**What's a Graph?**

A bunch of vertices connected by edges.

**Why Graph Algorithms?**

- They're fun.
- They're interesting.
- They have surprisingly many applications.

**Graphs are Everywhere**
Adjacency as a Graph
Each vertex represents a state, country, etc. There is an edge between two vertices if the corresponding areas share a border.

When a Graph?
Graphs are a good representation for any collection of objects and binary relation among them:
- The relationship in space of places or objects
- The ordering in time of events or activities
- Family relationships
- Taxonomy (e.g. animal - mammal - dog)
- Precedence (x must come before y)
- Conflict (x conflicts or is incompatible with y)
- Etc.

Graphs
A graph consists of two sets: set V of vertices and set E of edges.
Terminology: endpoints of the edge, loop edges, parallel edges, adjacent vertices, isolated vertex, subgraph, bridge edge
Directed graph (digraph) has each edge as an ordered pair of vertices

Basic Concepts
A graph is an ordered pair (V, E).
V is the set of vertices. (You can think of them as integers 1, 2, ..., n.)
E is the set of edges. An edge is a pair of vertices: (u, v).
Note: since E is a set, there is at most one edge between two vertices. (Hypergraphs permit multiple edges.)
Edges can be labeled with a weight.
Concepts: Directedness

In a directed graph, the edges are “one-way.” So an edge \((u, v)\) means you can go from \(u\) to \(v\), but not vice versa.

In an undirected graph, there is no direction on the edges: you can go either way. (Also, no self-loops.)

Concepts: Adjacency

Two vertices are adjacent if there is an edge between them.

For a directed graph, \(u\) is adjacent to \(v\) iff there is an edge \((v, u)\).

Special Graphs

Simple graph is a graph without loop or parallel edges

A complete graph of \(n\) vertices \(K_n\) is a simple graph which has an edge between each pair of vertices

A complete bipartite graph of \((n, m)\) vertices \(K_{n,m}\) is a simple graph consisting of vertices, \(v_1, v_2, \ldots, v_n\), and \(w_1, w_2, \ldots, w_m\) with the following properties:

- There is an edge from each vertex \(v_i\) to each vertex \(w_j\)
- There is no edge from any vertex \(v_i\) to any vertex \(v_j\)
- There is no edge from any vertex \(w_i\) to any vertex \(w_j\)

Concepts: Degree

Undirected graph: The degree of a vertex is the number of edges touching it.

For a directed graph, the in-degree is the number of edges entering the vertex, and the out-degree is the number leaving it. The degree is the in-degree + the out-degree.
**Paths and Circuits**

A walk in a graph is an alternating sequence of adjacent vertices and edges.

- A path is a walk that does not contain a repeated edge.
- Simple path is a path that does not contain a repeated vertex.
- A closed walk is a walk that starts and ends at the same vertex.
- A circuit is a closed walk that does not contain a repeated edge.
- A simple circuit is a circuit which does not have a repeated vertex except for the first and last.

**Concepts: Path**

A path is a sequence of adjacent vertices. The length of a path is the number of edges it contains, i.e., one less than the number of vertices.

Is there a path from 1 to 4?
What is its length?
What about from 4 to 1?
How many paths are there from 2 to 3? From 2 to 2? From 1 to 1?

We write $u \Rightarrow v$ if there is a path from $u$ to $v$. (The correct symbol, a wiggly arrow, is not available in standard fonts.) We say $v$ is reachable from $u$.

**Concepts: Cycle**

A cycle is a path of length at least 1 from a vertex to itself.

- A graph with no cycles is acyclic.
- A path with no cycles is a simple path.

The path $<2, 3, 4, 2>$ is a cycle.

**Concepts: Connectedness**

An undirected graph is connected iff there is a path between any two vertices.

An unconnected graph with three connected components.

The adjacency graph of U.S. states has three connected components. Name them.
(We say a directed graph is strongly connected iff there is a path between any two vertices.)
Connectedness

Two vertices of a graph are connected when there is a walk between two of them.

The graph is called connected when any pair of its vertices is connected.

If graph is connected, then any two vertices can be connected by a simple path.

If two vertices are part of a circuit and one edge is removed from the circuit then there still exists a path between these two vertices.

Graph H is called a connected component of graph G when H is a subgraph of G, H is connected and H is not a subgraph of any bigger connected graph.

Any graph is a union of connected components.

Euler Circuit

Euler circuit is a circuit that contains every vertex and every edge of a graph. Every edge is traversed exactly once.

If a graph has Euler circuit then every vertex has even degree. If some vertex of a graph has odd degree then the graph does not have an Euler circuit.

If every vertex of a graph has even degree and the graph is connected then the graph has an Euler circuit.

A Euler path is a path between two vertices that contains all vertices and traverses all edge exactly ones.

There is an Euler path between two vertices v and w iff vertices v and w have odd degrees and all other vertices have even degrees.

Hamiltonian Circuit

Hamiltonian circuit is a simple circuit that contains all vertices of the graph (and each exactly once).

Traveling salesperson problem.

Concepts: Trees

A free tree is a connected, acyclic, undirected graph.

To get a rooted tree (the kind we’ve used up until now), designate some vertex as the root.

If the graph is disconnected, it’s a forest.

Facts about free trees:

- \(|E| = |V| - 1\)
- Any two vertices are connected by exactly one path.
- Removing an edge disconnects the graph.
- Adding an edge results in a cycle.
Rooted Trees
- Rooted tree is a tree in which one vertex is distinguished and called a root.
- Level of a vertex is the number of edges between the vertex and the root.
- The height of a rooted tree is the maximum level of any vertex.
- Children, siblings and parent vertices in a rooted tree.
- Ancestor, descendant relationship between vertices.

Binary Trees
- Binary tree is a rooted tree where each internal vertex has at most two children: left and right. Left and right subtrees.
- Full binary tree.
- Representation of algebraic expressions.
- If $T$ is a full binary tree with $k$ internal vertices then $T$ has a total of $2k + 1$ vertices and $k + 1$ of them are leaves.
- Any binary tree with $t$ leaves and height $h$ satisfies the following inequality: $t \leq 2^h$.

Spanning Trees
- A subgraph $T$ of a graph $G$ is called a spanning tree when $T$ is a tree and contains all vertices of $G$.
- Every connected graph has a spanning tree.
- Any two spanning trees have the same number of edges.
- A weighted graph is a graph in which each edge has an associated real number weight.
- A minimal spanning tree (MST) is a spanning tree with the least total weight of its edges.

Graph Size
- We describe the time and space complexity of graph algorithms in terms of the number of vertices, $|V|$, and the number of edges, $|E|$.
- $|E|$ can range from 0 (a totally disconnected graph) to $|V|^2$ (a directed graph with every possible edge, including self-loops).
- Because the vertical bars get in the way, we drop them most of the time.
- E.g. we write $\Theta(V + E)$ instead of $\Theta(|V| + |E|)$.
Representing Graphs

Adjacency matrix: if there is an edge from vertex i to j, $a_{ij} = 1$; else, $a_{ij} = 0$.

- Space: $\Theta(V^2)$

Adjacency list: Adj[v] lists the vertices adjacent to v.

- Space: $\Theta(V+E)$

Represent an undirected graph by a directed one:

Depth-First Search

A way to “explore” a graph. Useful in several algorithms.

Remember preorder traversal of a binary tree?

Binary-Preorder(x):
1 number x
2 Binary-Preorder(left[x])
3 Binary-Preorder(right[x])

Can easily be generalized to trees whose nodes have any number of children.

This is the basis of depth-first search. We “go deep.”

DFS on Graphs

The wrong way:

Bad-DFS(u)
1 number u
2 for each v in Adj[u] do
3 Bad-DFS(v)

What’s the problem?

Fixing Bad-DFS

We’ve got to indicate when a node has been visited.

Following CLRS, we’ll use a color:

- WHITE never seen
- GRAY discovered but not finished (still exploring its descendants)
- BLACK finished
A Better DFS

Initially, all vertices are WHITE

Better-DFS(u)

\[ \text{color}[u] \leftarrow \text{GRAY} \]

Number u with a “discovery time”

For each v in Adj[u] do

If \( \text{color}[v] = \text{WHITE} \) then \( > \) avoid looping!

Better-DFS(v)

\[ \text{color}[u] \leftarrow \text{BLACK} \]

Number u with a “finishing time”

Depth-First Spanning Tree

As we’ll see, DFS creates a tree as it explores the graph. Let’s keep track of the tree as follows (actually it creates a forest not a tree):

When v is explored directly from u, we will make u the parent of v, by setting the predecessor, aka, parent \( \pi \) field of v to u:

Two More Ideas

1. We will number each vertex with discovery and finishing times—these will be useful later.

The “time” is just a unique, increasing number.

The book calls these fields \( d[u] \) and \( f[u] \).

2. The recursive routine we’ve written will only explore a connected component. We will wrap it in another routine to make sure we explore the entire graph.

DFS(G)

1. \( \text{for each vertex } u \in V[G] \)
2. \( \text{do } \text{color}[u] \leftarrow \text{WHITE} \)
3. \( \pi[u] \leftarrow \text{NIL} \)
4. \( \text{time} \leftarrow 0 \)
5. \( \text{for each vertex } u \in V[G] \)
6. \( \text{do if } \text{color}[u] = \text{WHITE} \)
7. \( \quad \text{then } \text{DFS-Visit}(u) \)

DFS-Visit(u)

1. \( \text{color}[u] \leftarrow \text{GRAY} \) ▷ White vertex u has just been discovered.
2. \( \text{time} \leftarrow \text{time} + 1 \)
3. \( d[u] \leftarrow \text{time} \)
4. \( \text{for each } v \in \text{Adj}[u] \) ▷ Explore edge \( (u, v) \).
5. \( \quad \text{do if } \text{color}[v] = \text{WHITE} \)
6. \( \quad \quad \text{then } \pi[v] \leftarrow u \)
7. \( \quad \text{DFS-Visit}(v) \)
8. \( \text{color}[u] \leftarrow \text{BLACK} \) ▷ Blacken u; it is finished.
9. \( f[u] \leftarrow \text{time} \leftarrow \text{time} + 1 \)
graphs from p. 1081

show strongly connected comps

from prev slide
On an undirected graph, any edge that is not a "tree" edge is a "back" edge (from descendant to ancestor).

DFS running time is $\Theta(V+E)$

- We visit each vertex once; we traverse each edge once.

DFS Example: digraph

B = back edge (descendant to ancestor, or self-loop)
F = forward edge (ancestor to descendant)
C = cross edge (between branches of a tree, or between trees)

DFS running time is $\Theta(V+E)$

- Called only on white vertices ($\Theta(V)$)
- $\sum_{u \in V} \text{has}[u] = \Theta(E)$
applications of DFS
Connected components of an undirected graph. Each call to DFS_VISIT (from DFS) explores an entire connected component (see ex. 22.3-11).

So modify DFS to count the number of times it calls DFS_VISIT:
5 for each vertex \( u \in V[G] \)
6.5 then \( cc \_\text{counter} \leftarrow cc \_\text{counter} + 1 \)
7 \( \text{DFS \_VISIT}(u) \)

Note: it would be easy to label each vertex with its cc number, if we wanted to (i.e. add a field to each vertex that would tell us which conn comp it belongs to).

Applications of DFS
Cycle detection: Does a given graph G contain a cycle?

Idea #1: If DFS ever returns to a vertex it has visited, there is a cycle; otherwise, there isn’t.

OK for undirected graphs, but what about:

No cycles, but a DFS from 1 will reach 4 twice.
Hint: what kind of edge is (3, 4)?

Cycle detection theorem

Theorem: A graph G (directed or not) contains a cycle if and only if a DFS of G yields a back edge.

\( \rightarrow \): Assume G contains a cycle. Let \( v \) be the first vertex reached on the cycle by a DFS of G. All the vertices reachable from \( v \) will be explored from \( v \), including the vertex \( u \) that is just “before” \( v \) in the cycle. Since \( v \) is an ancestor of \( u \), the edge \((u, v)\) will be a back edge.

\( \leftarrow \): Say the DFS results in a back edge from \( u \) to \( v \).
Clearly, \( u \rightarrow v \) (that should be a wiggly arrow, which means, “there is a path from \( u \) to \( v \),” or “\( v \) is reachable from \( u \)). And since \( v \) is an ancestor of \( u \) (by def of back edge), \( v \rightarrow u \) (again should be wiggly). So \( v \) and \( u \) must be part of a cycle. QED.

Back Edge Detection

How can we detect back edges with DFS? For undirected graphs, easy: see if we’ve visited the vertex before, i.e. color \( \neq \) WHITE.

For directed graphs: Recall that we color a vertex GRAY while its adjacent vertices are being explored. If we re-visit the vertex while it is still GRAY, we have a back edge.

We blacken a vertex when its adjacency list has been examined completely. So any edges to a BLACK vertex cannot be back edges.
TOPOLOGICAL SORT

“Sort” the vertices so all edges go left to right.

For topological sort to work, the graph $G$ must be a DAG (directed acyclic graph). $G$’s undirected version (i.e. the version of $G$ with the “directions” removed from the edges) need not be connected.

Theorem: Listing a dag’s vertices in reverse order of finishing time (i.e. from highest to lowest) yields a topological sort.

Implementation: modify DFS to stick each vertex onto the front of a linked list as the vertex is finished. see examples next slide....

More on Topological Sort

Theorem (again): Listing a dag’s vertices in order of highest to lowest finishing time results in a topological sort. Putting it another way: If there is an edge $(u,v)$, then $f[u] > f[v]$.

Proof: Assume there is an edge $(u,v)$.

Case 1: DFS visits $u$ first. Then $v$ will be visited and finished before $u$ is finished, so $f[u] > f[v]$.

Case 2: DFS visits $v$ first. There cannot be a path from $v$ to $u$ (why not?), so $v$ will be finished before $u$ is even discovered. So again, $f[u] > f[v]$.

QED.
Introduction

Start with a connected, undirected graph, and add real-valued weights to each edge.

The weights could indicate time, distance, cost, capacity, etc.

Definitions

A spanning tree of a graph $G$ is a tree that contains every vertex of $G$.

The weight of a tree is the sum of its edges' weights.

A minimal spanning tree is a spanning tree with lowest weight. (The left tree is not minimal. The right one is, as we will see.)

An Application of MSTs

Wire up a network among several computers so that every computer can reach (directly or indirectly) every other. Use the minimum amount of cable.

Vertices = computers
Weights = distance between computers

Another example: think about inter-connecting a set of $n$ pins, using $n-1$ wires, using the minimal amount of wire.

The MST Property: Intro

Divide the vertices of a graph into two sets (this is a cut of the graph).

Consider an edge of lowest weight on the cut (e.g. 4, above).
This edge is in some MST of the graph.
**Proving the MST Property: 1**

Recall: A tree is an acyclic, connected, undirected graph.

**Lemma 1**: Adding an edge to a tree results in a cycle.

**Proof**: Say the edge is from $u$ to $v$. There was already a path from $v$ to $u$ (since the tree is connected); now there is an edge from $u$ to $v$, forming a cycle.

---

**Proving the MST Property: 2**

**Lemma 2**: Adding an edge to a tree, then removing a different edge along the resulting cycle, still results in a tree.

**Proof**: Omitted.

---

**The MST Property**

**Theorem**: Given a cut of a graph, a lowest-weight edge crossing the cut will be in some MST for the graph.

**Proof**: By contradiction.
Assume an MST on the graph containing no lowest-weight edge crossing the cut. Then some other, higher-weight edge on the cut is in the MST (since it is a spanning tree). (continued)

---

**MST Property Proof, Cont.**

If we now add the lowest-weight edge to the supposed MST, we have a cycle. Removing the higher-weight one still results in a spanning tree, and it has a lower total weight than the alleged MST. Thus the assumption is false: some lowest-weight edge is in the MST. ■
MST Algorithms

Both of the MST algorithms we will study exploit the MST property.
- Kruskal’s: repeatedly add the lowest-weight legal edge to a growing MST.
- Prim’s (really Prim-Jarvik): keep track of a cut, and add the lowest-weight edge across the cut to the growing MST.

Kruskal’s Algorithm

Choose the lowest-weight edge in the graph. This is certainly the lowest weight edge for some cut, so it must be in an MST.
Continue taking the lowest-weight edge and adding it to the MST, unless it would result in a cycle.

Kruskal’s Example

KRUSKAL CODE

MST = Kruskal(G, w)
1. A ← Ø \( \triangleright \) A will be the set of edges that is the MST
2. for each vertex \( v \in V[G] \)
3. \quad do Make-Set(\( v \))
4. \quad sort the edges of \( E \) by nondecreasing weight \( w \)
5. \quad for each edge \( (u, v) \in E \), in order by nondecreasing weight
6. \quad do if Find-Set(\( u \)) \( \neq \) Find-Set(\( v \))
7. \quad \quad then \( A \leftarrow A \cup \{(u, v)\} \triangle \) add \( (u, v) \) to the MST
8. \quad \quad \quad \text{Union}(u, v)
9. return \( A \)
Aside: The Union-Find Problem

(This relates to the running time of Kruskal...) Given several items, in disjoint sets, support three operations:
- Make-Set(x): create a set containing x
- Find-Set(x): return the set to which x belongs
- Union(x, y): take the union of sets of x and y

The “disjoint-set forests” implementation of union-find, with certain heuristics, yields outstanding performance. The running time of these operations is (for all practical purposes) linear in m (the total number of operations). See section 21.3 for more details...

Fast Union-Find

Choose an item as the representative of its set.
- Find-Set(x): return x’s representative
- Union(x, y): make x’s representative point to y

With path compression:
- Union(x, y): make x and everything on the path to its representative point to y

Time for n Union-finds: \( \Theta(n \lg^* n) \).

\( \lg^* n \) is the iterated logarithm function: the number of times you must take \( \lg \) in order to get \( \leq 1 \).
For all practical purposes, \( \lg^* n < 4 \).

Running Time of Kruskal’s

\[
\text{MST} = \text{Kruskal}(G, w) \\
1. \ A \leftarrow \emptyset \quad (A \text{ will be the set of edges that is the MST}) \\
2. \text{for each vertex } v \in V(G) \\
3. \quad \text{do Make-Set}(v) \\
4. \quad \text{sort the edges of } E \text{ by nondecreasing weight } w \\
5. \quad \text{for each edge } (u, v) \in E \text{, in order by nondecreasing weight} \\
6. \quad \text{if Find-Set(u)} \neq \text{Find-Set(v)} \\
7. \quad \quad \text{then } \ A \leftarrow A \cup \{(u, v)\} \quad \text{do add (u, v) to the MST} \\
8. \quad \quad \text{Union}(u, v) \\
9. \quad \text{return } A \\
\]

Total: \( \Theta(E \lg E) \) time.

Prim’s Algorithm

Begin with any vertex. Choose the lowest weight edge connected to that vertex.

Add the vertex on the other side of that edge to the “active set.”

Again, choose the lowest-weight edge of any vertex in the active set that connects to a vertex outside the set, and repeat.
Prim's Example

Why Prim's Works

It comes right out of the MST property.

There are always two sets, the active set and everything else, making a cut. The MST property says that the lowest-weight edge crossing the cut is in an MST. This is just the edge Prim's chooses.

High-level Code for Prim's

MST - Prim(G)
S ← {any vertex in V[G]} \( \triangleright \) active set
T ← \( \emptyset \) \( \triangleright \) set of edges in MST (\( \emptyset \) = empty set)
while S \( \neq \) V[G] do
choose (u, v), a lowest - weight edge such that 
\( u \in S \), and \( v \in V[G] - S \)
T ← T \( \cup \) {(u, v)}
S ← S \( \cup \) {v}
return T

High-Level Prim (Cont.)

How can we choose an edge without looking through every edge in the graph?

Use a priority queue. But make it a queue of vertices, not edges.
A Sophisticated Implementation of Prim’s

1. Build a minimizing priority queue from the graph’s vertices, where the key of a vertex (the value minimized) is the weight of the lowest-weight edge from that vertex to a vertex in the MST.

2. Repeatedly do the following, until the queue is empty:
   a. Extract the minimum vertex from the queue. Call it u.
   b. For each vertex v adjacent to u that is still in the queue, if the weight of the edge between u and v is smaller than key[v], then update key[v] to that weight, and set the parent of v to be u.

Prim’s à la CLRS

MST - Prim(G, w, r)
1. \( Q \leftarrow V(G) \)
2. for each \( v \in Q \)
3. \( \text{do key}[v] \leftarrow \infty \)
4. \( \text{key}[r] \leftarrow 0 \)
5. \( \pi[r] \leftarrow \text{NIL} \)
6. while \( Q \neq \emptyset \)
7. \( \text{do } u \leftarrow \text{EXTRACT-MIN}(Q) \)
8. \( \text{for each } v \in \text{Adj}[u] \)
9. \( \text{do if } v \in Q \text{ and } \text{w}(u, v) < \text{key}[v] \)
10. \( \text{then } \pi[v] \leftarrow u \)
11. \( \text{key}[v] \leftarrow \text{w}(u, v) \)

Choose (A) to be the root.

<table>
<thead>
<tr>
<th>key</th>
<th>π</th>
<th>w (min vertex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>∞</td>
<td>6</td>
</tr>
<tr>
<td>C</td>
<td>∞</td>
<td>5</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td>E</td>
<td>∞</td>
<td>5</td>
</tr>
<tr>
<td>F</td>
<td>∞</td>
<td>6</td>
</tr>
</tbody>
</table>

CLRS Prim
Algorithm Amsterdam

Running Time of Prim

MST - Prim( G, w, r )
1 Q ← V[G ]
2 for each v ∈ Q
3 key [v] ← ∞
4 k[v] ← 0
5 π [r] ← NIL
6 while Q ≠ ∅
7 do u ← EXTRACT-MIN(Q) θ(v) iterations
8 for each v ∈ Adj[u] θ(E) total
9 do if v ∈ Q and w(u, v) < key[v] θ(1) per iteration
10 then π [v] ← u
11 key[v] ← w(u, v) O(1) per iteration

Running Time of Prim (Cont.)

Notes:
- The v ∈ Q test can be done in constant time
  by associating a bit with each vertex
  indicating whether or not it is in the queue.
- Line 11 is a call to DECREASE-KEY.
  That can be done in θ(lgn) time worst case
  from a heap of n items.
Result: O(V + VlgV + ElgV) = O(VlgV + ElgV) = O(ElgV)
Same as Kruskal’s algorithm.

Fibonacci Heaps

It is possible to do better. A Fibonacci Heap is a priority
queue implementation that can do
- EXTRACT-MIN in Θ(lgn)
- DECREASE-KEY in Θ(1)
  amortized time. (See CLRS, Chapter 20).
Improves the time from Θ(ElgV) to Θ(VlgV + E). (We had
Θ(VlgV + ElgV), but the second lgV becomes 1.)
This is better for dense graphs (E ≈ V²).
Theoretical interest only —Fibonacci heaps have a
large constant.
The Problems
Given a directed graph $G$ with edge weights, find
- The shortest path from a given vertex $s$ to all other vertices (Single Source Shortest Paths)
- The shortest paths between all pairs of vertices (All Pairs Shortest Paths)
where the length of a path is the sum of its edge weights.

Shortest Paths: Applications
- Flying times between cities
- Distance between street corners
- Cost of doing an activity
  - Vertices are states
  - Edge weights are costs of moving between states

Shortest Paths: Algorithms
Single-Source Shortest Paths (SSSP)
- Dijkstra’s
- Bellman-Ford
- DAG Shortest Paths
All-Pairs Shortest Paths (APSP)
- Floyd-Warshall
- Johnson’s

A Fact About Shortest Paths
**Theorem:** If $p$ is a shortest path from $u$ to $v$, then any subpath of $p$ is also a shortest path.

**Proof:** Consider a subpath of $p$ from $x$ to $y$. If there were a shorter path from $x$ to $y$, then there would be a shorter path from $u$ to $v$. 

[Diagram of paths with a question mark indicating the comparison of paths.]
Single-Source Shortest Paths

Given a directed graph with weighted edges, what are the shortest paths from some source vertex $s$ to all other vertices?

Note: shortest path to single destination cannot be done asymptotically faster, as far as we know.

Path Recovery

We would like to find the path itself, not just its length.

We’ll construct a shortest-paths tree:

Shortest-Paths Idea

$\delta(u,v) =$ length of the shortest path from $u$ to $v$.

All SSSP algorithms maintain a field $d[u]$ for every vertex $u$. $d[u]$ will be an estimate of $\delta(s,u)$. As the algorithm progresses, we will refine $d[u]$ until, at termination, $d[u] = \delta(s,u)$. Whenever we discover a new shortest path to $u$, we update $d[u]$.

In fact, $d[u]$ will always be an overestimate of $\delta(s,u)$:

$d[u] \geq \delta(s,u)$

We’ll use $\pi[u]$ to point to the parent (or predecessor) of $u$ on the shortest path from $s$ to $u$. We update $\pi[u]$ when we update $d[u]$.

SSSP Subroutine

RELAX($u$, $v$, $w$)

$\triangleright$ (Maybe) improve our estimate of the distance to $v$

$\triangleright$ by considering a path along the edge $(u,v)$.

if $d[u] + w(u,v) < d[v]$ then

$d[v] \leftarrow d[u] + w(u,v)$ \textbf{$\triangleright$ actually, DECREASE-KEY}

$\pi[v] \leftarrow u$ \textbf{$\triangleright$ remember predecessor on path}$

$\delta(u,v)$

$\pi[u]$
Dijkstra’s Algorithm
(pronounced “DIKE-stra”)
Assume that all edge weights are $\geq 0$.
Idea: say we have a set $K$ containing all vertices whose shortest paths from $s$ are known (i.e. $d[u] = \delta(s, u)$ for all $u$ in $K$).
Now look at the “frontier” of $K$—all vertices adjacent to a vertex in $K$.

At each frontier vertex $u$, update $d[u]$ to be the minimum from all edges from $K$.
Now pick the frontier vertex $u$ with the smallest value of $d[u]$.
Claim: $d[u] = \delta(s, u)$

Dijkstra’s: Proof
By construction, $d[u]$ is the length of the shortest path to $u$ going through only vertices in $K$.
Another path to $u$ must leave $K$ and go to $v$ on the frontier.
But the length of this path is at least $d[v]$, (assuming non-negative edge weights), which is $\geq d[u]$.

Proof Explained
Why is the path through $v$ at least $d[v]$ in length?
We know the shortest paths to every vertex in $K$.
We’ve set $d[v]$ to the shortest distance from $s$ to $v$ via $K$.
The additional edges from $v$ to $u$ cannot decrease the path length.
Dijkstra's Algorithm, Rough Draft

\[ K \leftarrow \{s\} \]

- Update \( d \) for frontier of \( K \)
- \( u \leftarrow \) vertex with minimum \( d \) on frontier
- we now know \( d[u] = \delta(s, u) \)
- \( K \leftarrow K \cup \{u\} \)

repeat until all vertices are in \( K \).

Example of Dijkstra’s

A Refinement

Note: we don’t really need to keep track of the frontier.

When we add a new vertex \( u \) to \( K \), just update vertices adjacent to \( u \).

Code for Dijkstra’s Algorithm

\begin{verbatim}
1 DIJKSTRA(G, w, s) => Graph, weights, start vertex
2 for each vertex v in V[G] do
3 \( d[v] \leftarrow \infty \)
4 \( \pi[v] \leftarrow \text{NIL} \)
5 \( d[s] \leftarrow 0 \)
6 Q \leftarrow \text{BUILD-PRIORITY-QUEUE(V[G])}
7 \( \triangleright Q \text{ is } V[G] - K \)
8 while Q is not empty do
9 \( u = \text{EXTRACT-MIN}(Q) \)
10 for each vertex v in Adj[u] do
11 \( \text{RELAX}(u, v, w) \) // DECREASE_KEY
\end{verbatim}
### Running Time of Dijkstra

- **Initialization:** $\Theta(V)$
- **Building priority queue:** $\Theta(V)$
- "while" loop done $|V|$ times
  - $|V|$ calls of EXTRACT-MIN
- Inner "edge" loop done $|E|$ times
  - At most $|E|$ calls of DECREASE-KEY
- **Total time:** $\Theta(V + V \times T_{\text{EXTRACT-MIN}} + E \times T_{\text{DECREASE-KEY}})$

### Dijkstra Running Time (cont.)

1. Priority queue is an array.
   - EXTRACT-MIN in $\Theta(n)$ time, DECREASE-KEY in $\Theta(1)$
   - Total time: $\Theta(V + V + E) = \Theta(V^2)$
2. ("Modified Dijkstra")
   - Priority queue is a binary (standard) heap.
   - EXTRACT-MIN in $\Theta(lgn)$ time, also DECREASE-KEY
   - Total time: $\Theta(Vlgn + Elgn)$
3. Priority queue is Fibonacci heap. (Of theoretical interest only.)
   - EXTRACT-MIN in $\Theta(lgn)$,
   - DECREASE-KEY in $\Theta(1)$ (amortized)
   - Total time: $\Theta(Vlgn+E)$

### The Bellman-Ford Algorithm

- Handles negative edge weights
- Detects negative cycles
- Is slower than Dijkstra

### Bellman-Ford: Idea

Repeatedly update $d$ for all pairs of vertices connected by an edge.

**Theorem:** If $u$ and $v$ are two vertices with an edge from $u$ to $v$, and $s \Rightarrow u \rightarrow v$ is a shortest path, and $d[u] = \delta(s,u)$,

then $d[u] + w(u,v)$ is the length of a shortest path to $v$.

**Proof:** Since $s \Rightarrow u \rightarrow v$ is a shortest path, its length is $\delta(s,u) + w(u,v) = d[u] + w(u,v)$. ■
Why Bellman-Ford Works

- On the first pass, we find $\delta(s,u)$ for all vertices whose shortest paths have one edge.
- On the second pass, the $d[u]$ values computed for the one-edge-away vertices are correct ($= \delta(s,u)$), so they are used to compute the correct $d$ values for vertices whose shortest paths have two edges.
- Since no shortest path can have more than $|V[G]|-1$ edges, after that many passes all $d$ values are correct.
- Note: all vertices not reachable from $s$ will have their original values of infinity. (Same, by the way, for Dijkstra).

Bellman-Ford: Algorithm

```
BELLMAN-FORD(G, w, s)
1   foreach vertex v \in V[G] do //INIT_SINGLE_SOURCE
2       d[v] ← \infty
3       \pi[v] ← NIL
4       d[s] ← 0
5   for i ← 1 to |V[G]|-1 do //each iteration is a "pass"
6       for each edge (u,v) in E[G] do
7           RELAX(u, v, w)
8   for each edge (u,v) in E[G] do
9       if d[v] > d[u] + w(u,v) then
10          return FALSE
11 return TRUE
```

Running time: $\Theta(VE)$

Negative Cycle Detection

What if there is a negative-weight cycle reachable from $s$?
Assume:
- $d[u] \leq d[x]+4$
- $d[v] \leq d[u]+5$
- $d[x] \leq d[v]-10$

Adding:
\[ d[u]+d[v]+d[x] \leq d[x]+d[u]+d[v]-1 \]

Because it's a cycle, vertices on left are same as those on right. Thus we get $0 \leq -1$; a contradiction. So for at least one edge $(u,v)$,
\[ d[v] > d[u] + w(u,v) \]
This is exactly what Bellman-Ford checks for.

SSSP in a DAG

Recall: a dag is a directed acyclic graph.

If we update the edges in topologically sorted order, we correctly compute the shortest paths.

Reason: the only paths to a vertex come from vertices before it in the topological sort.
SSP in a DAG Theorem

Theorem: For any vertex $u$ in a dag, if all the vertices before $u$ in a topological sort of the dag have been updated, then $d[u] = \delta(s,u)$.

Proof: By induction on the position of a vertex in the topological sort.

Base case: $d[s]$ is initialized to 0.

Inductive case: Assume all vertices before $u$ have been updated, and for all such vertices $v$, $d[v] = \delta(s,v)$. (continued)

Proof, Continued

Some edge $(v,u)$ where $v$ is before $u$, must be on the shortest path to $u$, since there are no other paths to $u$.

When $v$ was updated, we set $d[u]$ to

$$d[u] = d[v] + w(v,u)$$

$$= \delta(s,v) + w(v,u)$$

$$= \delta(s,u) \blacksquare$$

SSP-DAG Algorithm

DAG-SHORTEST-PATHS(G,w,s)
1 topologically sort the vertices of G
2 initialize $d$ and $\pi$ as in previous algorithms
3 for each vertex $u$ in topological sort order do
4 for each vertex $v$ in Adj[$u$] do
5 RELAX($u$, $v$, $w$)

Running time: $\Theta(V+E)$, same as topological sort

All-Pairs Shortest Paths

We now want to compute a table giving the length of the shortest path between any two vertices. (We also would like to get the shortest paths themselves.)

We could just call Dijkstra or Bellman-Ford $|V|$ times, passing a different source vertex each time.

It can be done in $\Theta(V^3)$, which seems to be as good as you can do on dense graphs.
Doing APSP with SSSP

Dijkstra would take time
\[ \Theta(V \times V^2) = \Theta(V^3) \] (standard version)
\[ \Theta(V \times (V\lg V + E)) = \Theta(V^2\lg V + VE) \]
(modified, Fibonacci heaps),
but doesn’t work with negative-weight edges.
Bellman-Ford would take \( \Theta(V \times VE) = \Theta(V^2E) \).

The Floyd-Warshall Algorithm

Represent the directed, edge-weighted graph in adjacency-matrix form.
\[
\begin{bmatrix}
 w_{11} & w_{12} & w_{13} \\
 w_{21} & w_{22} & w_{23} \\
 w_{31} & w_{32} & w_{33}
\end{bmatrix}
\]
• \( w_{ij} \) is the weight of edge \((i, j)\), or \( \infty \) if there is no such edge.
• Return a matrix \( D \), where each entry \( d_{ij} \) is \( \delta(i, j) \).
• Could also return a predecessor matrix, \( P \), where each entry \( \pi_{ij} \) is the predecessor of \( j \) on the shortest path from \( i \).

Floyd-Warshall: Idea

Consider intermediate vertices of a path:

\[ i \rightarrow \cdots \rightarrow j \]

Say we know the length of the shortest path from \( i \) to \( j \) whose intermediate vertices are only those with numbers 1, 2, ..., \( k-1 \). Call this length \( d_{ij}^{(k-1)} \)

Now how can we extend this from \( k-1 \) to \( k \)? In other words, we want to compute \( d_{ij}^{(k)} \). Can we use \( d_{ij}^{(k-1)} \), and if so how?

Floyd-Warshall Idea, 2

Two possibilities:
1. Going through the vertex \( k \) doesn’t help—the path through vertices 1...\( k-1 \) is still the shortest.
2. There is a shorter path consisting of two subpaths, one from \( i \) to \( k \) and one from \( k \) to \( j \).
Each subpath passes only through vertices numbered 1 to \( k-1 \).
Floyd-Warshall Idea, 3

Thus,

\[ d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) \]

Also, \( d_{ij}^{(0)} = w_{ij} \)

(since there are no intermediate vertices.)

When \( k = |V| \), we're done.

Dynamic Programming

Floyd-Warshall is a dynamic programming algorithm:

Compute and store solutions to sub-problems. Combine those solutions to solve larger sub-problems.

Here, the sub-problems involve finding the shortest paths through a subset of the vertices.

Code for Floyd-Warshall

Floyd-Warshall(W)

1. \( n \leftarrow \text{rows}(W) \) // number of vertices
2. \( D^{(0)} \leftarrow W \)
3. for \( k \leftarrow 1 \) to \( n \)
4. \hspace{1em} do for \( i \leftarrow 1 \) to \( n \)
5. \hspace{2em} do for \( j \leftarrow 1 \) to \( n \)
6. \hspace{3.5em} \( D^{(k)} \leftarrow \min(D^{(k-1)}, D^{(k-1)} + D^{(k-1)}) \)
7. return \( D^{(n)} \)

Running time: a zippy \( \Theta(n^3) \). (Small constant of proportionality, because operations are simple.)
Johnson's Algorithm

Makes clever use of Bellman-Ford and Dijkstra to do All-Pairs-Shortest-Paths efficiently on sparse graphs.

Motivation: By running Dijkstra $|V|$ times, we could do APSP in time $\Theta(V^2 \log V + VE \log V)$ (Modified Dijkstra), or $\Theta(V^2 \log V + V^2)$ (Fibonacci Dijkstra). This beats $\Theta(V^3)$ (Floyd-Warshall) when the graph is sparse.

Problem: negative edge weights.

The Basic Idea

Reweight the edges so that:
1. No edge weight is negative.
2. Shortest paths are preserved. (A shortest path in the original graph is still one in the new, reweighted graph.)

An obvious attempt: subtract the minimum weight from all the edge weights. E.g. if the minimum weight is -2:

\[-2 - (-2) = 0\]
\[-3 - (-2) = 5\]

Counterexample

Subtracting the minimum weight from every weight doesn’t work.

Consider:

Paths with more edges are unfairly penalized.

Johnson's Insight

Add a vertex $s$ to the original graph $G$, with edges of weight 0 to each vertex in $G$:

Assign new weights $\hat{w}$ to each edge as follows:

$$\hat{w}(u, v) = w(u, v) + \delta(s, u) - \delta(s, v)$$
**Question 1**

Are all the $\hat{w}$'s non-negative? Yes:

$$\hat{\delta}(s,u) + \delta(u,v) \geq \delta(s,v)$$

Otherwise, $s \Rightarrow u \rightarrow v$ would be shorter than the shortest path from $s$ to $v$.

\[ w(u,v) + \hat{\delta}(u,v) - \delta(u,v) \geq 0 \]

Rewriting:

\[ \hat{w}(u,v) \]

**Question 2**

Does the reweighting preserve shortest paths? Yes: Consider any path $p = v_1, v_2, \ldots, v_k$

\[ \hat{w}(p) = \sum_{i=1}^{k} \hat{w}(v_i, v_{i+1}) \]

\begin{align*}
\delta(v_1, v_k) &+ \delta(v_1, v_2) - \delta(v_2, v_3) + \delta(v_2, v_3) - \delta(v_3, v_4) + \cdots + \delta(v_{k-1}, v_k) - \delta(v_k, v_1) \\
&= \delta(v_1, v_k) - \delta(v_k, v_1)
\end{align*}

A value that depends only on the endpoints, not on the path.

In other words, we have adjusted the lengths of all paths by the same amount. So this will not affect the relative ordering of the paths—shortest paths will be preserved.

**Question 3**

How do we compute the $\delta(s,v)$'s?

Use Bellman-Ford.

This also tells us if we have a negative-weight cycle.

**Johnson's: Algorithm**

1. Compute $G'$, which consists of $G$ augmented with $s$ and a zero-weight edge from $s$ to every vertex in $G$.
2. Run Bellman-Ford($G'$, $w$, $s$) to obtain the $\hat{\delta}(s,v)$'s
3. Reweight by computing $\hat{w}$ for each edge
4. Run Dijkstra on each vertex to compute $\hat{\delta}$
5. Undo reweighting factors to compute $\delta$
Johnson’s: CLRS

\[ \text{JOHNSON}(E) \]
1. Compute \( G' \), where \( V[G'] = V[G] \cup \{s\} \),
   \[ E[G'] = E[G] \cup \{(s, v) : v \in V[G]\} \], and
   \[ w(s, v) = 0 \text{ for all } v \in V[G] \]
2. If \( \text{BELLMAN-FORD}(G', w, s) = \text{FALSE} \)
   then print “the input graph contains a negative weight cycle”
   else for each vertex \( v \in V[G] \)
      do set \( L(v) \) to the value of \( \Delta(v, s) \)
          computed by the Bellman-Ford algorithm
3. For each edge \((u, v) \in E[G']\)
   do \( \tilde{w}(u, v) = w(u, v) + \Delta(u) - \Delta(v) \)
4. For each vertex \( u \in V[G] \)
   do \( \tilde{d}(u) \leftarrow \tilde{w}(u, s) + \tilde{L}(u) - L(s) \)
5. Return \( D \)

Johnson: reweighting

\[ \tilde{w}(u, v) = w(u, v) + d(s, u) - d(s, v) \]

Johnson using Dijkstra

Johnson’s: Running Time

1. Computing \( G' \): \( \Theta(V) \)
2. Bellman-Ford: \( \Theta(VE) \)
3. Reweighting: \( \Theta(E) \)
4. Running (Modified) Dijkstra: \( \Theta(V^2 \log V + VE \log V) \)
5. Adjusting distances: \( \Theta(V^2) \)

Total is dominated by Dijkstra: \( \Theta(V^2 \log V + VE \log V) \)