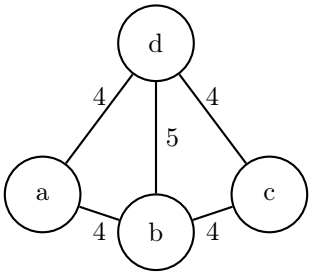



Figure 9: No matter where we start from, we will never output the MST. The MST has cost 12. If we start from $b$ or $d$, we will get a tree of cost 13. If we start from $a$ or $c$, we will get a tree of cost 16.

**Definition 1.10 (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 $V \setminus S$.

[3] 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?

**Theorem 1.11 (Cycles and Cuts)**

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

$$|C \cap \text{Cut}(S)| \tag{8}$$

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

Now time for a bizarre theorem.

**Theorem 1.12 (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 is precisely the MST! Therefore, an edge $e \in \text{MST}$ iff $e$ is a min-cost edge for some cut.

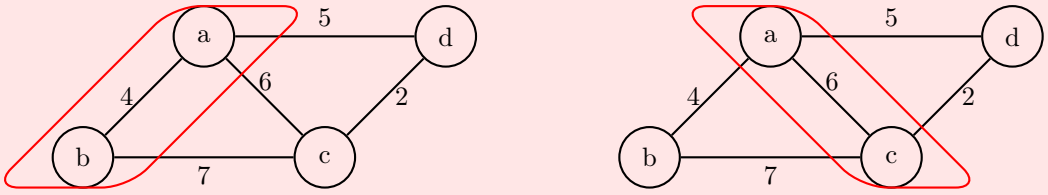



Figure 10: In the left cut, the edges are $(a, d), (b, c), (a, c)$. The minimum weight is 5 on the $(a, d)$ edge, so it must be in the MST. For the right cut, $(c, d)$ must be in the MST.

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 1.13 (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: $S \leftarrow \{s\}$ ▷Our initial cut
3: $\pi[v] \leftarrow$ list of size $|V|$ of $+\infty$ ▷$\pi[u]$ is min cost of getting from $u$ into the $S$
4: $\pi[v] = w_{sv}$ for all $v \in V$ ▷initialize list with neighboring nodes from start $s$
5: **while** $S \neq V$ **do**
6: $u = \text{argmin}_{v \notin S} \pi[v]$ ▷Find node having minimum cost to reach from $S$
7: $S \leftarrow S \cup \{u\}$ ▷Adding this node to $S$ to expand our cut
8: **for** $u \notin S$ **do** ▷Since we expanded $S$, our min reach distances
9: $\pi[u] \leftarrow \min\{\pi[u], w_{wu}\}$ ▷must be updated. It can only get shorter
10: **end for** ▷through a path from new $u$, so compare them
11: **end while**
12: **end function**

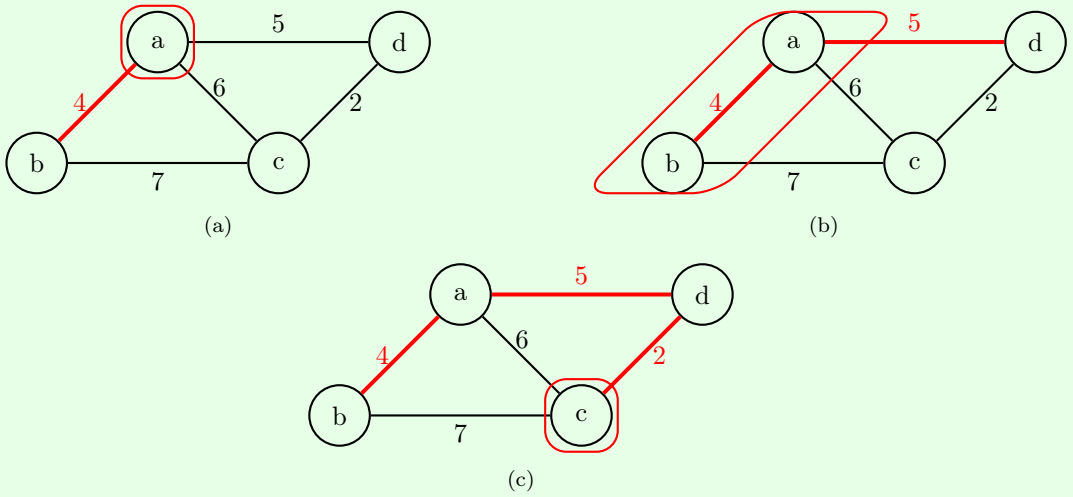



Figure 11: Step by step process of the method we mention above.

We are really going through two loops. We add to the cut $S$ $n$ times and for each time we add, we must compute the argmin, which is also $O(n)$, so our total time complexity of $O(n^2)$.

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

**Algorithm 1.14 (Prim's Algorithm with MinHeap)**

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

```
function PRIM(V, E)
    mst ← []                                          ▷Initialize mst array to return
    s ← 0                                             ▷Choose any starting node
    visited ← set()                                   ▷Our expanding set of cuts.
    edges ← minheap()                                 ▷The set of low-weight edges that we can explore from
    add (weight, s, next_node) for edges in E         ▷Look for edges from s to expand our cut from.
    while |visited| < |V| do                          ▷Until we have visited all cuts,
        weight, frm, to ← pop from edges              ▷Get the cheapest edge to explore
        if to ∉ visited then                          ▷If this isn't already in our cut,
            add to → visited                          ▷Add it to our cut. From cut
            add (frm, to, weight) to mst              ▷property, this must be added to mst
            for next_to, next_weight ∈ E[to] do       ▷After expanding, add newly discovered
                if next_to ∉ visited then             ▷edges for future exploration
                    push (next_weight, to, next_to) to edges
                end if
            end for
        end if
    end while
    return mst                                        ▷of form (from, to, weight)
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 $O((n+m)log(n)) = O(m\log n)$.

This can be sped up even faster if we use Fibonacci heaps or assume extra structure on the graph.

### 1.9.2 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 1.13 (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? Kruskal's algorithm answers this question, which takes a greedy approach.

**Algorithm 1.15 (General Kruskal's Algorithm)**

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: $T \leftarrow T \cup \{e\}$ ▷Cut property
7: **else** ▷Cycle property
8: continue ▷discard $e$ since from sorting, this edge is heaviest in cycle
9: **end if**
10: **end for**
11: **end function**

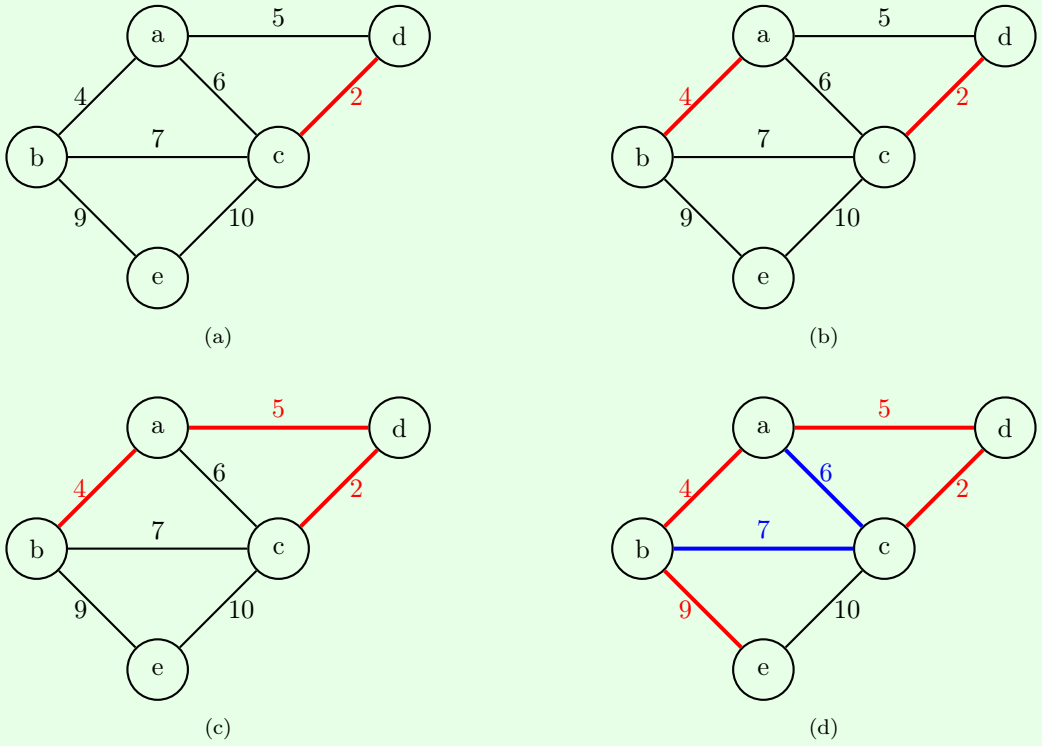



Figure 12: Kruskal's algorithm. In the last step, we see that the next minimum cost edge of 6 and 7 forms a cycle, so we add the edge of length 9.

The sorting of edges take $O(m \log m)$ time, and after sorting, we iterate through all the edges and apply $m$ find-union algorithm, which each take at most $O(\log n)$ time. Therefore, the overall complexity is $O(m \log m + m \log n)$. However, the value of $m$ can be at most $O(n^2)$, so the two logarithms are essentially the same, arriving at the final runtime of $O(m \log n)$.

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.

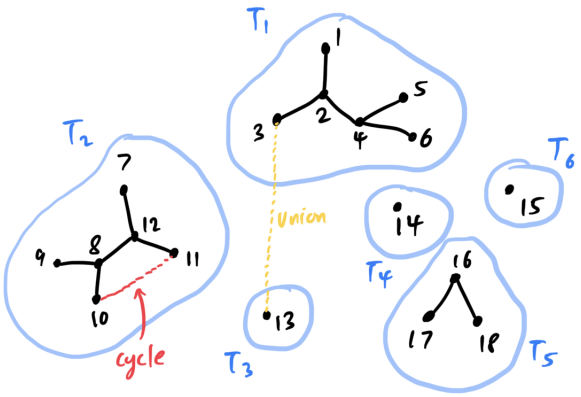



Figure 13: Each $T_i$ is a component formed by the edges chosen so far. For example, $T_1 = \{1, 2, 3, 4, 5, 6\}, \ldots$. We can either discard an edge (red) or include an edge (yellow).

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 1.16 (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 ← |V|
    sort E in increasing order of weights.                ▷Needed for Kruskal
    parent ← [0, ..., n − 1]                              ▷Initialize the disjoint cluster each node is in
    rank ← list of 0s of size n
    function FIND(x)
        if parent[x] ≠ x then
            parent[x] ← Find(parent[x])                   ▷Path compression
        end if
        return parent[x]
    end function
    function UNION(x, y)
        px, py = find(x), find(y)
        if px = py then
            return False
        end if
        if rank[px] < rank[py] then
            parent[px] ← py
        else if rank[px] > rank[py] then
            parent[py] ← px
        else
            parent[py] ← px
            rank[px] ← rank[px] + 1
        end if
        return True
    end function
    mst ← []
    for u, v, weight ∈ edges do
        if union(u, v) then
            add (u, v, weight) to mst
        end if
        if len(mst) = n - 1 then
            break
        end if
    end for
    return mst
end function
```

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

**Definition 1.11 (Ackerman Function)**

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

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases} \tag{9}$$

The inverse Ackerman function therefore grows extremely slowly.

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

$$O((m+n)\log^*(n)) \tag{10}$$

which is practically linear.

### 1.9.3 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 1.17 (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 1.14 ()**

The algorithm above minimizes the objective function.

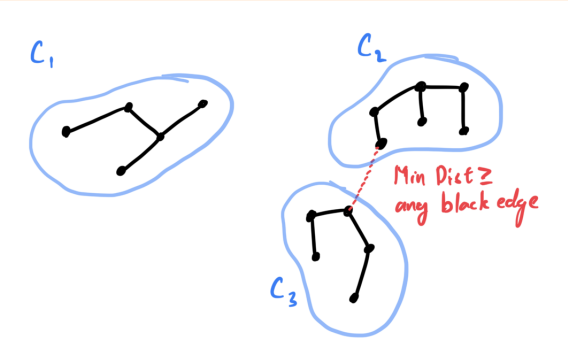
$$\underset{\mathbf{C}}{\operatorname{argmax}} \min_{p \in C_i, q \in C_j} \{d(p,q)\} \tag{11}$$

**Proof.**

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

$$\text{mindist}(\mathbf{C}) \leq \text{mindist}(\mathbf{C}^*) \tag{12}$$

Assume that this was not the case, so $\text{mindist}(\mathbf{C}) > \text{mindist}(\mathbf{C}^*)$ and therefore there exists a $p, q \in C_i, C_j$ such that $d(p,q) = \text{mindist}(\mathbf{C}) > \text{mindist}(\mathbf{C}^*)$. 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 mindist($\mathbf{C}^*$), meaning that 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.