# Algorithms 2

## Section 1: Graphs and Path-Finding Algorithms

# Graphs

A graph, G = (V, E), is a set of vertices and edges $\subseteq$ V x V. We usually care about finite graphs.

**Directed?** In an undirected graph, the edges are unordered pairs (or E is symmetric); directed graphs have ordered pairs of vertices in the edge set.

**Weighted?** A weighted graph has a function E $\rightarrow \mathbb{R}$ that associates a weight with each edge.

A graph is **fully connected** if E = V x V.

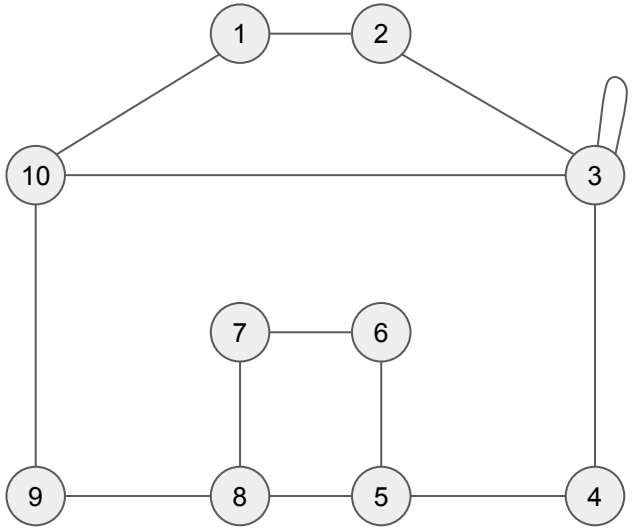
# Representing a graph

There are two basic representations of E: adjacency lists and adjacency matrices.

A |V| x |V| **adjacency matrix**, M is $\Theta(|V|^2)$ in size. If G is unweighted, $M_{u,v}$ = 1 if (u, v) $\in$ E and 0 otherwise. In weighted graphs, $M_{u,v}$ holds the weight of edge (u, v). If G is undirected, we only need to store the upper (or lower) triangle of M.

**Adjacency lists** are stored in an array of length |V| where A[u] stores a pointer to a linked list of the vertices, v $\in$ V such that (u, v) $\in$ E. If G is weighted, the lists store tuples (v, w) such that (u, v) $\in$ E and weight(u, v) = w.

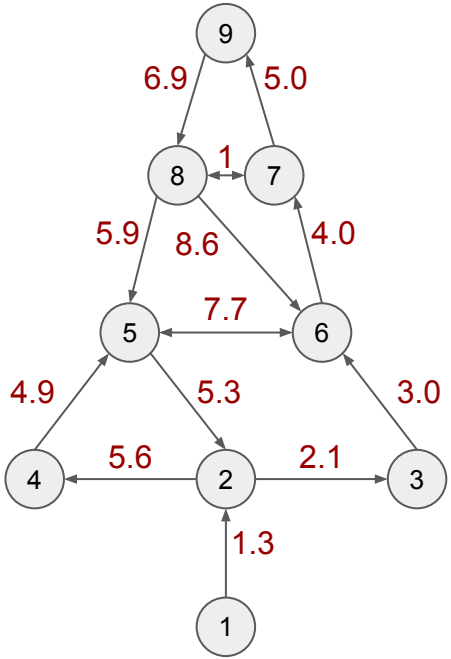
# Example Adjacency Matrix

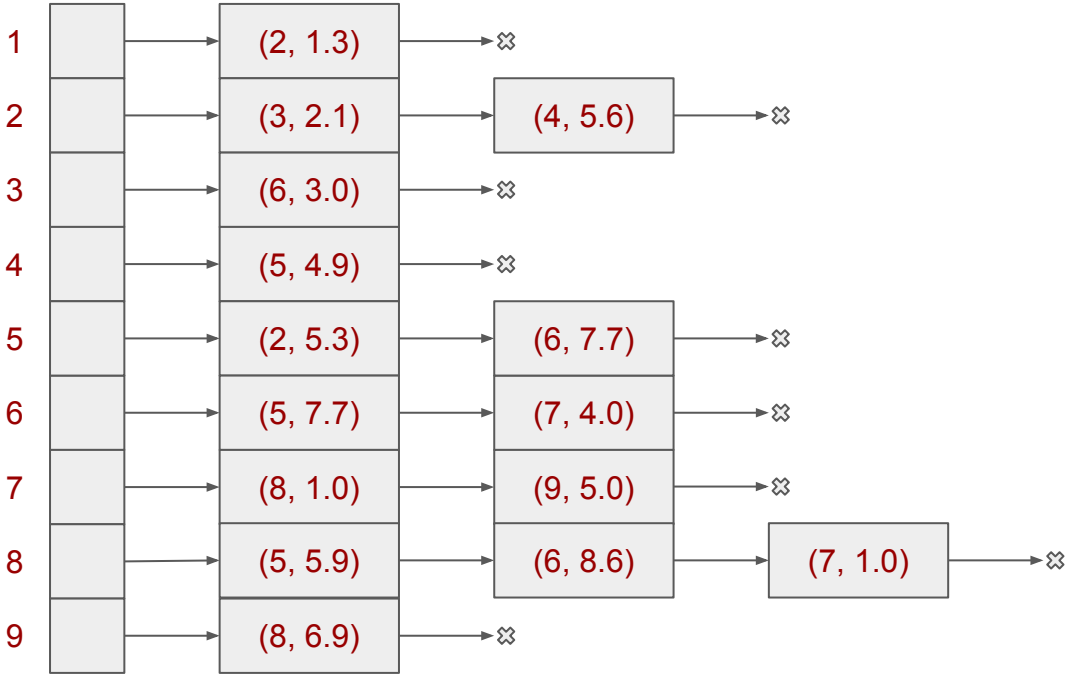



| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | | 1 | | | | | | | | 1 |
| 2 | 1 | | 1 | | | | | | | |
| 3 | | 1 | 1 | 1 | | | | | | 1 |
| 4 | | | 1 | | 1 | | | | | |
| 5 | | | | 1 | | 1 | | 1 | | |
| 6 | | | | | 1 | | 1 | | | |
| 7 | | | | | | 1 | | 1 | | |
| 8 | | | | | 1 | | 1 | | 1 | |
| 9 | | | | | | | | 1 | | 1 |
| 10 | 1 | | 1 | | | | | | 1 | |

💡 This undirected graph is represented with an adjacency matrix. The shaded cells do not need to be stored due to symmetry.

# Example Adjacency Lists




💡 This weighted, directed graph is represented with adjacency lists.

# Comparison of Adjacency Matrices and Adjacency Lists

| **Adjacency Matrices** | **Adjacency Lists** |
| --- | --- |
| Compact for dense graphs (no pointers, and only 1-bit per entry if unweighted) | Compact for sparse graphs |
| O(1) check whether $(u,v) \in E$ | O(\|V\|) check whether $(u,v) \in E$ |
| O(\|V\|) to list neighbouring nodes | O(num neighbours) to list neighbouring nodes |
| Can (approx) halve storage if G is undirected | Cannot halve storage for undirected graphs (without significant worsening of time complexity) |
| $O(\|V\|^2)$ to iterate through all edges | O(\|E\|) to iterate through all edges |

# Other terminology [1]

The **transpose** of a directed graph G = (V, E) is the graph $G^T = (V, E^T)$, which (by transposing the edge matrix) has all the directed edges reversed.

The **in-degree** and **out-degree** of a vertex in a directed graph are the numbers of incoming and outgoing edges, respectively. The **degree** of a vertex (in a directed or undirected graph) is the number of edges incident at that vertex.

The **square** of a graph G = (V, E) is the graph $G^2 = (V, E^2)$, in which an edge (u,v) is present if there is a path between u and v in G consisting of at most two edges.

Two *edges* are **adjacent** if they share a vertex.

# Other terminology [2]

A **complete graph** (also **fully connected graph**) is one with $E = V \times V$.

A **connected graph** is one where every pair of vertices are connected by at least one *path* (not edge!).

A **induced subgraph** of $G = (V, E)$ is another graph $G' = (V', E')$ where $V' \subseteq V$ and $E'$ is that subset of $E$ consisting of all edges $(u, v) \in E$ where $u,v \in V'$.

A **clique** within a graph G is any induced subgraph that is complete.

The **complement graph** of $G = (V, E)$ is the graph $G = (V, \bar{E})$ where $\bar{E} = \{ (u, v) \mid u,v \in V \land (u, v) \notin E \}$.

A graph is **acyclic** if no vertex can be reached by a path from itself.

# Other terminology [3]

**Vertex colouring** is the task of assigning colours to each $v \in V$ such that no adjacent vertices have the same colour.

**Edge colouring** is the task of assigning colours to each edge $e \in E$ such that no adjacent edges have the same colour.

**Face colouring** is the task of assigning colours to each face of a planar graph such that no adjacent faces have the same colour. A **planar graph** can be drawn on a plane such that no two edges intersect (other than at their vertices). A **face** is a region bounded by edges (including the infinite-area region around the 'outside').

# Breadth First Search, BFS(G, s)

BFS can be used on directed and undirected graphs.

BFS on a graph is slightly more complex than on a tree because we have to worry about duplicate ‘discoveries’ of a vertex.

s is the source vertex (where the exploration begins).

# BFS(G, s) – for trees!

```
1   for v in G.V
2       v.marked = false
3   Q = new Queue
4   ENQUEUE(Q, s)
```

```
5   while !QUEUE-EMPTY(Q)
6       u = DEQUEUE(Q)
7       u.marked = true
8       for v in u.adjacent
9           ENQUEUE(Q, v)
```

Line 7 is a placeholder. You should 'process' node u in whatever way makes sense for your algorithm. Marking a node to say we've been here is a trivial thing to do (and pointless if s is the root because we'll visit everywhere in the tree so all vertices will end up marked).

💡 Why doesn't this work for graphs?

# BFS(G, s) – for graphs?

```
1   for v in G.V
2       v.marked = false
3   Q = new Queue
4   ENQUEUE(Q, s)
```

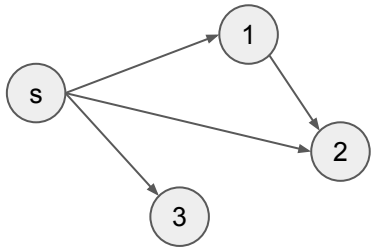
```
5   while !QUEUE-EMPTY(Q)
6       u = DEQUEUE(Q)
7       if (!u.marked)
8           u.marked = true
9           for v in G.E.adj[u]
10              ENQUEUE(Q, v)
```

When this terminates, all nodes reachable from s will have been marked (or ‘processed’ in any other way your algorithm wishes to process them at line 8).

💡 Why is this inefficient for graphs (in general)?

# BFS(G, s) on graphs

The previous algorithm does work but is inefficient because it can enqueue vertices more than once, wasting memory (and some time when dequeuing).




Enqueue s
Dequeue s → Mark s, Enqueue 1, 2, and 3. Q=[1,2,3]
Dequeue 1 → Mark 1, Enqueue 2. Q=[2,3,2]
The memory consumption for the queue is more than necessary.
(Line 7 prevents infinite looping on cyclic input.)

To fix this, we need to record when a vertex has been inserted into the queue already and avoid inserting a second time. Searching the queue would be O(q) in the length, q, of the queue so let's try a bit harder…

# BFS(G, s) – for graphs!

```
1   for v in G.V
2       v.marked = false
3       v.pending = false
4   s.pending = true
5   Q = new Queue
6   ENQUEUE(Q, s)
```

```
7   while !QUEUE-EMPTY(Q)
8       u = DEQUEUE(Q)
9       u.marked = true
10      for v in G.E.adj[u]
11          if !v.pending
12              ENQUEUE(Q, v)
13              v.pending = true
```

Or other processing

💡 The expected running time is O(|V|) for lines 1–3 and O(|E|) for 7–13 so O(|V|+|E|) overall.

💡 Why is line 4 necessary? Provide a graph that would cause this algorithm to go wrong without line 4.

# BFS(G, s) – with immutable graphs

```
1   let M = new HashTable
2   let P = new HashTable
3   HASH-INSERT(P, s)
4   Q = new Queue
5   ENQUEUE(Q, s)
```

```
6   while !QUEUE-EMPTY(Q)
7       u = DEQUEUE(Q)
8       HASH-INSERT(M, u)
9       for v in G.E.adj[u]
10          if !HASH-HAS-KEY(P,v)
11              ENQUEUE(Q, v)
12              HASH-INSERT(P, v)
```

Or other processing

This program puts all vertices reachable from s into the hash table M but you could do any other processing you like at line 8.

💡 We can store the pending set in a hash table if the graph vertices do not have a ‘pending’ attribute or we cannot modify the graph itself (such as in a multithreaded program – see IB Concurrent and Distributed Systems).

# 2-Vertex Colourability (for a connected, undirected graph)

**Input**: a connected, undirected graph, G = (V, E)

**Output**: true if G.V can be coloured using two colours; false otherwise.

Pick an arbitrary vertex, s. Set s.colour = BLACK. BFS from s, colouring the first level as RED, the next level BLACK, etc. ⇒ O(|V| + |E|) = O(|E|) since *connected*.

When the BFS completes, scan over the edges checking whether any adjacent vertices have the same colour. Return true/false as appropriate. ⇒ O(|E|)

O(|V| + |E|) overall – for adjacency list representations.
Both steps and overall are $O(|V|^2)$ if adjacency matrices are used.

⚠️ Implement this! There is a new concept: the “level” of the BFS.

# Single-Source All-Destinations Shortest Paths with BFS

- There are lots of “shortest path” problems and algorithms!
- A simple case concerns…
  - An unweighted graph that can be directed or undirected
  - The concept of “shortest” means fewest edges (hop count)
  - Single source, which is specified as an input to the algorithm
  - We want the path lengths AND the actual paths…
  - …and we want this for every destination
- We expect the source to have a distance of 0 from itself, and the path = [ ]
- We expect any vertices that are unreachable from the source to have distances of ∞ (and paths that are not initialised to any meaningful value).
- If the average path length O(|V|) then the output is $O(|V|^2)$.

# SSAD_HopCount(G, s)

```
1   for v in G.V
2       v.pending = false
3       v.d = ∞
4       v.π = NIL
5   s.pending = true
6   s.d = 0
7   s.π = NIL
8   Q = new Queue
9   Enqueue(Q, s)
10  while !Queue-Empty(Q)
11      u = Dequeue(Q)
12      for v in G.E.adj[u]
13          if !v.pending
14              v.pending = true
15              v.d = u.d + 1
16              v.π = u
17              Enqueue(Q, v)
```

💡 Subtlety! We have not provided the paths, only a data structure from which paths can be extracted. To find the path from s to v, start at v, follow v.π until v.π = NIL, then reverse the list of vertices visited.

# Analysis of SSAD_HopCount(G, s)

Initialisation loop (lines 1–4) costs $\Theta(|V|)$. Lines 5–7 are $O(1)$.

Line 8: initialising a new Queue with max length V takes $O(1)$ to $O(|V|)$ time, depending on how the memory allocator works and the queue implementation.

The WHILE loop and nested FOR loop eventually process every edge at most once (exactly once – the worst case – when G is connected) so this is $O(|E|)$ if we use an adjacency list representation.

Each vertex is enqueued and dequeued (both $O(1)$) at most once: total $O(V)$ time.

Total cost is $O(|V| + |E|)$.

# Analysis of SSAD_HopCount(G, s) with Adjacency Matrix

```
10  while !Queue-Empty(Q)
11      u = Dequeue(Q)
12      for v in G.V
13          if G.E.M[u][v]==1 && !v.pending
```

To use an adjacency matrix, we cannot loop through adjacent vertices on line 12. Instead, we loop through all vertices and process them only if the edge matrix (G.E.M) contains 1 (edge present) in position u,v.

This increases the cost of the loops to $O(|V|^2)$ (not $\Theta(|V|^2)$ because disconnected vertices will never be enqueued/dequeued).

# Correctness of SSAD_HopCount(G, s) [1]

**Goal**: prove that, when SSAD_HopCount terminates, for all v ∈ G.V, v.d is the length of *a* shortest path from s to v. ('a' as equal-shortest paths are possible.)

Let the **shortest-path distance** $\delta(s, v)$ be the actual shortest path length: the minimum number of edges on any path between s and v. If there is no path between s and v, we say that $\delta(s, v) = \infty$.

**Lemma 1**: if (u, v) ∈ G.E then $\delta(s, v) \leq \delta(s, u) + 1$. Proof: if u is unreachable from s then $\delta(s, u) = \infty$ and the inequality holds. If u is reachable then the shortest path to v is either shorter than going via u, or it isn't. If it is via u then the direct edge (u, v) is shorter than any other path from u to v. In all cases, the inequality holds.

# Correctness of SSAD_HopCount(G, s) [2]

Next we prove **Lemma 2**: on termination, for all $v \in G.V$ we have $v.d \geq \delta(s, v)$.

The proof is by induction on the number of ENQUEUE operations performed.

The induction hypothesis is that for all $v \in G.V$ we have $v.d \geq \delta(s, v)$.

**Base case**: immediately before the WHILE loop begins, we have $v.d = \infty$ for all vertices except the source, where $s.d = 0$.

- $\delta(s, s) = 0$ so the hypothesis holds for the source vertex.
- $\infty \geq \delta(s, v)$ so the hypothesis holds for the other vertices (even if disconnected from s).

# Correctness of SSAD_HopCount(G, s) [3]

**Inductive case**: the WHILE/FOR loops only change the value of v.d if v was not pending when the loop began, so non-pending nodes are those we must consider.

The hypothesis tells us that u.d ≥ $\delta$(s, u) and the assignment `v.d = u.d + 1` (line 15) gives us that:

v.d = u.d + 1
≥ $\delta$(s, u) + 1
≥ $\delta$(s, v) by Lemma 1

v.d is never changed again because its pending flag is set on line 14.
The result follows by induction.

💡 This tells us that the algorithm does not set any v.d to be too low but we have not yet proved that it has set any v.d low enough to correspond to a shortest path length.

# Correctness of SSAD_HopCount(G, s) [4]

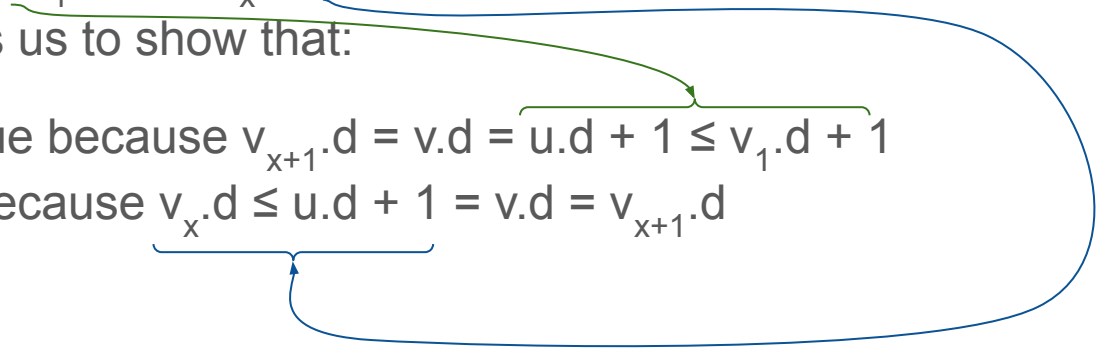
Next we show that the queue only ever contains vertices with at most two different values of v.d, using induction on the number of queue operations. **Lemma 3**: After each call to Enqueue and Dequeue, $\phi$ holds:
$\phi$: if Q = $v_1$, $v_2$, $v_3$, .. $v_x$ (head .. tail) then $v_x.d \leq v_1.d + 1$ and $v_i.d \leq v_{i+1}.d$ for i = 1..x-1

**Dequeue**: if dequeuing $v_1$ leaves the queue empty, then $\phi$ holds vacuously. Otherwise, $v_2$ becomes the new head and we know (from $\phi$ by induction) that $v_1.d \leq v_2.d$ and $v_x.d \leq v_1.d + 1$. The only new inequality that we must validate concerns the new head: $v_x.d \leq v_2.d + 1$. However, $v_x.d \leq v_1.d + 1 \leq v_2.d + 1$ so it follows immediately that the Dequeue operations in the algorithm preserve $\phi$.

# Correctness of SSAD_HopCount(G, s) [5]

**Enqueue**: when we enqueue v (line 17), v.d = u.d + 1 where u was just dequeued and v is one of u's adjacent vertices. When u was dequeued, the induction hypothesis assures us that $u.d \le v_1.d$ and $v_x.d \le u.d + 1$. Enqueuing v makes it $v_{x+1}$ in the queue and $\phi$ requires us to show that:

- $v_{x+1}.d \le v_1.d + 1$ which is true because $v_{x+1}.d = v.d = u.d + 1 \le v_1.d + 1$
- $v_x.d \le v_{x+1}.d$ which is true because $v_x.d \le u.d + 1 = v.d = v_{x+1}.d$



The induction hypothesis, $\phi$, thus holds after every DEQUEUE and ENQUEUE.

# Correctness of SSAD_HopCount(G, s) [6]

A corollary of Lemma 3 is useful: if SSAD_HopCount enqueues $v_a$ before $v_b$ then $v_a.d \leq v_b.d$ on termination.

Proof: vertices are only given a finite value *once* during the execution of the algorithm. Lemma 3 tells us that the 'd' attributes of queued elements are ordered so $v_a.d \leq v_b.d$ on termination. This comes directly from $\phi$ when a and b are in the queue simultaneously, and we appeal to the transitivity of ≤ when a and b are not simultaneously in the queue.

# Correctness of SSAD_HopCount(G, s) [7]

Finally, we can prove the correctness of SSAD_HopCount on a directed or undirected input graph, G. Explicitly, we want to show that the algorithm:

- Really does find all vertices $v \in G.V$ that are reachable from s; and
- Really does terminate with $v.d = \delta(s, v)$ for all $v \in G.V$.

To further prove that the paths discovered are correct, we must additionally show that:

- One of the shortest paths from s to v is a shortest path from s to $v.\pi$ followed by the edge $(v.\pi, v)$.

# Correctness of SSAD_HopCount(G, s) [8]

We use a proof by contradiction. If the algorithm doesn't work then at least one vertex was assigned an incorrect 'd' value. Let v be the vertex with the minimum $\delta$ (s, v) that has an incorrect v.d upon termination.

We can see from line 6 that $v \neq s$.

By Lemma 2, $v.d \geq \delta(s, v)$ so, since there's an error, $v.d > \delta(s, v)$. Furthermore, v must be reachable from s as, otherwise, we would have $\delta(s, v) = \infty \geq v.d$ which contradicts $v.d > \delta(s, v)$.

# Correctness of SSAD_HopCount(G, s) [9]

Let u be the node on a shortest path from s to v that comes immediately before v. $\delta(s, v) = \delta(s, u) + 1$ so $\delta(s, u) < \delta(s, v)$.

Because we chose v to be the incorrect vertex with minimum $\delta(s, v)$, we know that $\delta(s, u) = u.d$ (because $\delta(s, u)$ cannot equal $\delta(s, v)$ so u.d cannot also be incorrect).

$v.d > \delta(s, v) = \delta(s, u) + 1 = u.d + 1$ // this is what we will contradict

Now we consider what happened when u was dequeued.

# Correctness of SSAD_HopCount(G, s) [10]

When u was dequeued, vertex v might have been in one of three states:

1. Not yet been enqueued (pending = false)
2. Enqueued but not yet dequeued (pending = true and v.d = ∞)
3. Already been enqueued and dequeued (v.d is finite)

**Case 1**: if v has not yet been enqueued then v.pending = false so the IF statement executed when u is dequeued and processed will set v.d = u.d + 1. This contradicts v.d > u.d + 1 so vertex v cannot fall under case 1.

# Correctness of SSAD_HopCount(G, s) [11]

**Case 2**: if v is in the queue when u is dequeued and processed then some earlier vertex, w, must have encountered v as an adjacency and enqueued it.

When w was processed, v.d was set to w.d + 1.

We have that $w.d \leq u.d$ by the corollary to Lemma 3.

Hence $v.d = w.d + 1 \leq u.d + 1$.

This contradicts $v.d > u.d + 1$ so vertex v cannot fall under case 2.

# Correctness of SSAD_HopCount(G, s) [12]

**Case 3**: if v has already been dequeued when u is dequeued then v.d ≤ u.d, by the corollary to Lemma 3. This contradicts v.d > u.d + 1 so vertex v cannot fall under case 3.

All three cases yield a contradiction. As there were no mistakes (hopefully!) in the consideration of the three cases, we are left to conclude the mistake must lie in the assumption that led to the three cases, i.e. there can be no v with minimum $\delta$ (s, v) where v.d is incorrect.

If there is no “first time” that the algorithm goes wrong, then it must be correct!

# Correctness of SSAD_HopCount(G, s) [13]

To show that the paths are correct (over and above their lengths being correct), we simply note that the algorithm assigns $v.\pi = u$ whenever it assigns $v.d = u.d + 1$ during the processing of edge (u, v) so, since v.d finishes at the correct value it must be the case that a shortest path from s to v can be obtained by taking any shortest path from s to $v.\pi$ followed by the direct edge from $v.\pi$ to v.

# Predecessor Subgraph

Consider the edges $(v.\pi, v)$ for $v \in G.V \setminus \{s\}$ computed by SSAD_HOPCOUNT(G,s). (We remove s since $s.\pi = \text{NIL} \notin V$ so $(s.\pi, s)$ would not be a valid edge.)

These edges form a tree known as the **breadth-first tree**.

The tree is the predecessor subgraph of G:

$PSG = (V_{PSG}, E_{PSG})$

- $V_{PSG} = \{ v \in G.V \mid v.\pi \neq \text{NIL} \} \cup \{s\}$ // i.e. all vertices reachable from s
- $E_{PSG} = \{ (v.\pi, v) \mid v \in G.V \setminus \{s\} \}$

# Depth First Search, DFS(G)

DFS is similar to BFS but uses a stack instead of a queue, or a recursive implementation can use the call stack to govern the exploration order (next slide).

DFS is often used on undirected graphs and no source vertex is specified: in this case, DFS picks any vertex as the source, explores everything reachable, and repeats with another randomly-chosen (and as yet unvisited) vertex as the source until all vertices have been visited.  This yields a forest (multiple trees).

DFS can produces a depth-first tree augmented with some interesting properties.

Let “time” be a global clock that (effectively) numbers events in exploration order.

## DFS(G)

```
1   for v in G.V
2       v.marked = false
3       v.π = NIL
4   time = 0
5   for s in G.V
6       if !s.marked
7           DFS-HELPER(G, s)
```

## DFS-HELPER(G, u)

```
1   time = time + 1
2   u.discover_time = time
3   u.marked = true
4   for v in G.E.adj[u]
5       if !v.marked
6           v.π = u
7           DFS-HELPER(G, v)
8   time = time + 1
9   u.finish_time = time
```

💡 The running time is Θ(|V| + |E|) because all vertices and edges will eventually be explored.

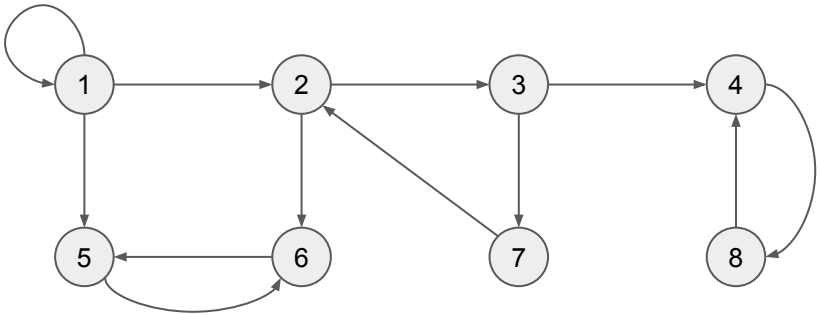
# What’s the time?

v.**discover_time** is the global time value when DFS first considered v.

v.**finish_time** is the global time value when DFS finished recursing into all the descendants of v.

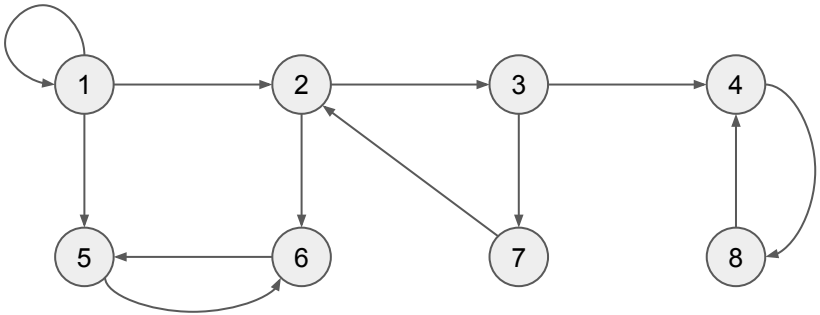
In the depth-first tree, a vertex v is a descendant of u if (and only if) v.discover_time is between u.discover_time and u.finish_time.

# Example DFS



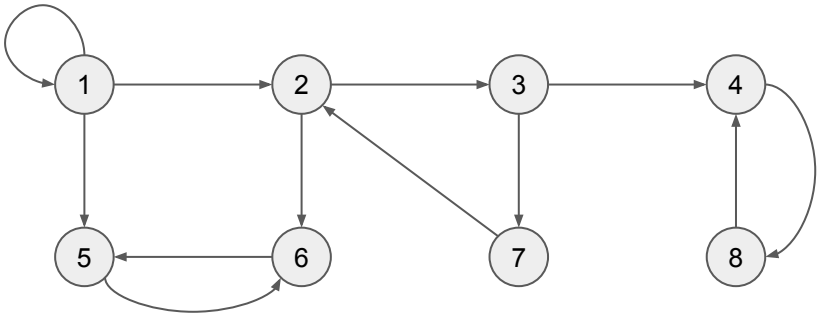

| Node | Discover | Finish |
|---|---|---|
| 1 | 1 | |
| 2 | | |
| 3 | | |
| 4 | | |
| 5 | | |
| 6 | | |
| 7 | | |
| 8 | | |

# Example DFS



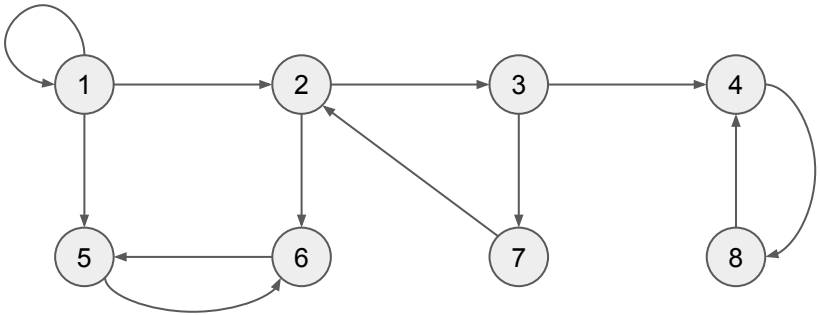

| Node | Discover | Finish |
|---|---|---|
| 1 | 1 | |
| 2 | 2 | |
| 3 | | |
| 4 | | |
| 5 | | |
| 6 | | |
| 7 | | |
| 8 | | |

# Example DFS



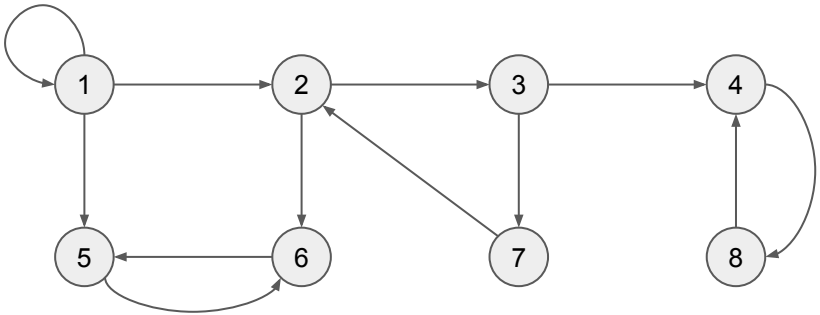

| Node | Discover | Finish |
|---|---|---|
| 1 | 1 | |
| 2 | 2 | |
| 3 | 3 | |
| 4 | | |
| 5 | | |
| 6 | | |
| 7 | | |
| 8 | | |

# Example DFS



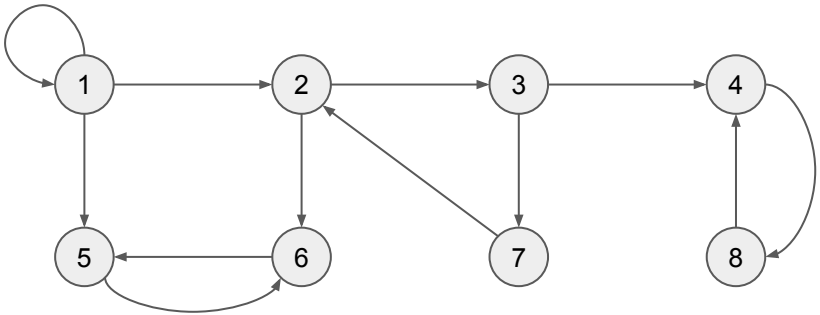

| Node | Discover | Finish |
|---|---|---|
| 1 | 1 | |
| 2 | 2 | |
| 3 | 3 | |
| 4 | 4 | |
| 5 | | |
| 6 | | |
| 7 | | |
| 8 | | |

# Example DFS



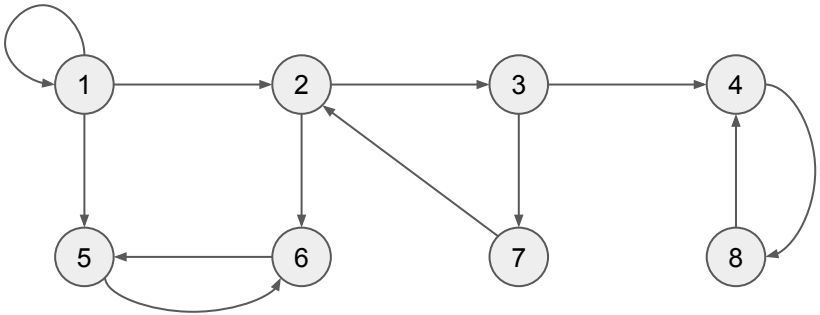

| Node | Discover | Finish |
|---|---|---|
| 1 | 1 | |
| 2 | 2 | |
| 3 | 3 | |
| 4 | 4 | |
| 5 | | |
| 6 | | |
| 7 | | |
| 8 | 5 | |

# Example DFS



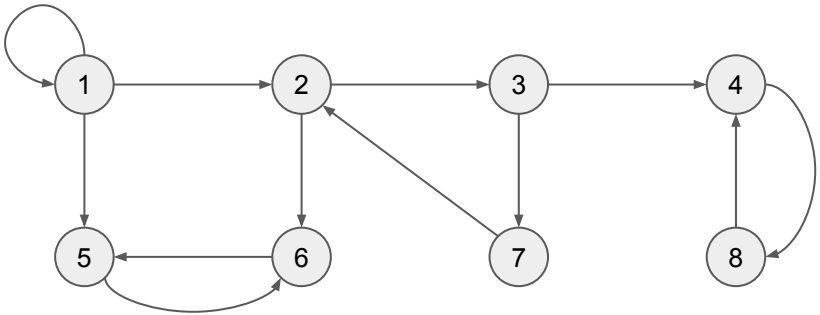

| Node | Discover | Finish |
|---|---|---|
| 1 | 1 | |
| 2 | 2 | |
| 3 | 3 | |
| 4 | 4 | |
| 5 | | |
| 6 | | |
| 7 | | |
| 8 | 5 | 6 |

# Example DFS



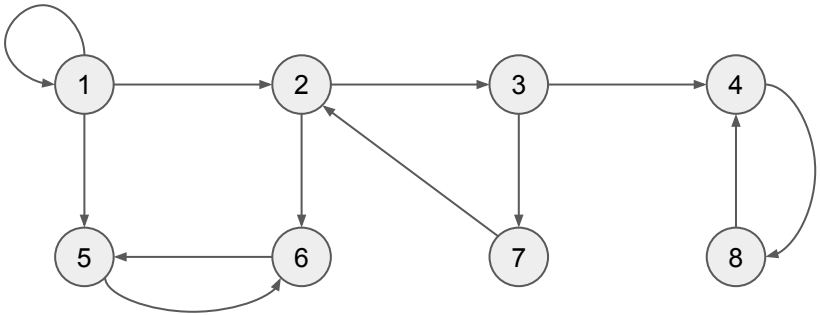

| Node | Discover | Finish |
|---|---|---|
| 1 | 1 | |
| 2 | 2 | |
| 3 | 3 | |
| 4 | 4 | 7 |
| 5 | | |
| 6 | | |
| 7 | | |
| 8 | 5 | 6 |

# Example DFS



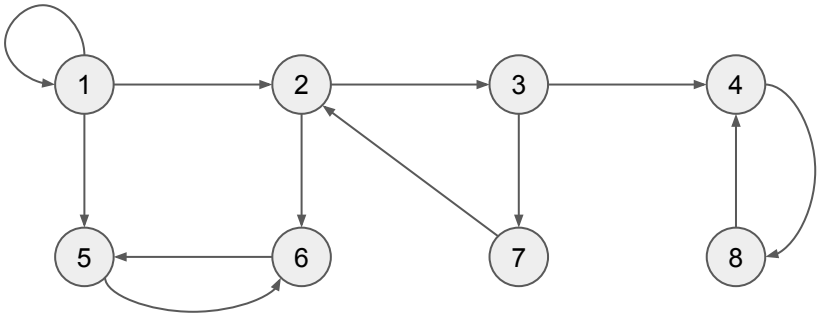

| Node | Discover | Finish |
|---|---|---|
| 1 | 1 | |
| 2 | 2 | |
| 3 | 3 | |
| 4 | 4 | 7 |
| 5 | | |
| 6 | | |
| 7 | 8 | |
| 8 | 5 | 6 |

# Example DFS



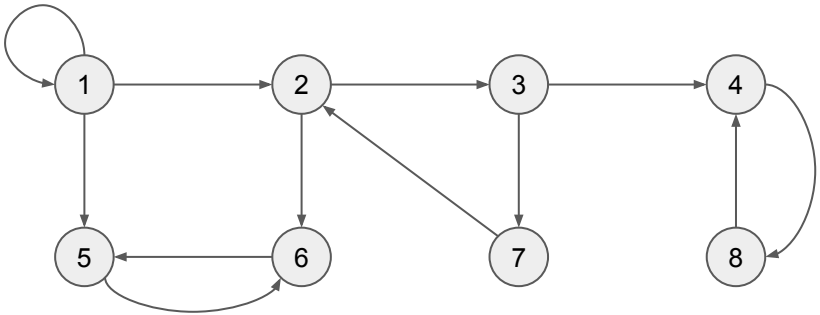

| Node | Discover | Finish |
|---|---|---|
| 1 | 1 | |
| 2 | 2 | |
| 3 | 3 | |
| 4 | 4 | 7 |
| 5 | | |
| 6 | | |
| 7 | 8 | 9 |
| 8 | 5 | 6 |

# Example DFS



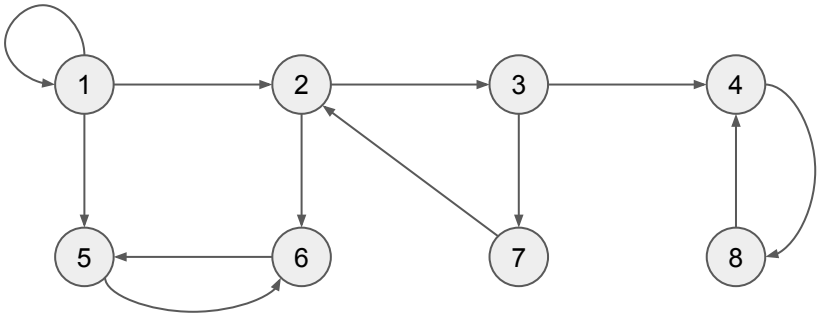

| Node | Discover | Finish |
|---|---|---|
| 1 | 1 | |
| 2 | 2 | |
| 3 | 3 | 10 |
| 4 | 4 | 7 |
| 5 | | |
| 6 | | |
| 7 | 8 | 9 |
| 8 | 5 | 6 |

# Example DFS



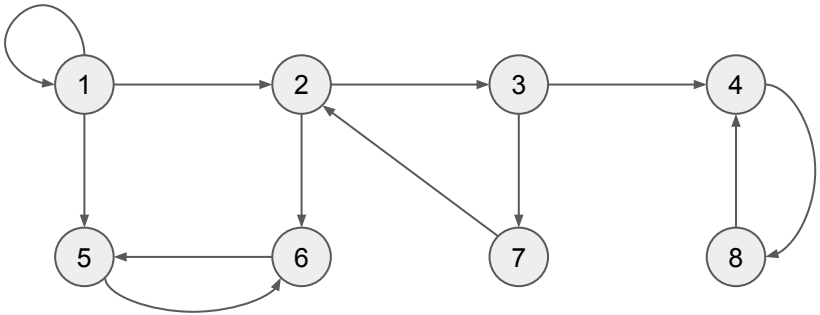

| Node | Discover | Finish |
|---|---|---|
| 1 | 1 | |
| 2 | 2 | |
| 3 | 3 | 10 |
| 4 | 4 | 7 |
| 5 | | |
| 6 | 11 | |
| 7 | 8 | 9 |
| 8 | 5 | 6 |

# Example DFS



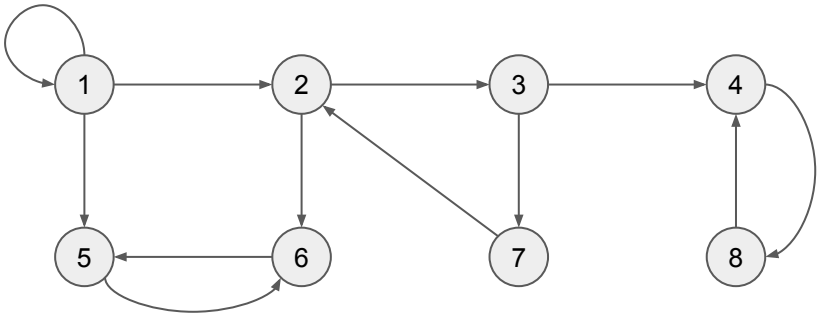

| Node | Discover | Finish |
|---|---|---|
| 1 | 1 | |
| 2 | 2 | |
| 3 | 3 | 10 |
| 4 | 4 | 7 |
| 5 | 12 | |
| 6 | 11 | |
| 7 | 8 | 9 |
| 8 | 5 | 6 |

# Example DFS



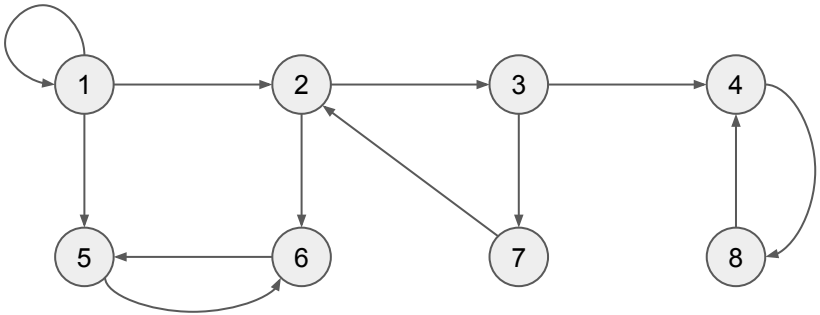

| Node | Discover | Finish |
|---|---|---|
| 1 | 1 | |
| 2 | 2 | |
| 3 | 3 | 10 |
| 4 | 4 | 7 |
| 5 | 12 | 13 |
| 6 | 11 | |
| 7 | 8 | 9 |
| 8 | 5 | 6 |

# Example DFS



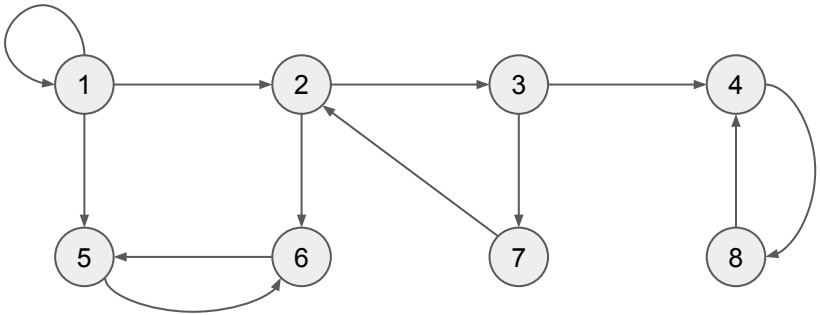

| Node | Discover | Finish |
|---|---|---|
| 1 | 1 | |
| 2 | 2 | |
| 3 | 3 | 10 |
| 4 | 4 | 7 |
| 5 | 12 | 13 |
| 6 | 11 | 14 |
| 7 | 8 | 9 |
| 8 | 5 | 6 |

# Example DFS




| Node | Discover | Finish |
|---|---|---|
| 1 | 1 | |
| 2 | 2 | 15 |
| 3 | 3 | 10 |
| 4 | 4 | 7 |
| 5 | 12 | 13 |
| 6 | 11 | 14 |
| 7 | 8 | 9 |
| 8 | 5 | 6 |

# Example DFS




| Node | Discover | Finish |
|---|---|---|
| 1 | 1 | 16 |
| 2 | 2 | 15 |
| 3 | 3 | 10 |
| 4 | 4 | 7 |
| 5 | 12 | 13 |
| 6 | 11 | 14 |
| 7 | 8 | 9 |
| 8 | 5 | 6 |

# Classification of edges

We can classify the edges in G.E into four kinds:

1. An edge (u, v) ∈ G.E is a **tree edge** if v was discovered by exploring (u, v).
2. For a directed graph, an edge (u, v) ∈ G.E can be a **back edge** if it connects u to some ancestor, v, in the depth-first tree.
3. An edge (u, v) ∈ G.E is a **forward edge** if it is not in the depth-first tree and connects u to a descendant, v, in the tree.
4. All the other edges are **cross edges** and can run between vertices in the same depth-first tree provided one vertex is not an ancestor of the other, or they can run between depth-first trees (only possible in a directed graph).

⚠️ These are not mutually exclusive: edges can have multiple classifications!

# Properties [1]

- Every edge in an undirected graph is either a tree edge or a back edge.

In directed and undirected graphs…

- An edge $(u, v) \in G.E$ is a tree edge or forward edge if and only if $u.discover\_time < v.discover\_time < v.finish\_time < u.finish\_time$
- An edge $(u, v) \in G.E$ is a back edge if and only if $v.discover\_time \leq u.discover\_time < u.finish\_time \leq v.finish\_time$
- An edge $(u, v) \in G.E$ is a cross edge if and only if $v.discover\_time < v.finish\_time < u.discover\_time < u.finish\_time$

# Properties [2]

- Given an undirected graph, DFS will identify the connected components (because it doesn't matter which vertices we explore from when the edges are undirected). The number of times DFS calls DFS-HELPER is the number of connected components.
- If we run DFS on a directed graph then sort the vertices by finish time in descending order, we have a topological sort for the original graph!

# Strongly Connected Components

The Strongly Connected Components problem is defined as:

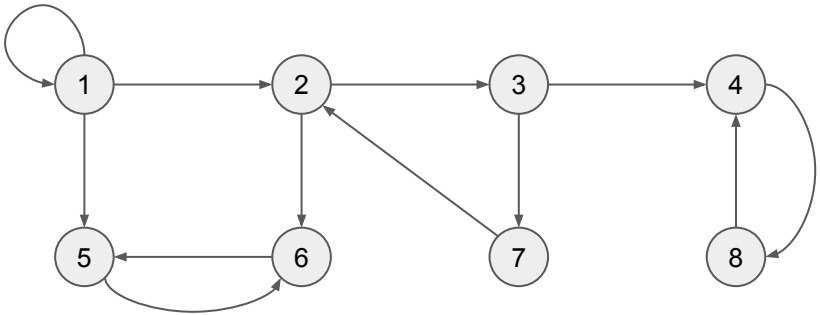
**Input**: a directed graph, G = (V, E)

**Output**: the strongly connected components of G

A **strongly connected component** is a maximal set of vertices $C \subseteq V$ such that for all $u,v \in C$, we have *both* that v is reachable from u *and* that u is reachable from v using edges in E.

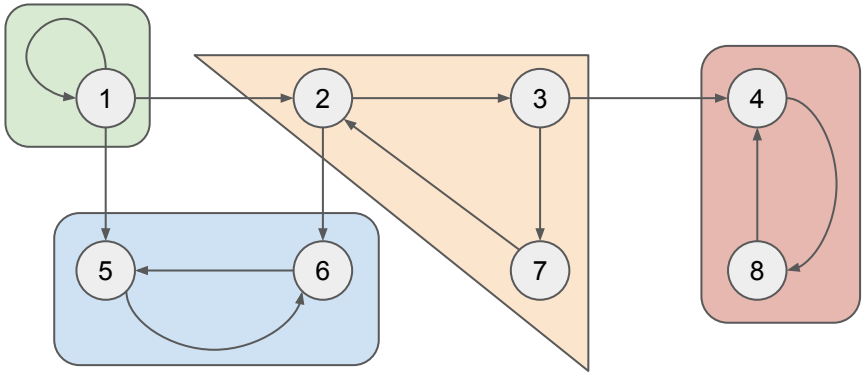
The algorithm uses the transpose graph $G^T = (V, E^T)$.

# Strongly Connected Components Problem Instance




Input graph, G = (V, E), directed

Output: the strongly connected components of G



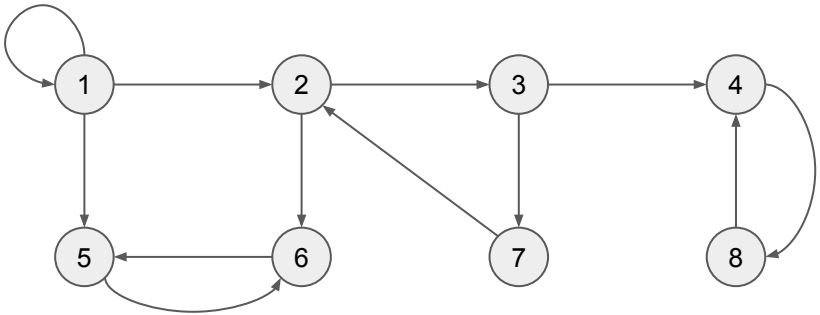
💡 See CLRS chapter 22

# STRONGLY-CONNECTED-COMPONENTS(G)

1. Run DFS on G to populate the finish_time for each vertex v ∈ G.V.
2. Compute $G^T$
3. Run DFS on $G^T$ but in the main loop of DFS, call DFS-HELPER on vertices in order of descending finish_time as computed in step 1.
4. For each tree in the forest produced by DFS($G^T$), output the vertices as a separate strongly connected component of G.

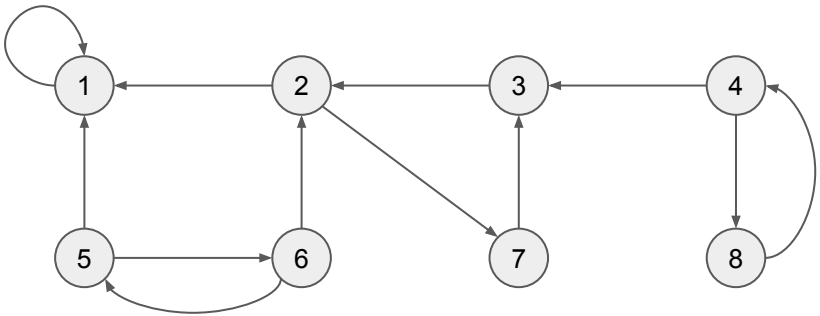
Proof of correctness is in CLRS chapter 22 (pages 617–620 of 3rd edition).

# SCC1: Run DFS on original graph



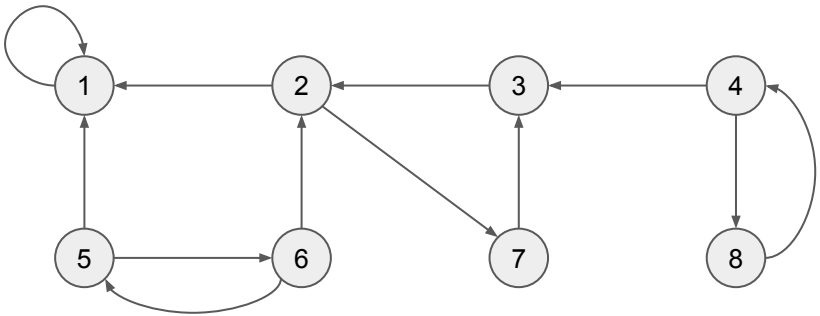

| Node | Discover | Finish |
|---|---|---|
| 1 | 1 | 16 |
| 2 | 2 | 15 |
| 3 | 3 | 10 |
| 4 | 4 | 7 |
| 5 | 12 | 13 |
| 6 | 11 | 14 |
| 7 | 8 | 9 |
| 8 | 5 | 6 |

# SCC2: Compute $G^T$




| Node | Discover | Finish |
|---|---|---|
| 1 | 1 | 16 |
| 2 | 2 | 15 |
| 3 | 3 | 10 |
| 4 | 4 | 7 |
| 5 | 12 | 13 |
| 6 | 11 | 14 |
| 7 | 8 | 9 |
| 8 | 5 | 6 |

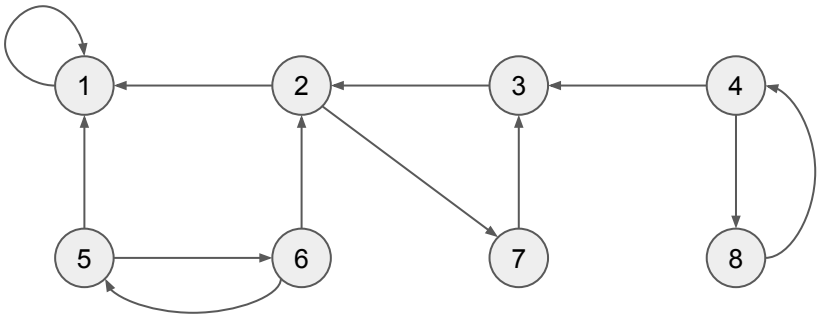
# SCC3a: Reverse sort nodes by finish, DFS in that order




DFS starting from node 1

| Finish1 | Node | Discover | Finish |
|---|---|---|---|
| 16 | 1 | 1 | 2 |
| 15 | 2 | | |
| 14 | 6 | | |
| 13 | 5 | | |
| 10 | 3 | | |
| 9 | 7 | | |
| 7 | 4 | | |
| 6 | 8 | | |

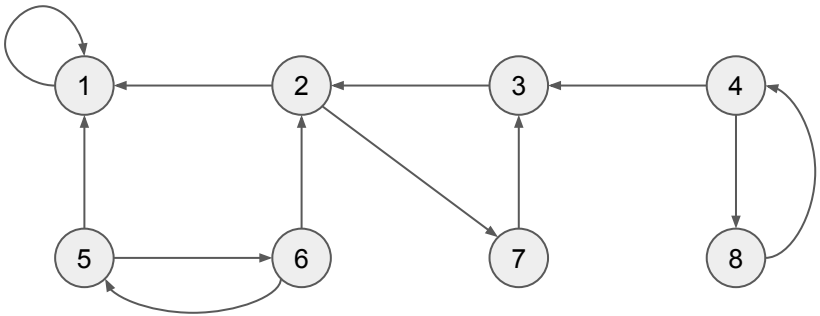
# SCC3b: Continue DFS in that order




| Finish1 | Node | Discover | Finish |
|---|---|---|---|
| 16 | 1 | 1 | 2 |
| 15 | 2 | 3 | 8 |
| 14 | 6 | | |
| 13 | 5 | | |
| 10 | 3 | 5 | 6 |
| 9 | 7 | 4 | 7 |
| 7 | 4 | | |
| 6 | 8 | | |

2 is the first node (in order) that we have not yet visited so DFS starts a new tree of calls to DFS-HELPER starting at node 2.

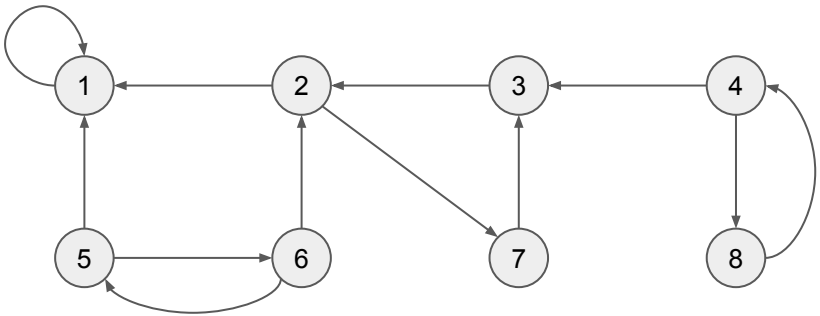
# SCC3c: Continue DFS in that order




| Finish1 | Node | Discover | Finish |
|---|---|---|---|
| 16 | 1 | 1 | 2 |
| 15 | 2 | 3 | 8 |
| 14 | 6 | 9 | 12 |
| 13 | 5 | 10 | 11 |
| 10 | 3 | 5 | 6 |
| 9 | 7 | 4 | 7 |
| 7 | 4 | | |
| 6 | 8 | | |

6 is the first node (in order) that we have not yet visited so DFS starts a new tree of calls to DFS-HELPER starting at node 6.

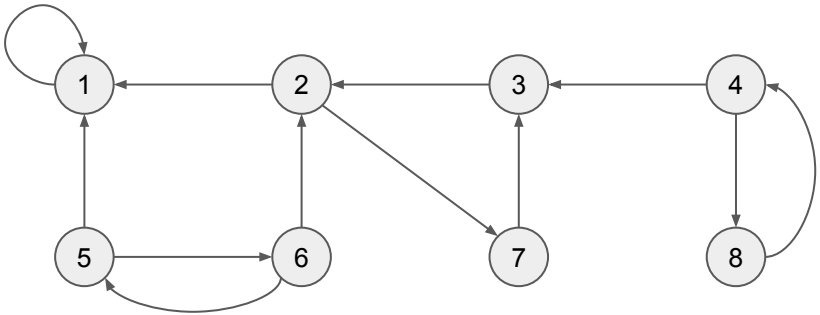
# SCC3d: Continue DFS in that order

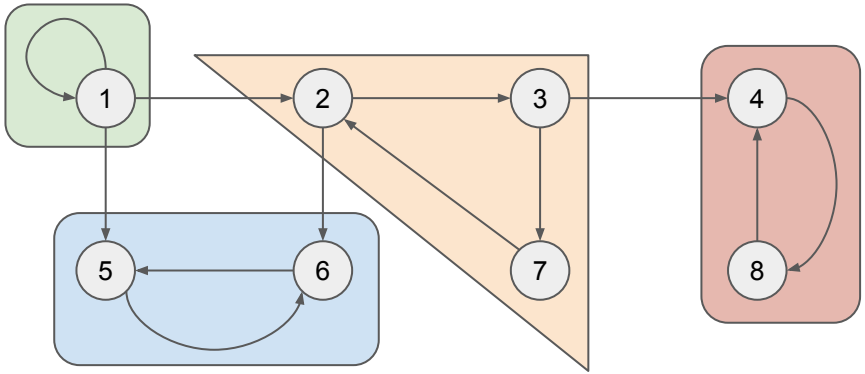



| Finish1 | Node | Discover | Finish |
|---|---|---|---|
| 16 | 1 | 1 | 2 |
| 15 | 2 | 3 | 8 |
| 14 | 6 | 9 | 12 |
| 13 | 5 | 10 | 11 |
| 10 | 3 | 5 | 6 |
| 9 | 7 | 4 | 7 |
| 7 | 4 | 13 | 16 |
| 6 | 8 | 14 | 15 |

4 is the first node (in order) that we have not yet visited so DFS starts a new tree of calls to DFS-HELPER starting at node 4.

# SCC4: Emit vertices of each DF-Tree as a component




| Finish1 | Node | Discover | Finish |
|---|---|---|---|
| 16 | 1 | 1 | 2 |
| 15 | 2 | 3 | 8 |
| 14 | 6 | 9 | 12 |
| 13 | 5 | 10 | 11 |
| 10 | 3 | 5 | 6 |
| 9 | 7 | 4 | 7 |
| 7 | 4 | 13 | 16 |
| 6 | 8 | 14 | 15 |

# Shortest Path Problems [1]

**Input**: a directed, weighted graph, G = (V, E), with its weight function w: E → $\mathbb{R}$.

We define the **weight of a path**, p = $v_0$, $v_1$, $v_2$, … $v_k$, as the linear sum of the edge weights:

$w(p) = \Sigma^{k}_{i=1} w(v_{i-1}, v_i)$

The **edge weights** can represent any *additive* metric: time, cost, distance.

Shortest path problems correspond to additive metrics that we want to minimise.

# Shortest Path Problems [2]

The **shortest path weight** from u to v, $\delta(u, v) = \infty$ if there is no path from u to v, and $\delta(u, v) = \min_p(w(p))$ otherwise, where the minimisation over p considers all paths $u \rightsquigarrow v$.

A **shortest path** from u to v is any such path, p, with $w(p) = \delta(u, v)$.

BFS solved one variant of the shortest path problem: the single-source shortest path problem for unweighted graphs or, equivalently, weighted graphs were all the edge weights have the same (finite, positive) value.

# Shortest Path Problems [3]

What's the output?  Actually, there are several kinds of shortest path problems!

**Single-Source Shortest Paths**: find the shortest paths through a directed, weighted graph from a specified source to all destinations.

**Single-Destination Shortest Paths**: find the shortest paths from every source to a single, specified destination vertex. Same as SSSP in $G^T$.

**Single-Pair Shortest Path**: find the shortest path from u to v (both specified). Best known algorithm has the same worst case cost as best SSSP algorithms.

**All-Pairs Shortest Paths**: find the shortest path between every pair of vertices.

# Complications

It turns out that more efficient algorithms are possible in a subset of cases.

These factors make it harder to solve shortest path problems:

1. Negative-weight edges, and especially negative-weight cycles, make it hard to define what the correct answer is.
2. Cycles. Although it is clear that a path containing a positive-weight cycle can never be a shortest path, and negative weight cycles mean there is no correct answer, what about zero-weight cycles?  If there is a shortest path containing a zero weight cycle, there must be a shortest path without that cycle.

# BELLMAN-FORD(G, w, s)

This finds shortest paths from s $\in$ G.V to every vertex in G.V that is reachable from s – single source shortest paths – in O(|V||E|) time.

If the algorithm finds a negative weight cycle, it return false. This indicates that there is no solution to the single-source shortest paths problem for G.

If there is no negative weight cycle, it returns true. This indicates that the paths found are valid. Paths are acyclic (they exclude zero-weight cycles).

Shortest paths are not returned explicitly but are encoded as $\pi$ attributes. This takes less time to produce and no additional time to consume.

## BELLMAN-FORD(G, w, s)

```
1   for v in G.V
2       v.d = ∞
3       v.π = NIL
4   s.d = 0
5   for i = 1 to |G.V|-1
6       for (u,v) in G.E
7   for (u,v) in G.E
8   return true
```

## RELAX(u, v, w)

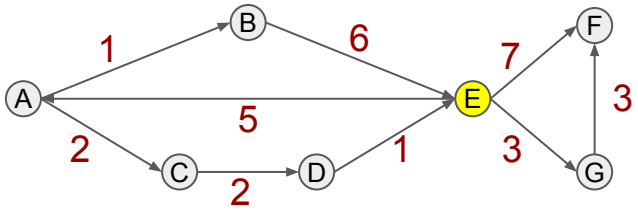
```
1   if v.d > u.d + w(u, v)
2       v.d = u.d + w(u, v)
3       v.π = u
```

```
RELAX(u, v, w)
```

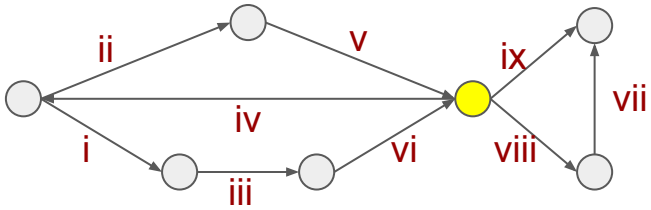
```
if v.d > u.d + w(u, v)    return false
```

Initialisation Θ(|V|). Line 5 runs Θ(|V|) times and line 6 takes Θ(|E|). Final check Θ(|E|). Overall O(|V||E|).

# Example: BELLMAN-FORD(G, w, “E”)




Weighted, directed graph (input)

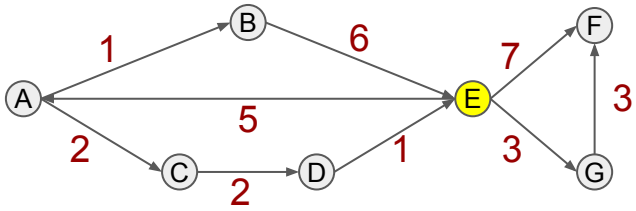



Order of edges in E

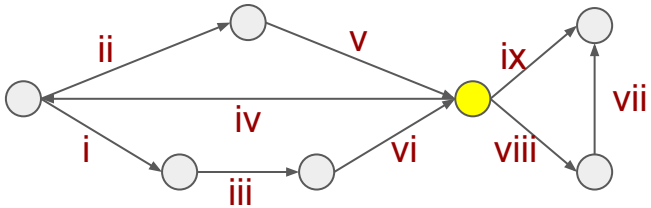
Initialisation

| Vertex | d | π |
|---|---|---|
| A | ∞ | NIL |
| B | ∞ | NIL |
| C | ∞ | NIL |
| D | ∞ | NIL |
| E | 0 | NIL |
| F | ∞ | NIL |
| G | ∞ | NIL |

# Example: BELLMAN-FORD(G, w, “E”)




Weighted, directed graph (input)



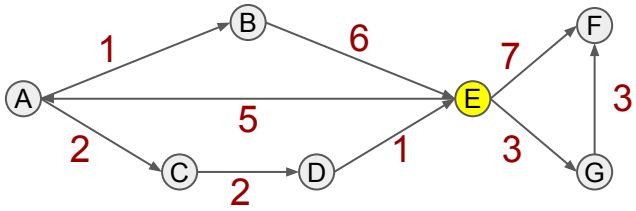

Order of edges in E

Iteration i=1

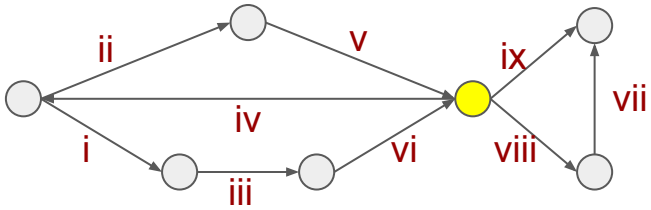
| Vertex | d | π |
|---|---|---|
| A | 5 | E |
| B | ∞ | NIL |
| C | ∞ | NIL |
| D | ∞ | NIL |
| E | 0 | NIL |
| F | 7 | E |
| G | 3 | E |

💡 Order of edges means we relax E→G too late to get F.d=6 in this iteration.

# Example: BELLMAN-FORD(G, w, “E”)




Weighted, directed graph (input)



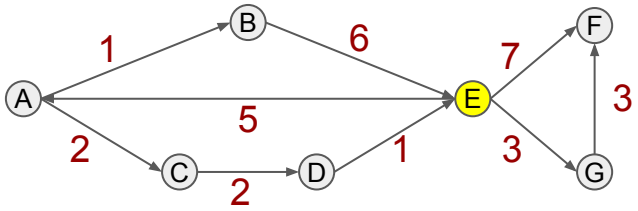

Order of edges in E

Iteration i=2

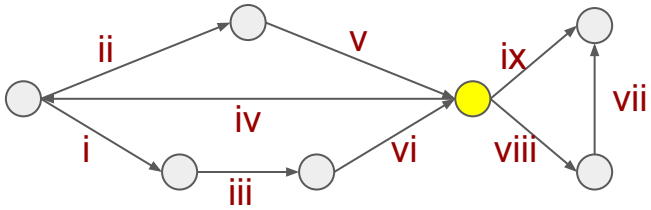
| Vertex | d | π |
|---|---|---|
| A | 5 | E |
| B | 6 | A |
| C | 7 | A |
| D | 9 | C |
| E | 0 | NIL |
| F | 6 | G |
| G | 3 | E |

💡 C and D changed in the same iteration, due to the order of edges. This only speeds up convergence.

# Example: BELLMAN-FORD(G, w, “E”)




Weighted, directed graph (input)

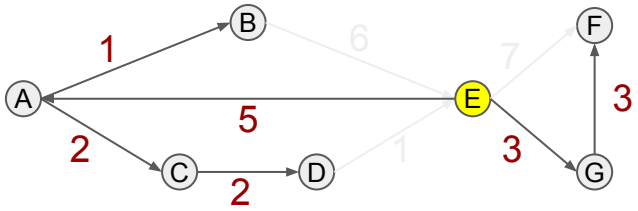



Order of edges in E

Iteration i=3,4,5,6,7 (no changes)

| Vertex | d | π |
|---|---|---|
| A | 5 | E |
| B | 6 | A |
| C | 7 | A |
| D | 9 | C |
| E | 0 | NIL |
| F | 6 | G |
| G | 3 | E |

# Example: BELLMAN-FORD(G, w, “E”)

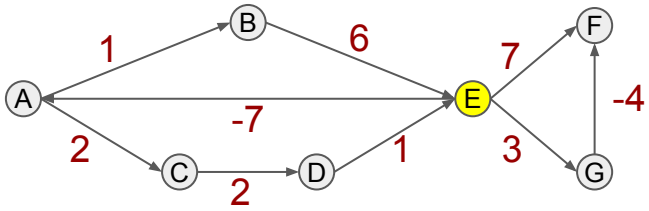



Shortest paths (output)

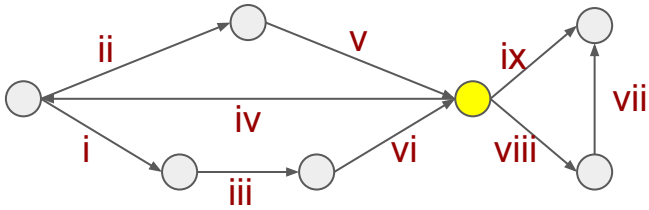
| Vertex | d | π |
|---|---|---|
| A | 5 | E |
| B | 6 | A |
| C | 7 | A |
| D | 9 | C |
| E | 0 | NIL |
| F | 6 | G |
| G | 3 | E |

Final check: any v.d>u.d+w(u,v)?
No ⇒ return true
Distances and shortest paths are valid

# Negative Cycle Example: BELLMAN-FORD(G, w, “E”)




Weighted, directed graph (input)

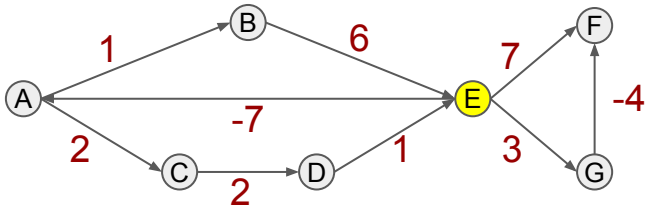



Order of edges in E

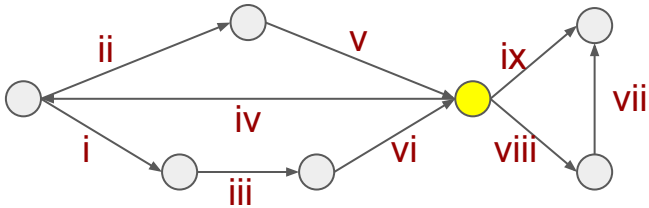
Initialisation

| Vertex | d | π |
|---|---|---|
| A | ∞ | NIL |
| B | ∞ | NIL |
| C | ∞ | NIL |
| D | ∞ | NIL |
| E | 0 | NIL |
| F | ∞ | NIL |
| G | ∞ | NIL |

# Negative Cycle Example: BELLMAN-FORD(G, w, "E")




Weighted, directed graph (input)

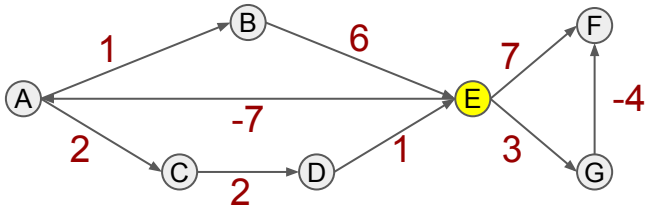



Order of edges in E

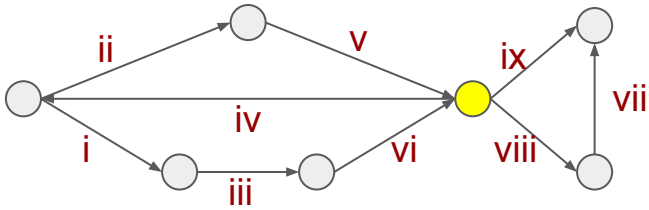
Iteration i=1

| Vertex | d | π |
|---|---|---|
| A | -7 | E |
| B | ∞ | NIL |
| C | ∞ | NIL |
| D | ∞ | NIL |
| E | 0 | NIL |
| F | 7 | E |
| G | 3 | E |

# Negative Cycle Example: BELLMAN-FORD(G, w, “E”)




Weighted, directed graph (input)

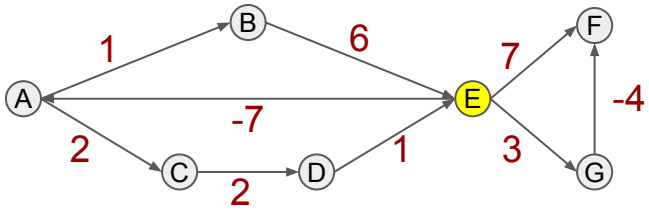



Order of edges in E

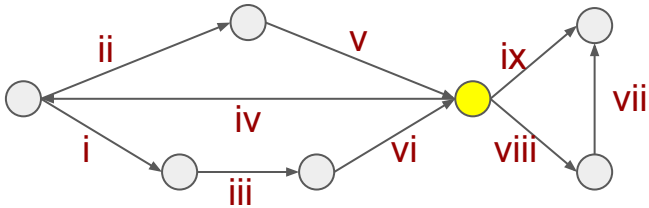
Iteration i=2

| Vertex | d | π |
|---|---|---|
| A | -7 | E |
| B | -6 | A |
| C | -5 | A |
| D | -3 | C |
| E | -2 | D |
| F | -1 | G |
| G | 1 | E |

# Negative Cycle Example: BELLMAN-FORD(G, w, "E")




Weighted, directed graph (input)

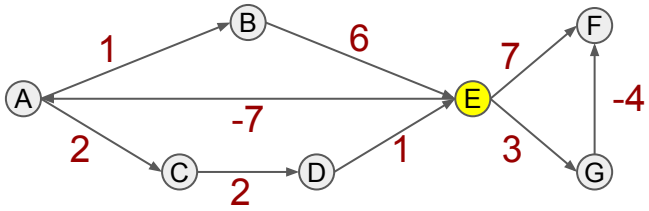



Order of edges in E

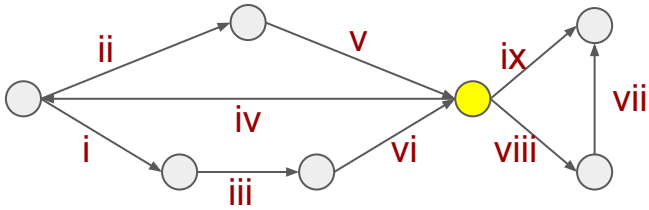
Iteration i=3

| Vertex | d | π |
|---|---|---|
| A | -9 | E |
| B | -6 | A |
| C | -5 | A |
| D | -3 | C |
| E | -2 | D |
| F | -3 | G |
| G | 1 | E |

# Negative Cycle Example: BELLMAN-FORD(G, w, “E”)




Weighted, directed graph (input)

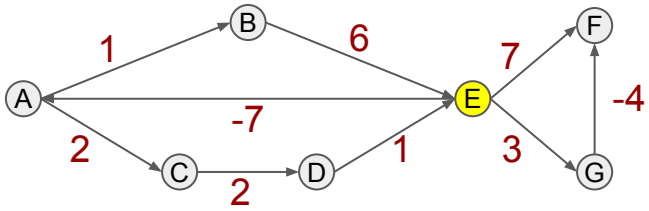



Order of edges in E

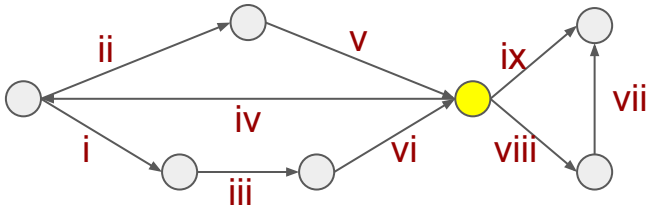
Iteration i=4

| Vertex | d | π |
|---|---|---|
| A | -9 | E |
| B | -8 | A |
| C | -7 | A |
| D | -5 | C |
| E | -4 | D |
| F | -3 | G |
| G | -1 | E |

# Negative Cycle Example: BELLMAN-FORD(G, w, “E”)




Weighted, directed graph (input)

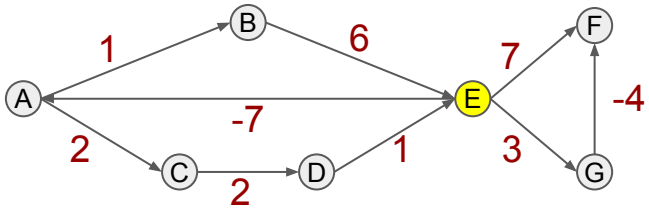



Order of edges in E

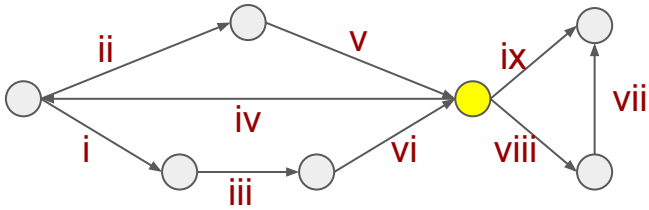
Iteration i=5

| Vertex | d | π |
|---|---|---|
| A | -11 | E |
| B | -8 | A |
| C | -7 | A |
| D | -5 | C |
| E | -4 | D |
| F | -5 | G |
| G | -1 | E |

# Negative Cycle Example: BELLMAN-FORD(G, w, "E")




Weighted, directed graph (input)

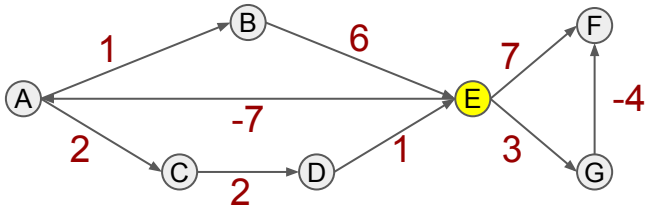



Order of edges in E

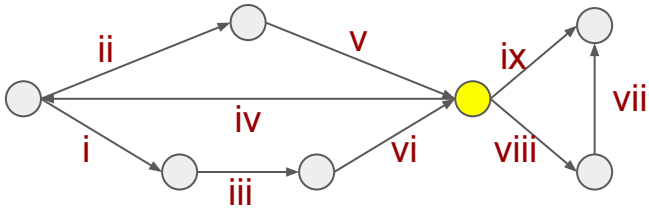
Iteration i=6

| Vertex | d | π |
|---|---|---|
| A | -11 | E |
| B | -10 | A |
| C | -9 | A |
| D | -7 | C |
| E | -6 | D |
| F | -5 | G |
| G | -3 | E |

# Negative Cycle Example: BELLMAN-FORD(G, w, "E")




Weighted, directed graph (input)

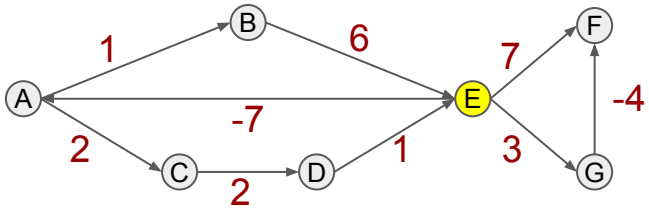



Order of edges in E

Iteration i=7

| Vertex | d | π |
|---|---|---|
| A | -13 | E |
| B | -10 | A |
| C | -9 | A |
| D | -7 | C |
| E | -6 | D |
| F | -7 | G |
| G | -3 | E |

# Negative Cycle Example: BELLMAN-FORD(G, w, "E")




Weighted, directed graph (input)

Final check: any v.d>u.d+w(u,v)?
Yes, (A,C): -9>-13+2 ⇒ return false
Distances are invalid. No shortest paths.

Termination

| Vertex | d | π |
|---|---|---|
| A | -13 | E |
| B | -10 | A |
| C | -9 | A |
| D | -7 | C |
| E | -6 | D |
| F | -7 | G |
| G | -3 | E |

# Special cases for DAGs

Many important problems give rise to directed graphs that are naturally acyclic.

It turns out that we can solve this special case of the single-source shortest paths problem with lower asymptotic time complexity than the general case: Θ(|V| + |E|).

```
1   for v in G.V
2       v.d = ∞
3       v.π = NIL
4   s.d = 0
```

```
5   TOPOLOGICAL-SORT(G)
6   for u in G.V (sorted order)
7       for v in G.E.adj[u]
8           RELAX(u, v, w)
```

Initialisation (lines 1–4) is Θ(|V|). Topological sort is Θ(|V| + |E|). Lines 6–8 are Θ(|E|). Total Θ(|V| + |E|).

# Optimal Substructure

- If $p = u \rightsquigarrow v = u, .. v_i, .. v_j, .. v$ is a shortest path from u to v through the weighted edges of some graph G,
- …and it goes via $v_i$ and $v_j$ in that order (although not necessarily adjacently),
- …then the subpath from $v_i \rightsquigarrow v_j$ is a shortest path from $v_i$ to $v_j$.

Proof: if $v_i \rightsquigarrow v_j$ isn't a shortest path, replacing it in p with any shortest path $v_i \rightsquigarrow v_j$ yields a shorter path $u \rightsquigarrow v$ and contradicts that p was shortest.

This means we can look to dynamic programming methods and greedy algorithms to provide efficient solutions to problems involving shortest paths! Let's see how to exploit this in other algorithms.

# DIJKSTRA(G, w, s)

Dijkstra's algorithm solves the single-source shortest paths problem using a greedy strategy to exploit the optimal substructure of shortest paths.

Dijkstra's algorithm works on directed graphs with non-negative edge weights, i.e. $w(u,v) \geq 0$ for all $(u, v) \in G.E$.

The greedy algorithm achieves a lower cost than Bellman-Ford (albeit that Bellman-Ford can handle negative edges and detects negative cycles).

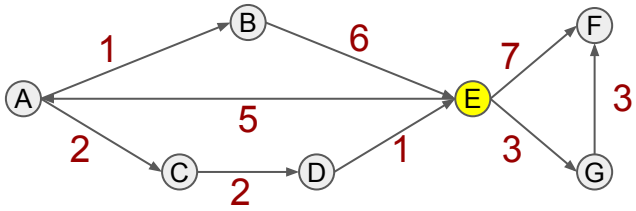
# DIJKSTRA(G, w, s)

```
1   for v in G.V
2       v.d = ∞
3       v.π = NIL
4   s.d = 0
5   S = EMPTY-SET
6   Q = new PriorityQueue(G.V)
7   while !PQ-EMPTY(Q)
8       u = PQ-EXTRACT-MIN(Q)
9       S = S ∪ {u}
10      for v in G.E.adj[u]
11          RELAX(u, v, w)
```

💡 The priority queue uses the ‘d’ attribute as the ordering key. Changing ‘d’ (in RELAX) implicitly calls DECREASE-KEY.

💡 Note that the set, S, of nodes whose shortest paths have been found, is not used. We could delete lines 5 and 9 without consequence. S is included in most presentations of Dijkstra’s Algorithm because Dijkstra’s original description used it, and we will use it for the proof of correctness.

# Example: DIJKSTRA(G, w, “E”)



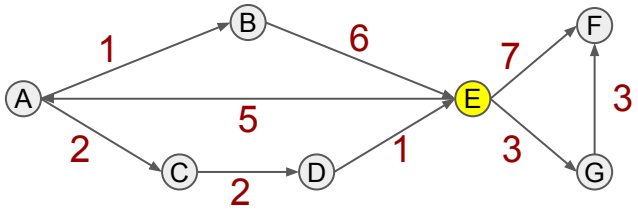

Weighted, directed graph (input)

PQ: → (G,∞) (F,∞) (D,∞) (C,∞) (B,∞) (A,∞) (E,0) →

Initialisation

| Vertex | d | π |
|---|---|---|
| A | ∞ | NIL |
| B | ∞ | NIL |
| C | ∞ | NIL |
| D | ∞ | NIL |
| E | 0 | NIL |
| F | ∞ | NIL |
| G | ∞ | NIL |

# Example: DIJKSTRA(G, w, “E”)



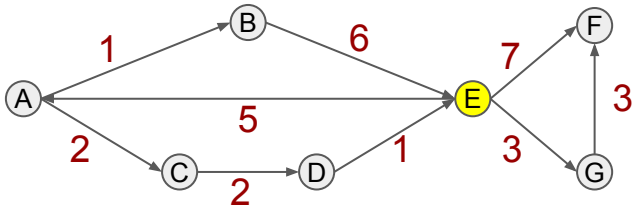

Weighted, directed graph (input)

PQ: → (D,∞) (C,∞) (B,∞) (F,7) (A,5) (G,3) →

Iteration 1

| Vertex | d | π |
|---|---|---|
| A | 5 | E |
| B | ∞ | NIL |
| C | ∞ | NIL |
| D | ∞ | NIL |
| E | 0 | NIL |
| F | 7 | E |
| G | 3 | E |

# Example: DIJKSTRA(G, w, “E”)



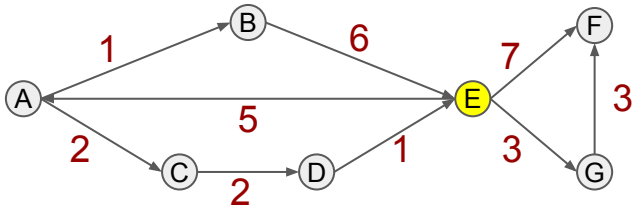

Weighted, directed graph (input)

PQ: → (D,∞) (C,∞) (B,∞) (F,6) (A,5) →

Iteration 2

| Vertex | d | π |
|---|---|---|
| A | 5 | E |
| B | ∞ | NIL |
| C | ∞ | NIL |
| D | ∞ | NIL |
| E | 0 | NIL |
| F | 6 | G |
| G | 3 | E |

# Example: DIJKSTRA(G, w, “E”)



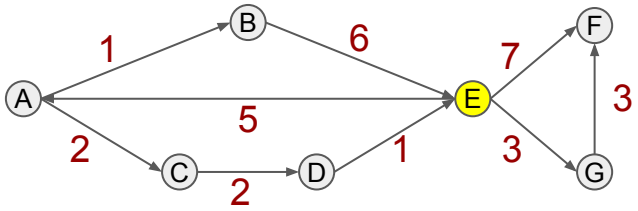

Weighted, directed graph (input)

PQ: → (D,∞) (C,7) (B,6) (F,6) →

Iteration 3

| Vertex | d | π |
|---|---|---|
| A | 5 | E |
| B | 6 | A |
| C | 7 | A |
| D | ∞ | NIL |
| E | 0 | NIL |
| F | 6 | G |
| G | 3 | E |

# Example: DIJKSTRA(G, w, “E”)



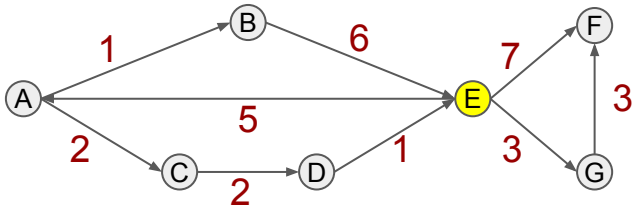

Weighted, directed graph (input)

PQ: → (D,∞) (C,7) (B,6) →

Iteration 4

| Vertex | d | π |
|---|---|---|
| A | 5 | E |
| B | 6 | A |
| C | 7 | A |
| D | ∞ | NIL |
| E | 0 | NIL |
| F | 6 | G |
| G | 3 | E |

# Example: DIJKSTRA(G, w, “E”)



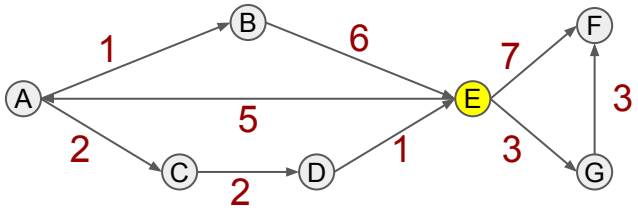

Weighted, directed graph (input)

PQ: → (D,∞) (C,7) →

Iteration 5

| Vertex | d | π |
|---|---|---|
| A | 5 | E |
| B | 6 | A |
| C | 7 | A |
| D | ∞ | NIL |
| E | 0 | NIL |
| F | 6 | G |
| G | 3 | E |

# Example: DIJKSTRA(G, w, “E”)



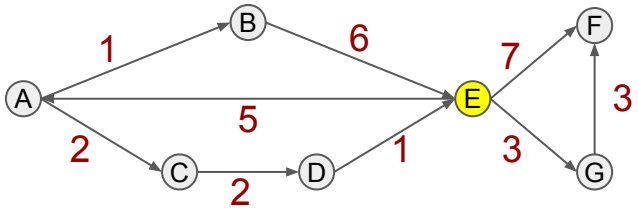

Weighted, directed graph (input)

PQ: → (D,9) →

Iteration 6

| Vertex | d | π |
|---|---|---|
| A | 5 | E |
| B | 6 | A |
| C | 7 | A |
| D | 9 | C |
| E | 0 | NIL |
| F | 6 | G |
| G | 3 | E |

# Example: DIJKSTRA(G, w, “E”)



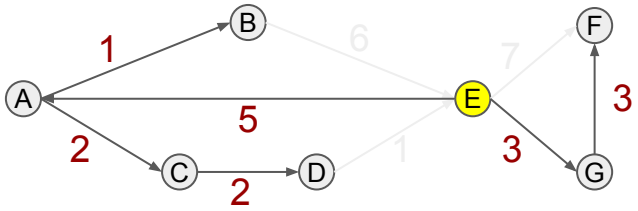

Weighted, directed graph (input)

PQ: → →

Iteration 7

| Vertex | d | π |
|---|---|---|
| A | 5 | E |
| B | 6 | A |
| C | 7 | A |
| D | 9 | C |
| E | 0 | NIL |
| F | 6 | G |
| G | 3 | E |

# Example: DIJKSTRA(G, w, “E”)




PQ: → →

Shortest paths (output)

Termination

| Vertex | d | π |
|---|---|---|
| A | 5 | E |
| B | 6 | A |
| C | 7 | A |
| D | 9 | C |
| E | 0 | NIL |
| F | 6 | G |
| G | 3 | E |

# Correctness of DIJKSTRA(G, w, s) [1]

We want to show that when DIJKSTRA runs on a directed graph, G = (V, E), with non-negative edge weights and source s, it terminates with v.d = $\delta$(s, v) for all v $\in$ G.V.

The proof is by induction on the cardinality of set, S.

We show that the following property is true at the start of each iteration of the WHILE loop (lines 7–11):

$\phi$: v.d = $\delta$(s, v) for all v $\in$ S

We proceed as usual with Initialisation, Maintenance and Termination…

# Correctness of DIJKSTRA(G, w, s) [2]

**Initialisation**: at the start of the first iteration, S = ∅ so ϕ is vacuously true.

**Maintenance**: proof by contradiction. Let u be the first vertex that, when it is added to S, has u.d ≠ $\delta$(s, u). Consider the iteration of the WHILE loop that added u to S.

We know that u ≠ s since s.d = 0 = $\delta$(s, s), and hence S ≠ ∅ when u was added. There must be some path s ⇝ u to be found, since otherwise u.d = ∞ = $\delta$(s, u), and hence some shortest path to be found.

Let's consider such a shortest path s ⇝ u…

# Correctness of DIJKSTRA(G, w, s) [3]

Before we add u to S, the shortest path $p = s \rightsquigarrow u$ can be split $p = s \rightsquigarrow y \rightsquigarrow u$, where $y \notin S$ is the first vertex in p not be in S. Let $x \in S$ be the predecessor to y in path p then we can write $p = s \rightsquigarrow_{p1} x \rightarrow y \rightsquigarrow_{p2} u$. (Either/Both p1 and p2 might be empty.)

We know that $x.d = \delta(s, x)$ when x was added to S because u is the first vertex for which this failed. The edge (x, y) was relaxed in the iteration that added x to S so we know that $y.d = \delta(s, y)$ – this is known as the convergence property.

**Convergence property of RELAX(i, j)**:
If $s \rightsquigarrow i \rightarrow j$ is a shortest path in G and $i.d = \delta(s, i)$ before edge (i, j) is relaxed, then $j.d = \delta(s, j)$ afterwards.

# Correctness of DIJKSTRA(G, w, s) [4]

**Proof of Convergence property of RELAX(i, j)**:

If $s \rightsquigarrow i \rightarrow j$ is a shortest path in G and $i.d = \delta(s, i)$ before edge (i, j) is relaxed, then $j.d = \delta(s, j)$ afterwards.

After relaxing edge (i, j) we know that

$$\begin{aligned} j.d &\leq i.d + w(i, j) \\ &= \delta(s, i) + w(i, j) \\ &= \delta(s, j) \end{aligned}$$

And since we know that j.d never underestimates $\delta(s, j)$, we have $j.d = \delta(s, j)$.

# Correctness of DIJKSTRA(G, w, s) [5]

Back to Dijkstra. Since the weights are non-negative and y is before u in our shortest path, p, we know that $\delta(s, y) \leq \delta(s, u)$ and hence…

$$
\begin{aligned}
y.d &= \delta(s, y) \\
&\leq \delta(s, u) \\
&\leq u.d \qquad \text{(since we assume u.d is incorrect and it cannot be less)}
\end{aligned}
$$

Both $u \notin S$ and $y \notin S$ when u was taken from the priority queue so $u.d \leq y.d$.

Combining these, we have $y.d = \delta(s, y) = \delta(s, u) = u.d$. Contradicts assumption!

Hence $u.d = \delta(s, u)$ when u was added to S so $\phi$ is maintained by the loop.

# Correctness of DIJKSTRA(G, w, s) [6]

**Termination**: when we terminate, the priority queue, Q, is empty. Since Q = V \ S we must have processed all vertices when DIJKSTRA terminates.

Therefore the maintained property applies to every vertex and we have that v.d = $\delta$(s, v) for all v $\in$ V, i.e. $\phi$ is true and we have proved the correctness of Dijkstra's algorithm.

It follows that the predecessor subgraph $G_\pi$, is a shortest path tree rooted at s, i.e. not only are the distances correct but the paths obtained by following the $\pi$ attributes are also correct.

# Analysis of DIJKSTRA(G, w, s)

The initialisation takes Θ(|V|) time. Initialising a priority queue takes O(1) to O(|V|) depending on the type of priority queue (and memory allocator) used.

- Every vertex is PQ-INSERTed once.
- PQ-EXTRACT-MIN'ed once.
- We check PQ-EMPTY |V|+1 times.
- RELAX triggers PQ-DECREASE-KEY once per edge in the worst case.

The final cost depends on the type of priority queue we use.

# Analysis of DIJKSTRA(G, w, s) with an array / hash table

We can implement the priority queue using an array (or hash table) holding (d, $\pi$) for each vertex v $\in$ [1, 2, .. |V|]. PQ-INSERT takes O(1) time per vertex. PQ-EXTRACT-MIN take O(|V|) time to search the array for the smallest 'd'. PQ-EMPTY is O(1) because we can keep a counter. PQ-DECREASE-KEY is O(1), because we must only change 'd' in one array position.

The final cost is O(|V|1 + |V||V| + |V|1 + |E|1) = O($|V|^2$ + |E|).
(initialisation + extractions + empty checks + decrease keys)

# Analysis of DIJKSTRA(G, w, s) with a min-heap

For a min-heap keyed by ‘d’, PQ-INSERT takes O(lg |V|) time per vertex, or, smarter, we can insert all vertices then run FULL-HEAPIFY to build a heap in O(|V|) time.

PQ-EXTRACT-MIN take O(lg |V|) time. PQ-EMPTY is O(1) because we can keep a counter. PQ-DECREASE-KEY is O(lg |V|), to REHEAPIFY that node in the heap.

The final cost is O(|V| + |V| lg |V| + |V|1 + |E| lg |V|) = O((|V| + |E|) lg |V|).
(initialisation + extractions + empty checks + decrease keys)

This is O(|E| lg |V|) if every vertex is reachable from s.

💡 We will do better later in the course!

# All-Pairs Shortest Paths

**Input**: a weighted, directed graph G = (V, E)

**Output**: a |V|x|V| matrix, D = ($d_{ij}$), where $d_{ij} = \delta(i, j)$ is the shortest path weight from i to j (∞ if j is unreachable from i).

One solution is obvious: run a single-source shortest path algorithm with each vertex v ∈ G.V in turn as the source.

# All-Pairs Shortest Paths via BELLMAN-FORD

One solution is obvious: run a single-source shortest path algorithm with each vertex $v \in G.V$ in turn as the source.

BELLMAN-FORD(G, w, v) on a single source vertex has running time $O(|V||E|)$ so repeating that for each vertex takes $O(|V|^2|E|)$ time.

If the graph is dense, this is $O(|V|^4)$.

# All-Pairs Shortest Paths via DIJKSTRA

If the edge weights are non-negative, $w(i, j) \geq 0$ for all $i,j \in G.V$, we can use Dijkstra's algorithm.

Using a heap for the priority queue, each source costs $O((|V| + |E|) \lg |V|)$ and overall we have $O((|V| + |E|) |V| \lg |V|)$ running time.

Using an asymptotically optimal priority queue (as we shall see later in the course), we can achieve $O(|V|^2 \lg |V| + |V||E|)$ overall running time.

However, it is possible to do better.

# Matrix Methods

We use the adjacency matrix representation.

If G.E.M is the square matrix of edge weights, consider the matrix G.E.M x G.E.M (i.e. the matrix multiplied by itself).

- If we reinterpret the scalar + and scalar * operations that are used in matrix multiplication as MIN and + respectively then…
- Element (i,j) in the resulting matrix is $MIN_k\{i \rightarrow k + k \rightarrow j\}$, over all $k \in G.V$ (because regular multiplication would set (i,j) to $*_k\{(i,k)+(k,j)\}$ over all k).

This adds one ‘hop’ to the end of all paths represented in the left matrix.

# Repeated squaring

Because there can be no shortest paths longer than |V| - 1, the matrix $(G.E.M)^x$ is a matrix of all shortest paths provided $x \geq |V| - 1$.

By analogy with ordinary matrix multiplication, we can use repeated squaring to find this matrix with running time in $O(|V|^3 \ln |V|)$.

A useful supervision exercise is to flesh out the details.
For now, we want to think about these matrices differently.

# Dynamic Programming on Graphs: Floyd-Warshall [1]

We can use dynamic programming to solve the all-pairs shortest path problem.

We use the adjacency matrix representation.

If a (simple) path, $p = v_1, v_2, .. v_x$ then we define an **intermediate vertex** as any of $v_2..v_{x-1}$. The Floyd-Warshall algorithm notes an optimal substructure property.

For any $i,j \in G.V$, consider a minimum weight path $p = i \rightsquigarrow j$ that *only* has intermediate vertices in a subset $\{1, 2, .. k\} \subseteq G.V$.

# Dynamic Programming on Graphs: Floyd-Warshall [2]

For any i,j $\in$ G.V, consider a minimum weight path p = i $\rightsquigarrow$ j that *only* has intermediate vertices in a subset {1, 2, .. k} $\subseteq$ G.V.

Either p has k as an intermediate vertex, or it does not.

- If k is not an intermediate vertex in p then a minimum weight path using intermediate vertices {1, 2, .. k} is also a minimum weight path using intermediate vertices {1, 2, .. k-1}.
- If k is an intermediate vertex in p then we can decompose as p = i $\rightsquigarrow_{p1}$ $v_k$ $\rightsquigarrow_{p2}$ j where p1 and p2 are subpaths that only use {1, 2, .. k-1} as intermediates. (p1 and p2 do not go via $v_k$ as p would be cyclic and hence not shortest.)

# Dynamic Programming on Graphs: Floyd-Warshall [3]

This observation gives us a dynamic programming approach! Working bottom-up, the minimum weight paths i $\rightsquigarrow$ j using no intermediates are the edge weights.

For k = 1 to |G.V|

    For each i,j $\in$ G.V

        Lookup the min weight path i $\rightsquigarrow$ j only using vertices {1, 2, .. k-1}  [x]

        Lookup the min weight paths i $\rightsquigarrow$ k and k $\rightsquigarrow$ j using only {1, 2, .. k-1}[y,z]

        Set the min weight path i $\rightsquigarrow$ j using {1, 2, .. k} as MIN(x, y+z)

The two “Lookup” steps refer to smaller instances of the same problem that have already been solved. The “Set…” step saves a value that will be looked up later.

# FLOYD-WARSHALL(G, w)

```
1   D^(0) = w
2   for k = 1 to |G.V|
3       let D^(k) = (d_ij^(k)) be a new matrix
4       for i = 1 to |G.V|
5           for j = 1 to |G.V|
6               d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
7   return D^(|G.V|)
```

💡 Floyd-Warshall finds the matrix of all-pairs shortest path lengths in $O(|V|^3)$ running time.

# Extensions to Floyd-Warshall

1. In parallel with $D^{(k)}$, keep a matrix $\Pi^{(k)} = (\pi_{ij}^{(k)})$ where $\pi_{ij}^{(k)}$ is the predecessor of j in a minimum weight path from i using intermediates in {1, 2, .. k}.
   a. Initialise $\pi_{ij}^{(0)}$ to NIL if i = j or (i,j) $\notin$ G.E; and to i otherwise.
   b. Set $\pi_{ij}^{(k)}$ to $\pi_{ij}^{(k-1)}$ or $\pi_{kj}^{(k-1)}$ corresponding to which of the two options was selected by MIN on line 6.
2. To compute the transitive closure of G.E, $G.E^*$, run Floyd-Warshall with w(i, j) = 1 for all (i, j) $\in$ G.E. Interpret the output matrix, $D = (d_{ij})$, as follows:
   a. If $d_{ij} < \infty$ then (i, j) $\in$ $G.E^*$
   b. Otherwise, (i, j) $\notin$ $G.E^*$

To compute $G.E^*$, we can also interpret G.E as Booleans (edge = true) then run Floyd-Warshall with MIN interpreted as Boolean OR and + as AND.

# Johnson's Algorithm

Johnson's algorithm solves the all-pairs shortest paths problem with expected running time $O(|V|^2 \lg |V| + |V||E|)$.

Johnson's algorithm can handle negative edge weights, and will detect negative cycles and report that no solution exists.

Provided G is sparse (more precisely, if $|E| \in o(|V|^2)$, Johnson's algorithm is asymptotically cheaper than Floyd-Warshall. Johnson is also faster than repeated squaring.

Johnson's algorithm is based on a clever trick known as **reweighting**.

# Reweighting [1]

In order to run Dijkstra's algorithm with every vertex as the source, we need to ensure there are no negative edge weights.

Specifically, we require a new set of edge weights, $w(u, v)$, such that…

1. For all edges, $(u, v) \in G.E$, $w(u, v)$ is non-negative; and
2. For all pairs of vertices $u,v \in G.V$, if p is a shortest path (sum of edge weights) under the original weight function, w, then p is also a shortest path under $w$.

We cannot add a bias, b, to every edge weight such that $b + w(u, v) \geq 0$ for all $(u, v) \in G.E$ because paths are different lengths: longer paths would be penalised.

# Reweighting [2]

Define $w(u, v) = w(u, v) + h(u) - h(v)$
where $h : V \rightarrow \mathbb{R}$ is a function mapping *vertices* to real numbers.

1. For all edges, $(u, v) \in G.E$, $w(u, v)$ is non-negative; and
2. For all pairs of vertices $u,v \in G.V$, if p is a shortest path (sum of edge weights) under the original weight function, w, then p is also a shortest path under $w$.

We cannot add a bias, b, to every edge weight such that $b + w(u, v) \geq 0$ for all $(u, v) \in G.E$ because paths are different lengths: longer paths would be penalised.

# Reweighting [3]

It is easy to show that there is a negative cycle under $w$ if there is a negative cycle under w: consider a cyclic path p = $v_1$, $v_2$, .. $v_1$. The sum of edge weights under $w$ is, $w(p) = w(1, 2) + w(2, 3) + \ldots + w(n, 1)$

$= w(1, 2) + h(1) - h(2) + w(2, 3) + h(2) - h(3) + \ldots + w(n, 1) + h(n) - h(1)$

$= w(1, 2) + w(2, 3) + \ldots + w(n, 1)$

$= w(p)$

If p = $v_1$, $v_2$, .. $v_n$, is a shortest path under w then it also is under $w$ because $w(p) = w(p) + h(v_1) - h(v_n)$ but $h(v_1)$ and $h(v_n)$ do not depend on the path. If some path $v_1 \rightsquigarrow v_n$ minimises w(p), it must also minimise $w(p)$.

# Reweighting [4]

From our input graph G = (V, E), construct an augmented graph, G' = (V', E'):

V' = V ∪ {s} // add a new vertex, s

E' = E ∪ {(s, v) | v ∈ G.V} // edges from s to all original vertices

- G' has negative weight cycles if and only if G has.
- The only paths in G' involving s start from s (no inbound edges to s).

Set h(v) = $\delta$(s, v) for all v ∈ G.V.

Note: this ensures that $w$(u, v) = w(u, v) + h(u) - h(v) ≥ 0 as h(v) ≤ h(u) + w(u, v).

# JOHNSON(G, w)

```
1   Compute G′ = (G.V ∪ {s}, E ∪ {(s,v) | v ∈ G.V})
2   if !BELLMAN-FORD(G′, w, s) then error("Negative cycle!")
3   for (u,v) in G.E   w(u,v) = w(u,v) + G′.V[u].d - G′.V[v].d
4   let D = (d_uv) be a new matrix
5   for u in G.V
6       DIJKSTRA(G, w, u)
7       for v in G.V    d_uv = G.V[v].d - G′.V[u].d + G′.V[v].d
8   return D
```

h(x) = x.d = δ(s,x), as computed by Bellman-Ford

Undo the reweighting to restore original weights

# Analysis of JOHNSON(G, w)

- (Line 1) Computing G' costs O(|V|) time.
- (Line 2) BELLMAN-FORD takes O(|V'||E'|) = O(|V||E|).
- (Line 3) Calculating new edge weights take O(|E|) time.
- (Line 6) DIJKSTRA run |G.V| times costs $O(|V|^2 \lg |V| + |V||E|)$ time (using a clever priority queue), or O(|V||E| lg |V|) with a heap.
- (Line 7) Un-reweighting costs $O(|V|^2)$

Total cost is dominated by line 6: $O(|V|^2 \lg |V| + |V||E|)$.

As claimed, this is asymptotically faster than Floyd-Warshall if G is sparse!