Algorithms

Lent Term 2024/25

Dr John Fawcett
jkf21@cam.ac.uk

Computer Science Tripos, Part IA

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.

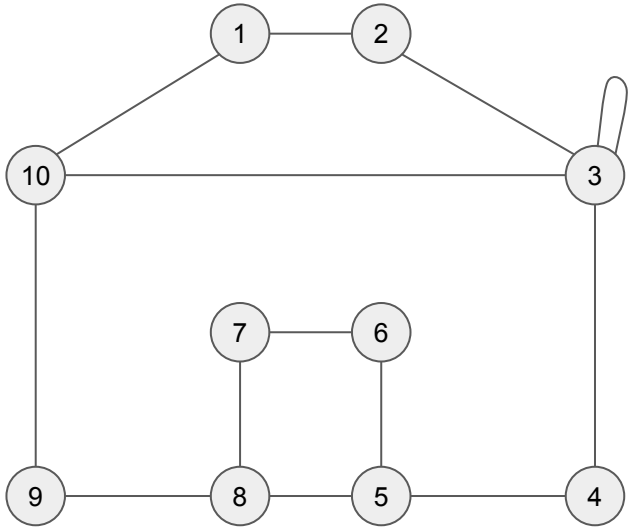
Representing a graph

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.

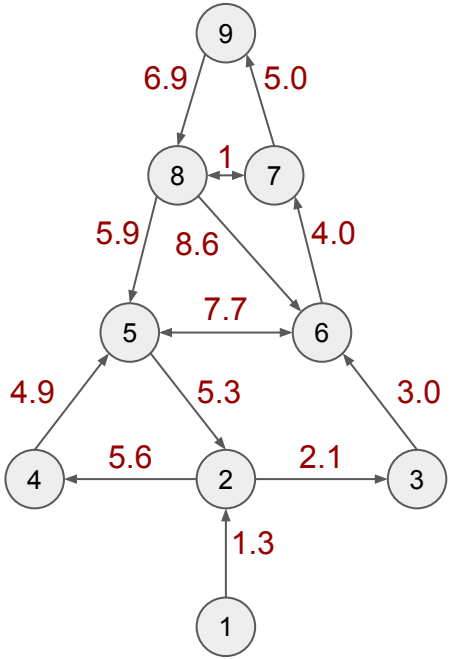
Example Adjacency Matrix

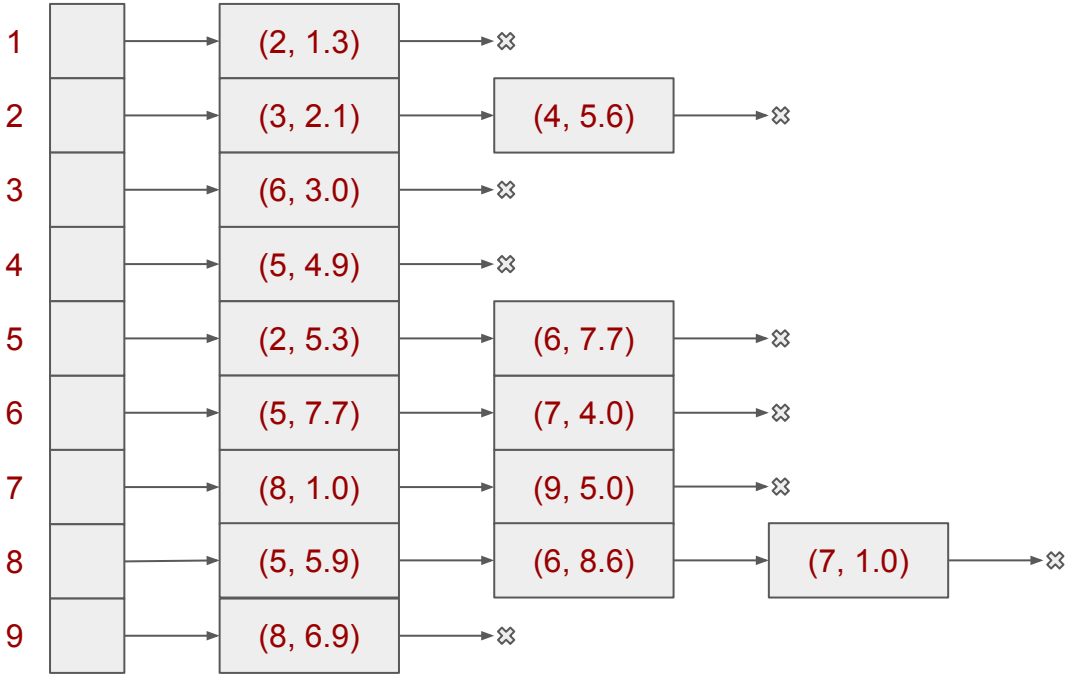


	1	2	3	4	5	6	7	8	9	10
1		1								1
2	1		1							
3		1	1	1						1
4			1		1					
5				1		1		1		
6					1		1			
7						1		1		
8					1		1		1	
9								1		1
10	1		1						1	

💡 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 significantly worsening the 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!).

An **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)
```

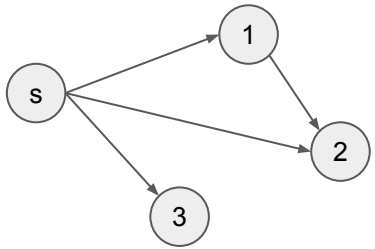
```
5   while !QUEUE-EMPTY(Q)
6       u = DEQUEUE(Q)
7       if (!u.marked)
8           u.marked = true
9           for v in G.E.adj[u]
10              ENQUEUE(Q, v)
```

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 (O(|E|) since *connected*) – 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 is 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 ≥ δ(s, u) and the assignment `v.d = u.d + 1` (line 15) gives us that:

v.d = u.d + 1
≥ δ(s, u) + 1
≥ δ(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]

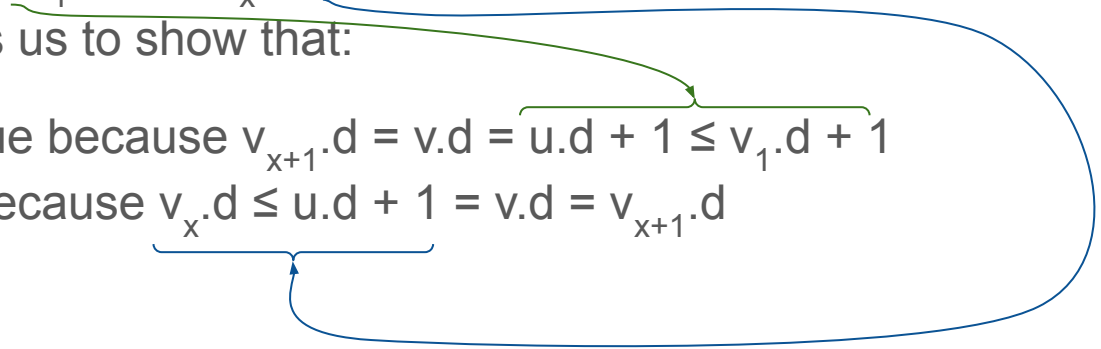
Next we show that the queue only ever contains vertices with at most two different values of v.d, using induction on the number of queue operations. **Lemma 3**: After each call to Enqueue and Dequeue, ϕ holds:
ϕ: if Q = v_1, v_2, v_3, .. v_x (head .. tail) then $v_x.d \leq v_1.d + 1$ and $v_i.d \leq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

Dequeue: if dequeuing v_1 leaves the queue empty, then ϕ holds vacuously. Otherwise, v_2 becomes the new head and we know (from ϕ 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 ϕ.

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 ϕ 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, ϕ, 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 ϕ 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 δ (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 that is 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 δ (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 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.

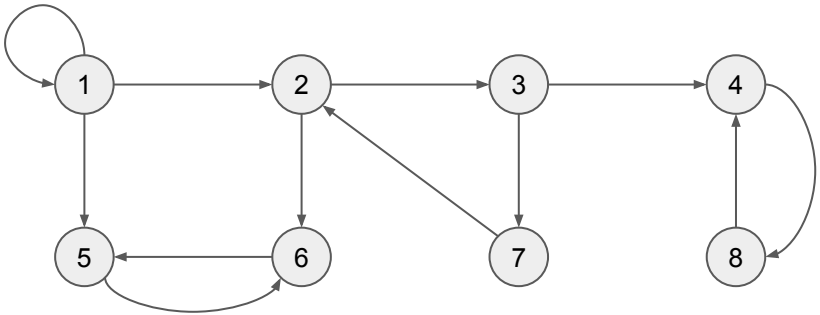
What’s the time?

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.

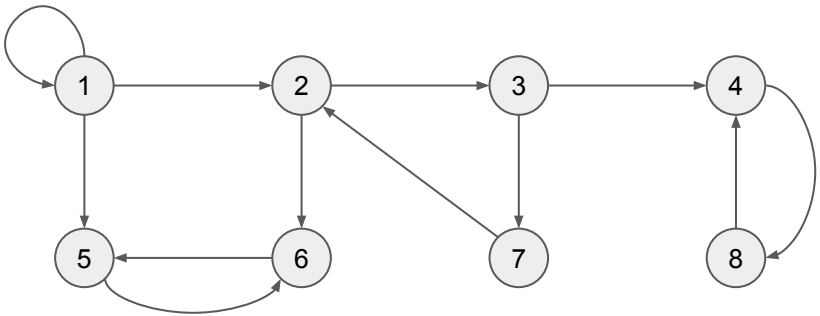
In the depth-first tree, a vertex v is a descendant of u if (and only if) v.discover_time is between u.discover_time and u.finish_time.

Example DFS



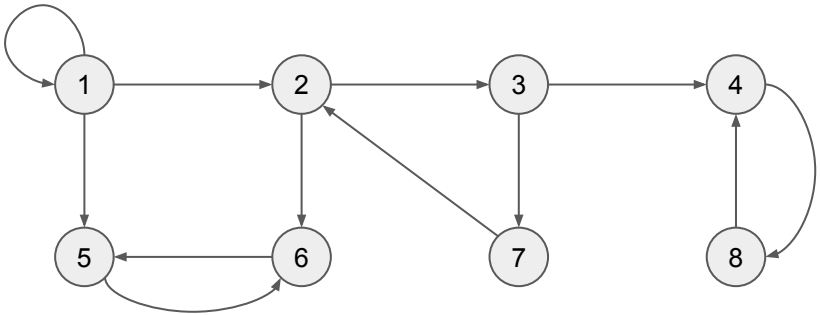
Node	Discover	Finish
1	1	
2		
3		
4		
5		
6		
7		
8		

Example DFS



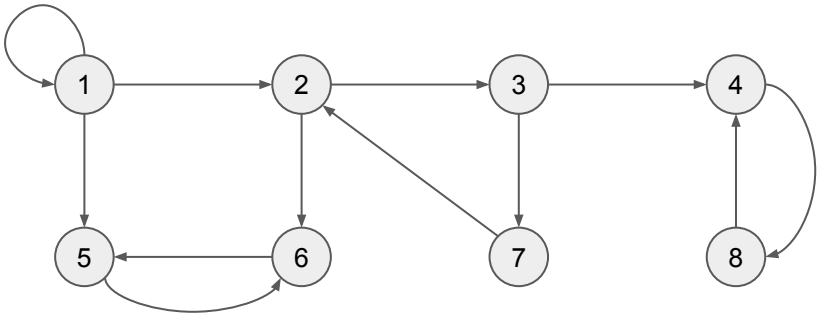
Node	Discover	Finish
1	1	
2	2	
3		
4		
5		
6		
7		
8		

Example DFS



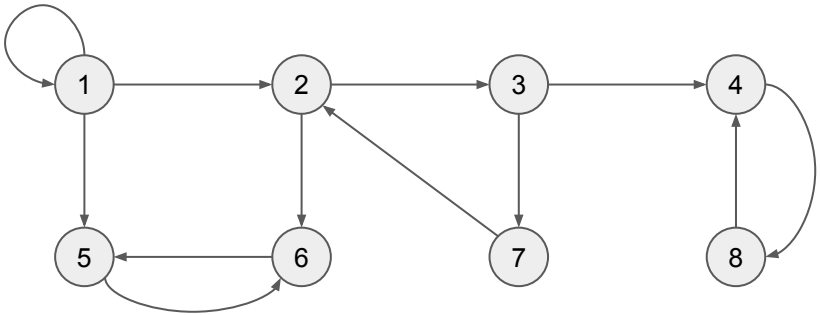
Node	Discover	Finish
1	1	
2	2	
3	3	
4		
5		
6		
7		
8		

Example DFS



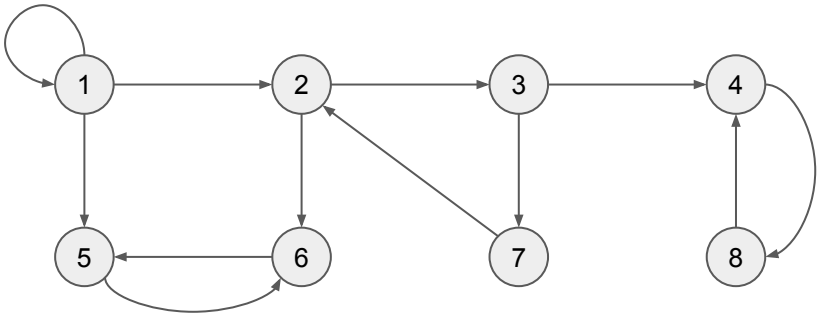
Node	Discover	Finish
1	1	
2	2	
3	3	
4	4	
5		
6		
7		
8		

Example DFS



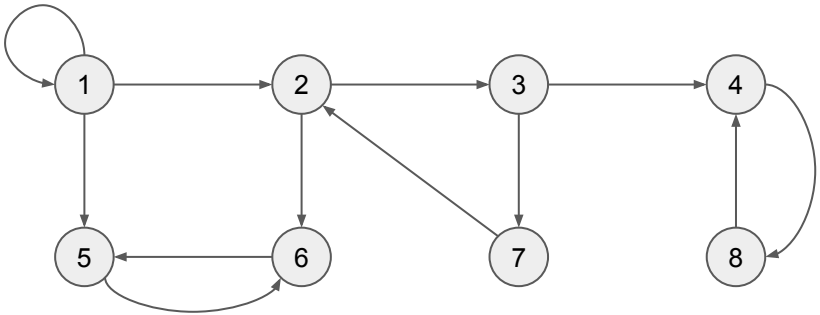
Node	Discover	Finish
1	1	
2	2	
3	3	
4	4	
5		
6		
7		
8	5	

Example DFS



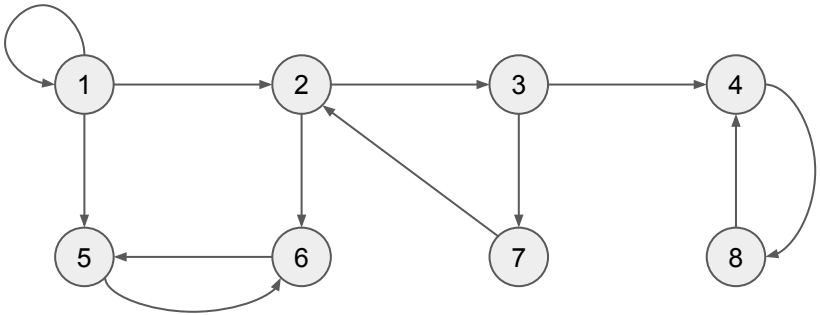
Node	Discover	Finish
1	1	
2	2	
3	3	
4	4	
5		
6		
7		
8	5	6

Example DFS



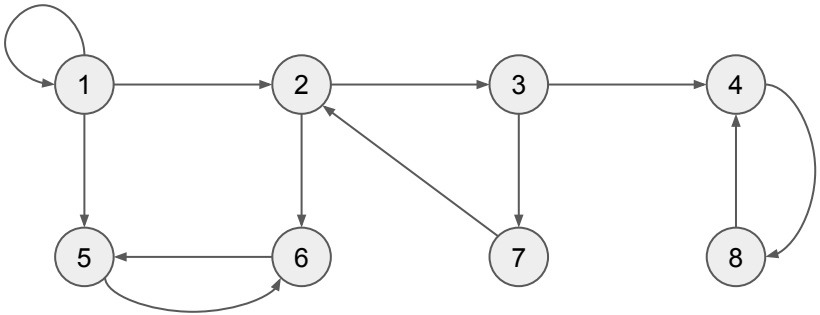
Node	Discover	Finish
1	1	
2	2	
3	3	
4	4	7
5		
6		
7		
8	5	6

Example DFS



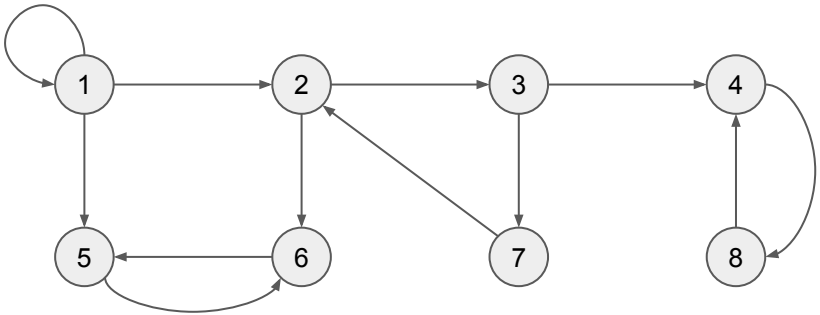
Node	Discover	Finish
1	1	
2	2	
3	3	
4	4	7
5		
6		
7	8	
8	5	6

Example DFS



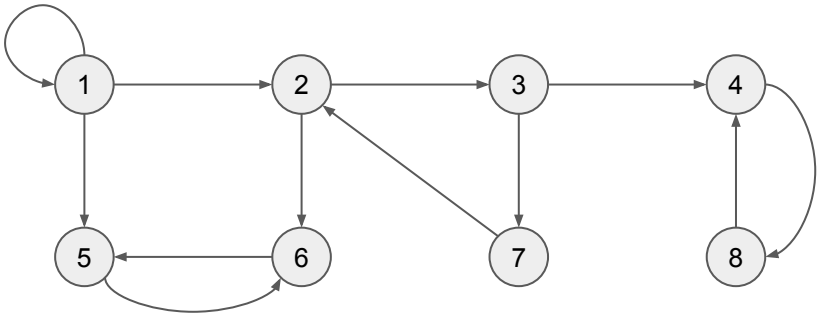
Node	Discover	Finish
1	1	
2	2	
3	3	
4	4	7
5		
6		
7	8	9
8	5	6

Example DFS



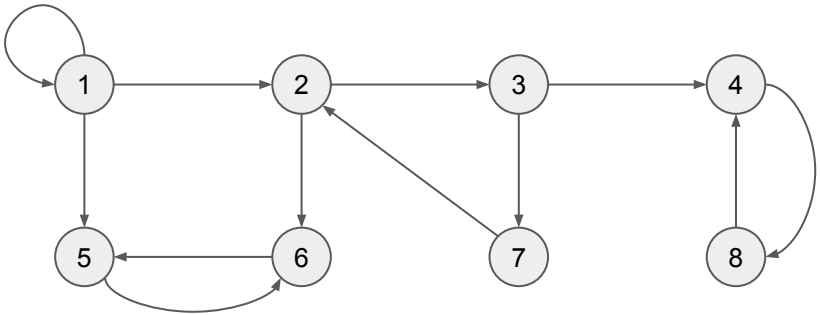
Node	Discover	Finish
1	1	
2	2	
3	3	10
4	4	7
5		
6		
7	8	9
8	5	6

Example DFS



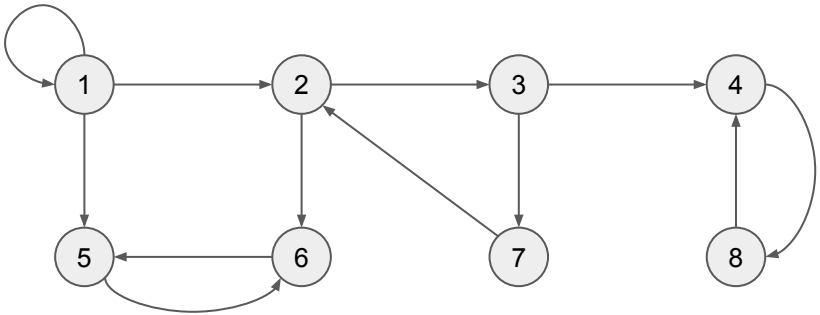
Node	Discover	Finish
1	1	
2	2	
3	3	10
4	4	7
5		
6	11	
7	8	9
8	5	6

Example DFS



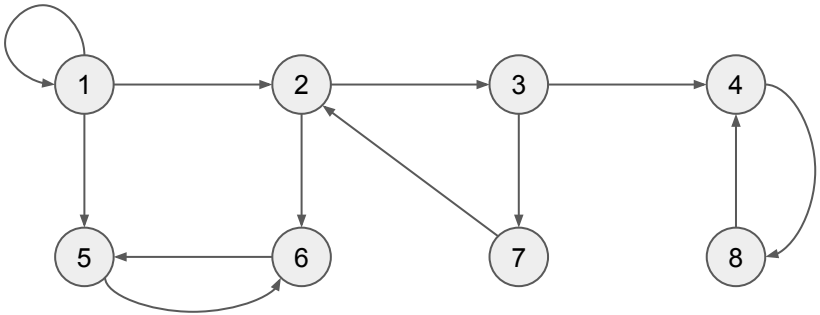
Node	Discover	Finish
1	1	
2	2	
3	3	10
4	4	7
5	12	
6	11	
7	8	9
8	5	6

Example DFS



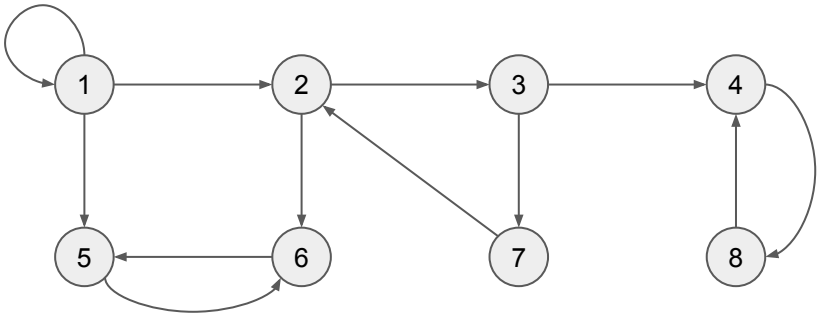
Node	Discover	Finish
1	1	
2	2	
3	3	10
4	4	7
5	12	13
6	11	
7	8	9
8	5	6

Example DFS



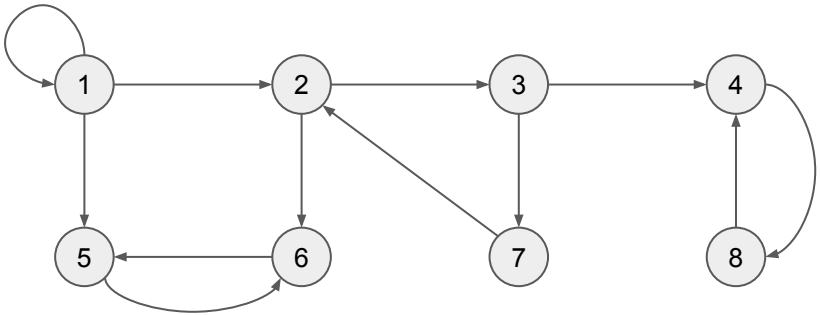
Node	Discover	Finish
1	1	
2	2	
3	3	10
4	4	7
5	12	13
6	11	14
7	8	9
8	5	6

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:

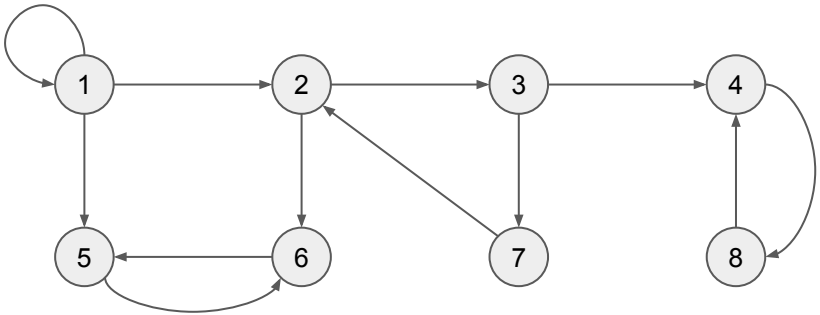
Input: a directed graph, G = (V, E)

Output: the strongly connected components of G

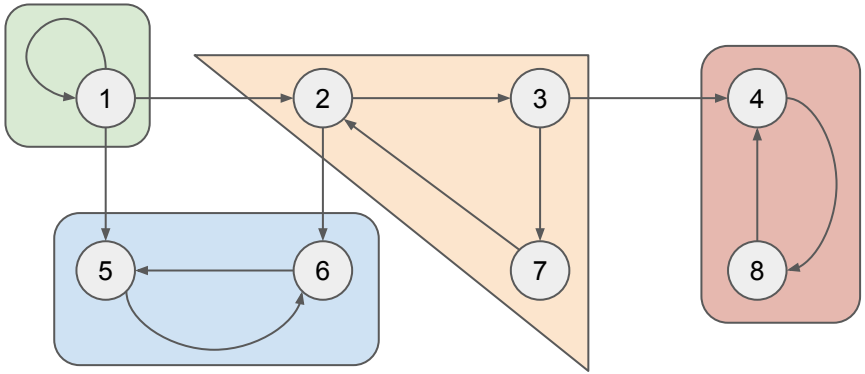
A **strongly connected component** is a maximal set of vertices $C \subseteq V$ such that for all $u,v \in C$, we have *both* that v is reachable from u *and* that u is reachable from v using edges in E.

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

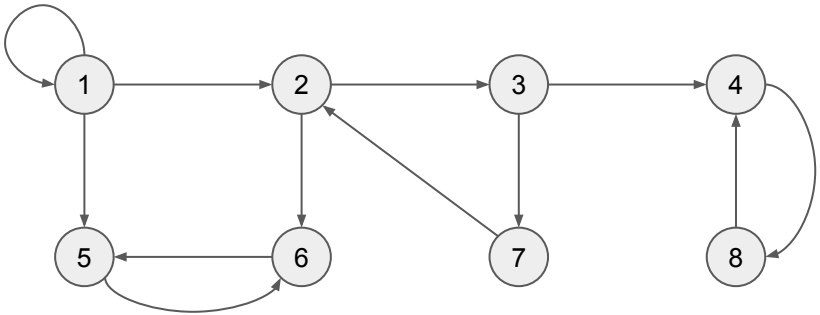
💡 See CLRS chapter 22

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.

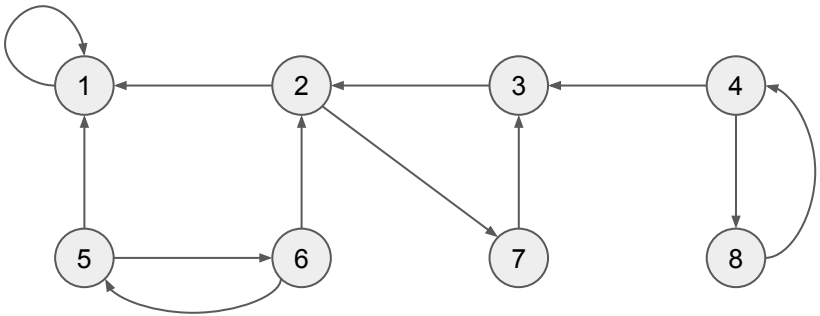
Proof of correctness is in CLRS chapter 22 (pages 617–620 of 3rd edition).

SCC1: Run DFS on original graph



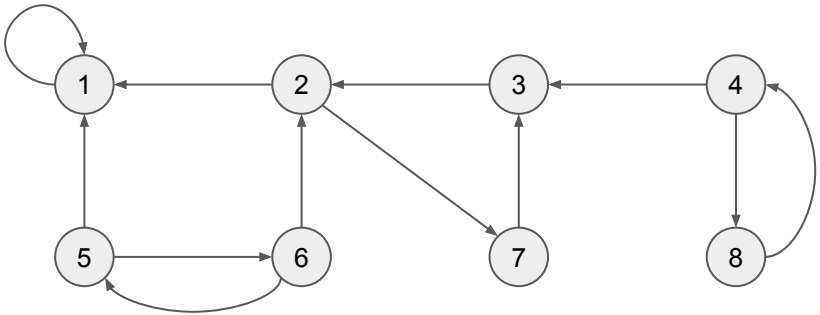
Node	Discover	Finish
1	1	16
2	2	15
3	3	10
4	4	7
5	12	13
6	11	14
7	8	9
8	5	6

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

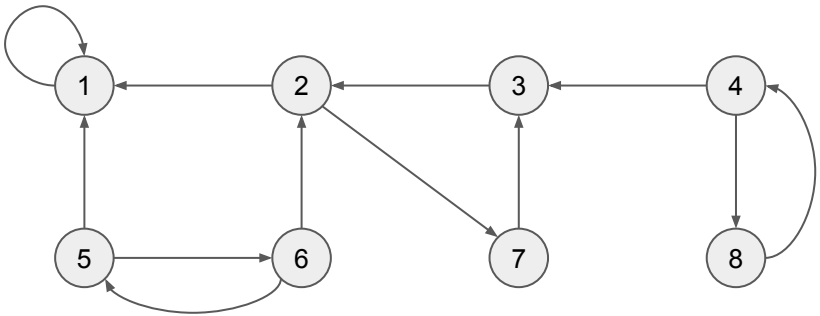
SCC3a: Reverse sort nodes by finish, DFS in that order



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		

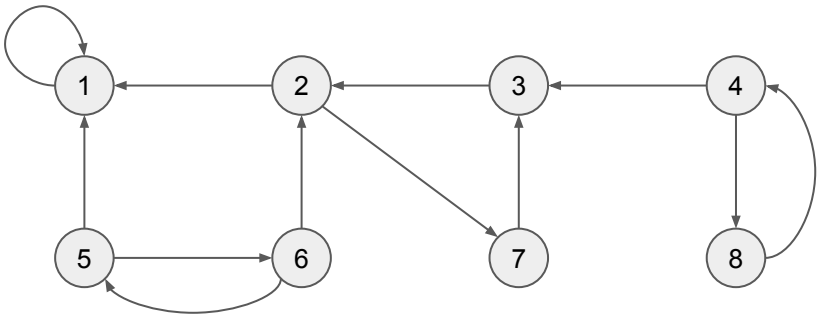
SCC3b: Continue DFS in that order



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.

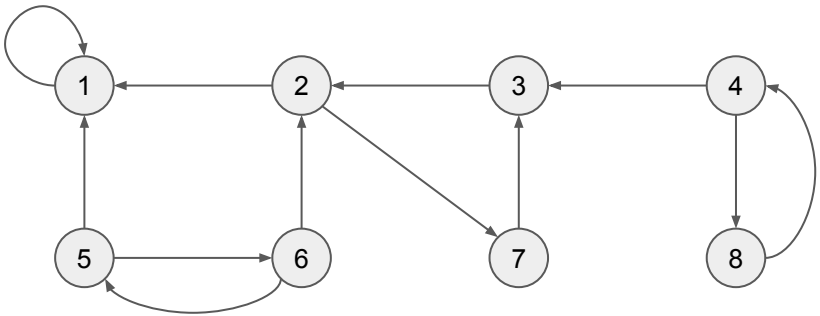
SCC3c: Continue DFS in that order



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.

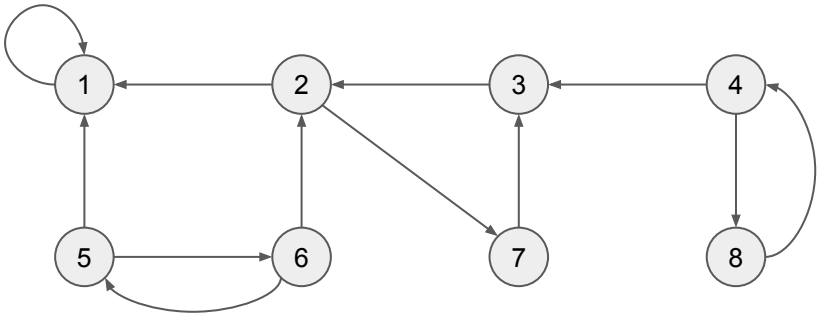
SCC3d: Continue DFS in that order

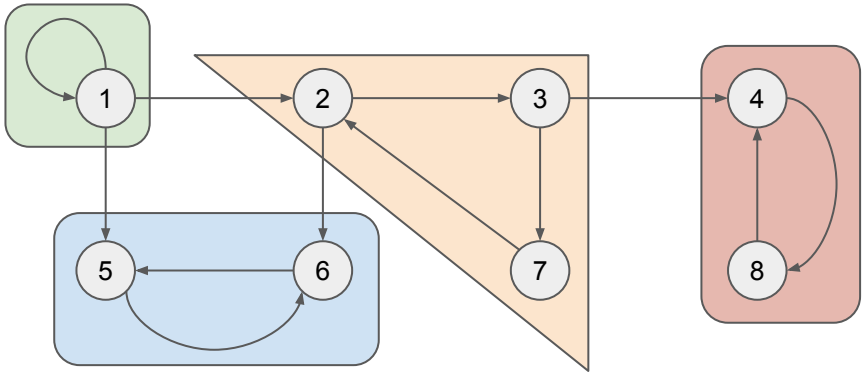


Finish1	Node	Discover	Finish
16	1	1	2
15	2	3	8
14	6	9	12
13	5	10	11
10	3	5	6
9	7	4	7
7	4	13	16
6	8	14	15

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 π 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
```

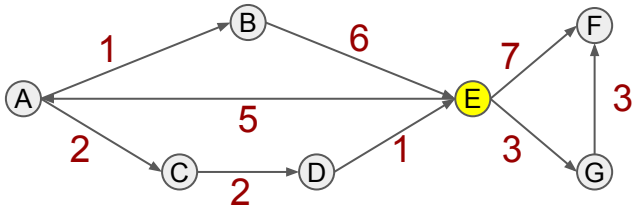
RELAX(u, v, w)

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

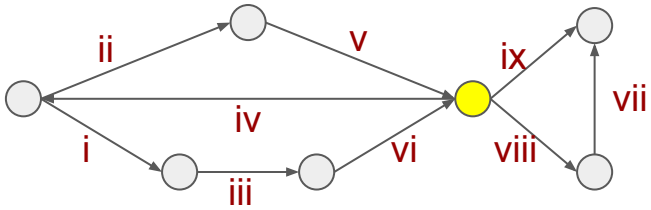
```
RELAX(u, v, w)
    if v.d > u.d + w(u, v)    return false
```

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)

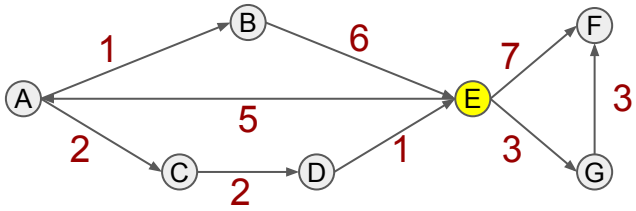


Order of edges in E

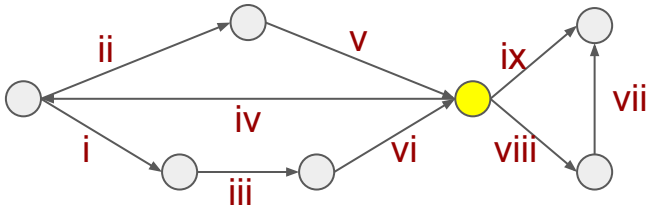
Initialisation

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)



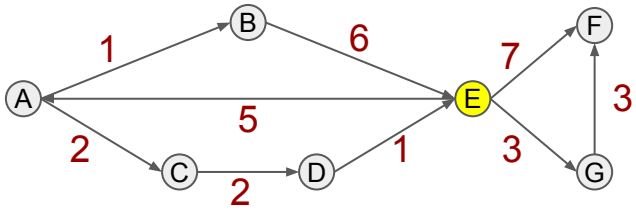
Order of edges in E

Iteration i=1

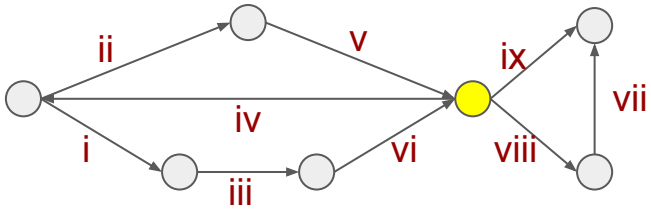
Vertex	d	π
A	5	E
B	∞	NIL
C	∞	NIL
D	∞	NIL
E	0	NIL
F	7	E
G	3	E

💡 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)



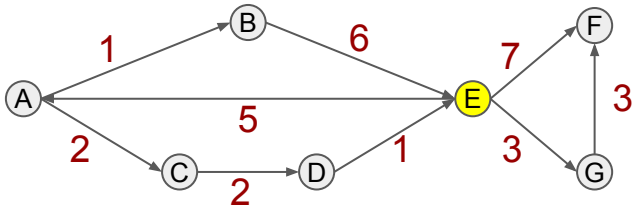
Order of edges in E

Iteration i=2

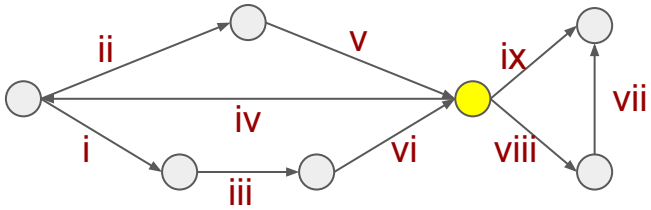
Vertex	d	π
A	5	E
B	6	A
C	7	A
D	9	C
E	0	NIL
F	6	G
G	3	E

💡 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)

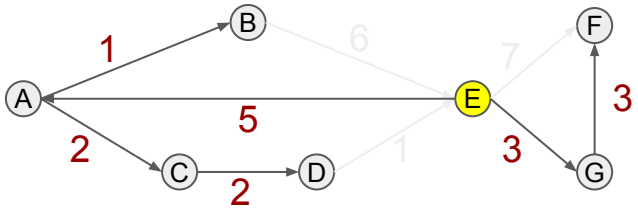


Order of edges in E

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”)

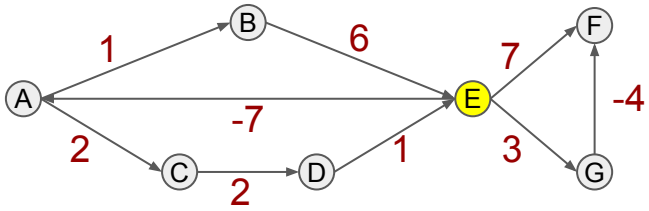


Shortest paths (output)

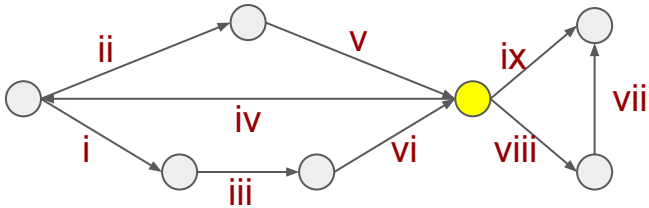
Vertex	d	π
A	5	E
B	6	A
C	7	A
D	9	C
E	0	NIL
F	6	G
G	3	E

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)

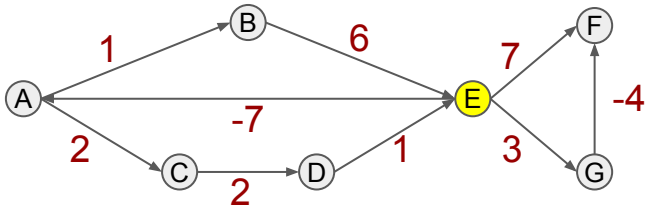


Order of edges in E

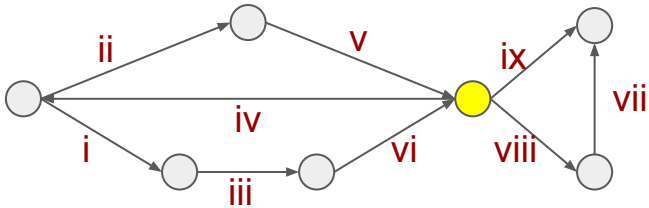
Initialisation

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)

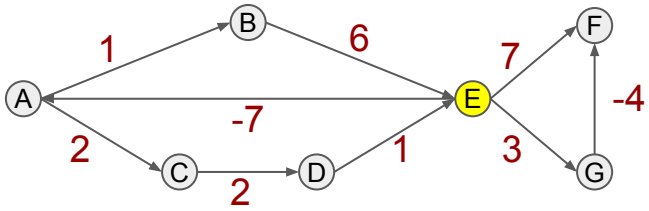


Order of edges in E

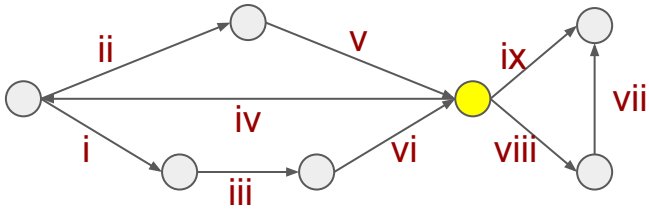
Iteration i=1

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)

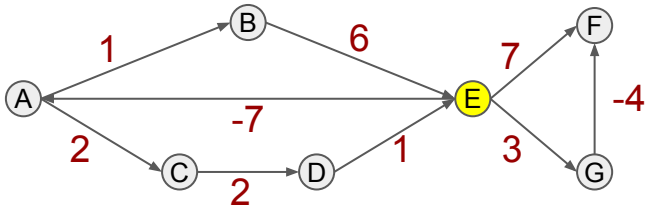


Order of edges in E

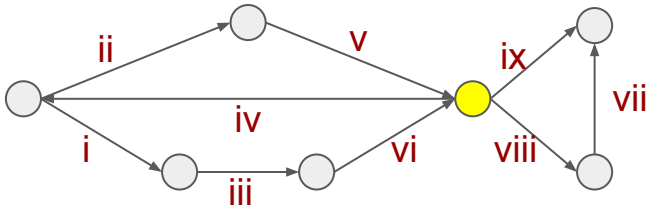
Iteration i=2

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)

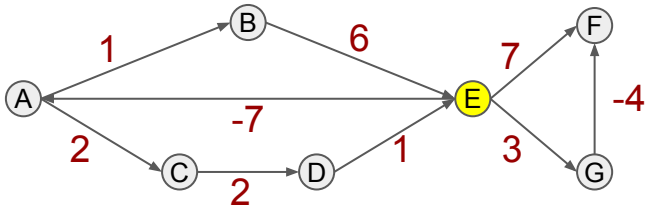


Order of edges in E

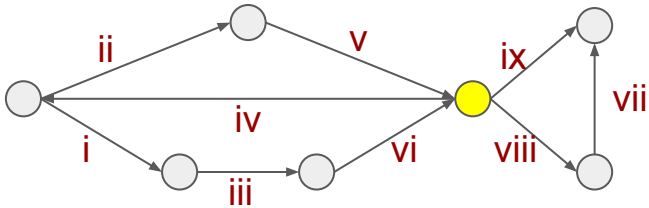
Iteration i=3

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)

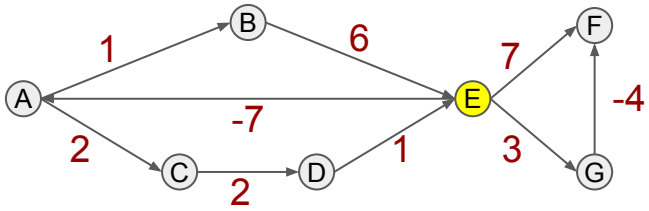


Order of edges in E

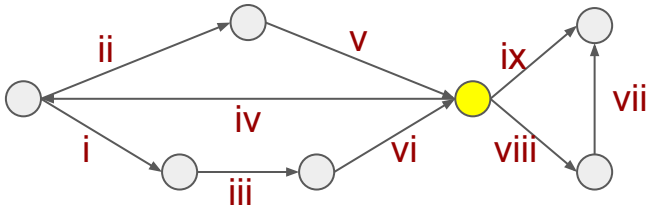
Iteration i=4

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)

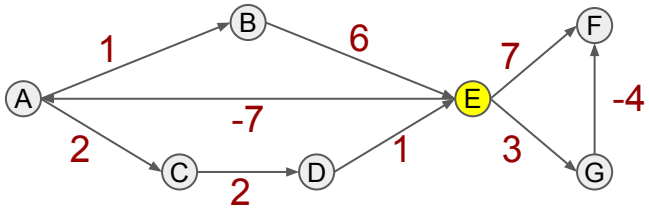


Order of edges in E

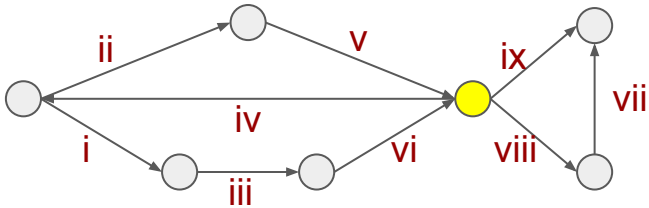
Iteration i=5

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)

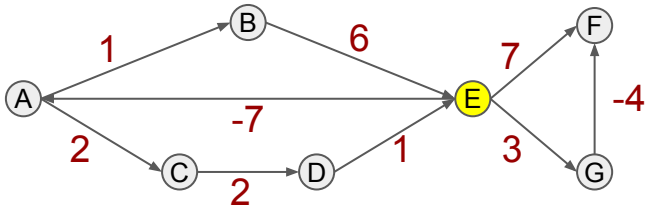


Order of edges in E

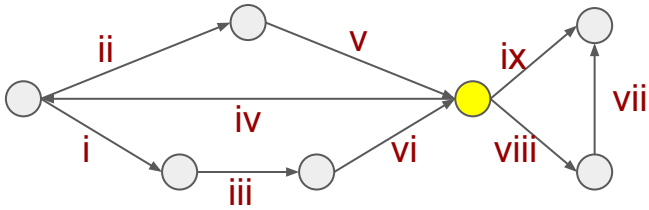
Iteration i=6

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)

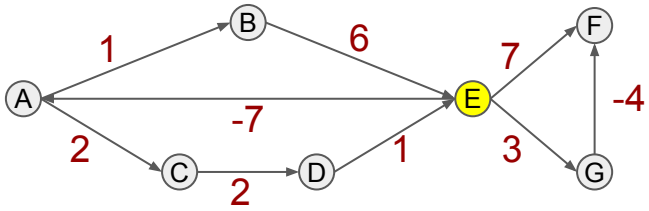


Order of edges in E

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).

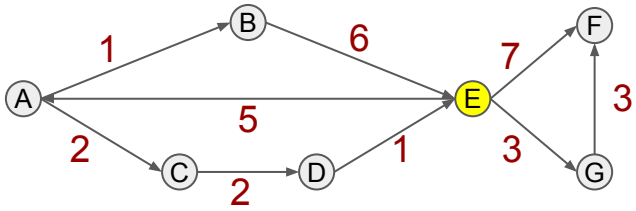
DIJKSTRA(G, w, s)

```
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”)



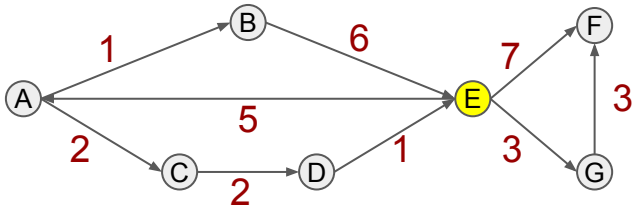
Weighted, directed graph (input)

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”)



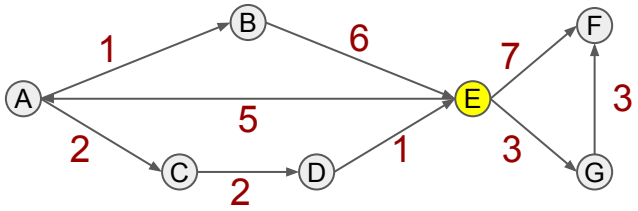
Weighted, directed graph (input)

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”)



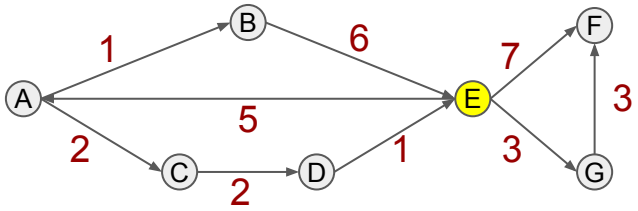
Weighted, directed graph (input)

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”)



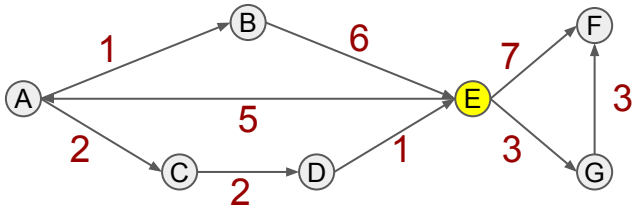
Weighted, directed graph (input)

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”)



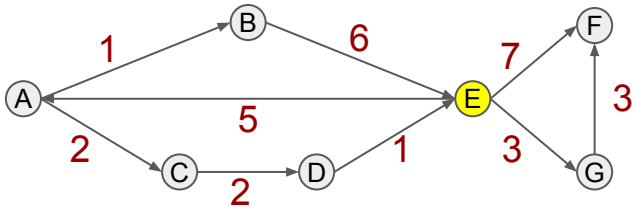
Weighted, directed graph (input)

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”)



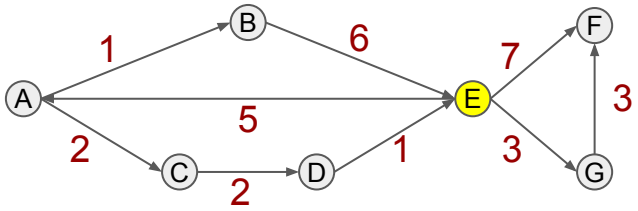
Weighted, directed graph (input)

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”)



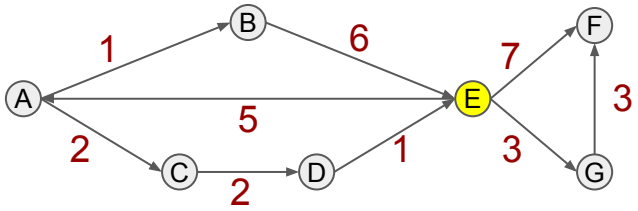
Weighted, directed graph (input)

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”)



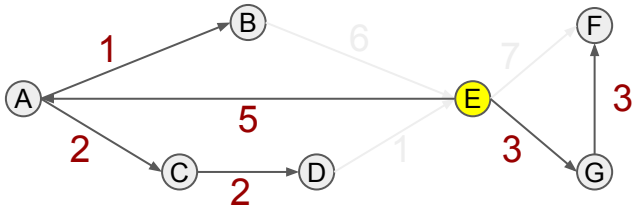
Weighted, directed graph (input)

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 = δ(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):

ϕ: v.d = δ(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 ≠ δ(s, u). Consider the iteration of the WHILE loop that added u to S.

We know that u ≠ s since s.d = 0 = δ(s, s), and hence S ≠ ∅ when u was added. There must be some path s ⇝ u to be found, since otherwise u.d = ∞ = δ(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 to 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 ϕ 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. ϕ is true and we have proved the correctness of Dijkstra's algorithm.

It follows that the predecessor subgraph G_π, is a shortest path tree rooted at s, i.e. not only are the distances correct but the paths obtained by following the π 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, π) 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 (although if s is first there is no need since all other keys are infinite).

PQ-EXTRACT-MIN takes 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.

Remember that we want…

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) = δ(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!

Algorithms 2

Section 2: Graphs and Subgraphs

Flow Networks [1]

Flow networks concern weighted, directed graphs $G = (V, E)$.

V contains two distinguished vertices, s and t, known as the source and the sink of the flow.

We require two properties of E:

1. No self loops at any vertex: $\forall v \in V . (v, v) \notin E$
2. No antiparallel edges: $\forall u,v \in V . (u, v) \in E \rightarrow (v, u) \notin E$

The weights, known as **capacities**, are non-negative: $\forall (u, v) \in E . c(u, v) \geq 0$ and it will be convenient to define $c(u, v) = 0$ if $(u, v) \notin E$.

Flow Networks [2]

All vertices are on some path $s \rightsquigarrow v \rightsquigarrow t$ so $|E| \geq |V| - 1$ (every vertex other than s must have at least one inbound edge).

Put another way, we can delete any vertex v (and its incident edges) if v is not reachable from s or t is not reachable from v. For the problems we want to solve, such vertices never alter the solution.

Definition of a flow

A **flow** f(u, v) in G is a function of type $V \times V \to \mathbb{R}$ with two properties:

1. Flows are subject to the **capacity constraint**: $\forall u,v \in V \,.\, 0 \leq f(u, v) \leq c(u, v)$
2. At every vertex $u \in V \setminus \{s, t\}$, we have **flow conservation**:

   $$\Sigma_{v \in V} f(v, u) = \Sigma_{v \in V} f(u, v)$$

where f(u, v) = 0 if $(u, v) \notin E$.

The flow is defined between all pairs of vertices in G and is known as the flow from u to v.

Value of a flow

We denote the value of a flow f as |f| where

$$|f| = \Sigma_{v \in V} f(s, v) - \Sigma_{v \in V} f(v, s)$$

⚠️ |..| is not absolute value or set cardinality!

The second term is usually zero because there is no flow into the source but, as we will see, we want to generalise the networks to which we apply this idea and the edges into the source will not always have zero weight.

Maximum Flow Problem

Input: a flow network, i.e. a directed graph $G = (V, E)$ with edge capacities $c(u, v) \geq 0$, and two distinguished vertices $s,t \in V$ being the source and sink.

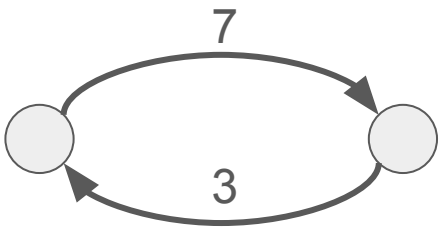
Output: any flow having maximum value.

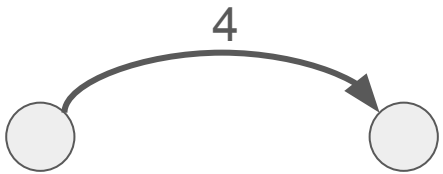
Note that we seek to determine the flow, not just the flow value.

Antiparallel edges [1]

We said that we do not allow having both (u, v) and (v, u) be edges in E. Several algorithms that solve the Maximum Flow Problem require this.

We cannot simplify antiparallel edges to a single edge with the net capacity because we might want to use only the capacity in the smaller magnitude direction, or all the capacity in the larger direction.





Antiparallel edges [2]

We can handle antiparallel edges by introducing additional vertices to split one of the edges. Two new edges are assigned the same capacity as the original they replace.

This means we can require no antiparallel edges without limiting the set of problems our algorithms can solve.

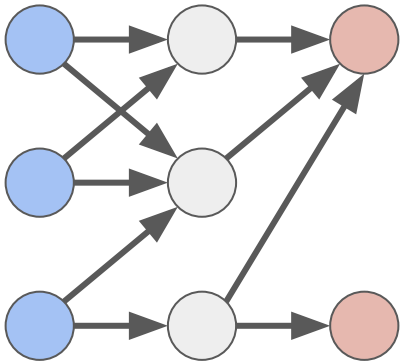
Supersources and Supersinks [1]

If we want to model a system where flow originates from multiple sources (s_1 .. s_m) and is consumed by multiple sinks (t_1 .. t_n), we can add additional vertices and edges:

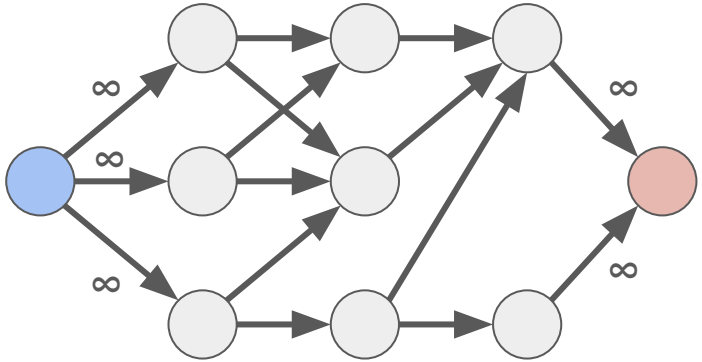
- two additional vertices for the supersource s and supersink t
- edges (s, s_i) for i = 1 .. m and (t_j, t) for j = 1 .. n, all with capacity c = ∞

This reduces the multiple source, multiple sink problem to the single source, single sink problem. We lose no generality by only considering solutions to the single source, single sink problem.

Supersources and Supersinks [2]



3 sources, 2 sinks



1 source, 1 sink

Ford-Fulkerson Methods

Ford and Fulkerson covers several algorithms based on a few key ideas, which we can also use to solve related problems:

1. Residual networks
2. Augmenting paths
3. Cuts

Residual Networks

Given a flow network G = (V, E) and a flow f, the **residual network** G_f contains **residual edges** showing how we can change the flow:

1. If an edge $(u, v) \in E$ and $f(u, v) < c(u, v)$ then we can add more flow to the edge: up to $c(u, v) - f(u, v)$ more. NB: there is no edge if $f(u, v) = c(u, v)$!!

2. If $f(u, v) > 0$ then we can **cancel** flow that is already present by adding flow in the reverse direction: up to $f(u, v)$ along edge (v, u) [note reverse direction]

Note that (2) allows the residual network to contain edges not in G, and G_f might include antiparallel edges.

Residual Capacity

Given a flow network G = (V, E) and a flow f, the residual edges (u, v) in the residual network G_f have **residual capacities** $c_f(u, v)$ where

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{if } (u, v) \in E \\ f(v, u) & \text{if } (v, u) \in E \\ 0 & \text{otherwise} \end{cases}$$

Exactly one case applies because of the antiparallel edge constraint on E.

Augmentation

Any flow f’ in the residual network G_f can be added to the flow f to make a valid flow because the flow assigned to every edge cannot exceed its capacity and cannot become negative. This is **augmentation**, written as f □ f’.

(f □ f’) (u, v) = f(u, v) + f’(u, v) - f’(v, u) if (u, v) ∈ E

(f □ f’) (u, v) = 0 otherwise

The value of the augmented flow, |f □ f’| = |f| + |f’|.

Augmenting Paths

Given a flow network G and a flow f, an **augmenting path** is a simple path p from s to t in the residual network.

The maximum amount by which we can increase the flow along each edge in p is called the **residual capacity** of the path p:

$c_f(p) = \min\{c_f(u, v) \mid (u, v) \text{ is on } p\}$

Notice that if we augment flow f with the residual capacities along each edge (u, v) on p, then we get a flow with strictly larger value: $|f \uparrow f_p| = |f| + |f_p| > |f|$.

FORD-FULKERSON(G, s, t)

```
1   Initialise flow f to 0 on all edges
2   while there exists an augmenting path p in
                          the residual network G_f
3       augment the flow f along p
4   return f
```

Why do we get a maximum flow?

We saw that Ford-Fulkerson augments a flow using augmenting paths until no more augmenting paths can be found.

The question to be answered is whether it is guaranteed that Ford-Fulkerson terminates only when a maximum flow has been found.

The Max-Flow Min-Cut Theorem tells us that this technique will work.

Cuts [1]

A **cut** (S, T) of a flow network G = (V, E) is a partition of V into S and T = V \ S such that $s \in S$ and $t \in T$.

For a flow f, we define the **net flow** f(S, T) across the cut (S, T) as

$$f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in S} \sum_{v \in T} f(v, u)$$

Given a flow network G with source s and sink t, and a flow f, let (S, T) be any cut of G. The net flow across (S, T) is f(S, T) = |f|. (The proof follows from the definition of flow conservation.)

Cuts [2]

The **capacity** of the cut (S, T) is $c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v)$.

A **minimum cut** of a network is a cut whose capacity is minimum over all cuts of the network.

The value of any flow f in a flow network G is bounded from above by the capacity of any cut of G.

Max-Flow Min-Cut Theorem

If f is a flow in a flow network G = (V, E) with source s and sink t, then the following conditions are equivalent:

1. f is a maximum flow in G;
2. The residual network G_f contains no augmenting paths; and
3. |f| = c(S, T) for some cut (S, T) of G.

Proof of Max-Flow Min-Cut Theorem [1]

1. *f is a maximum flow in G;*
2. *The residual network G_f contains no augmenting paths; and*
3. *$|f| = c(S, T)$ for some cut (S, T) of G.*

Proof that 1 => 2:

Suppose f is a maximum flow in G but G_f contains an augmenting path p. The flow found by augmenting f using p has value $|f| + |f_p| > |f|$, which contradicts that f was maximum.

Note that $|f_p| > 0$ because we did not add edges to G_f with zero capacity.

Proof of Max-Flow Min-Cut Theorem [2]

1. *f is a maximum flow in G;*
2. *The residual network G_f contains no augmenting paths; and*
3. *$|f| = c(S, T)$ for some cut (S, T) of G.*

Proof that 3 => 1:

Remember that the value of any flow f in a flow network G is bounded from above by the capacity of any cut of G, i.e. $|f| \leq c(S, T)$.

If $|f| = c(S, T)$ then f must be a maximum flow.

Proof of Max-Flow Min-Cut Theorem [3]

1. *f is a maximum flow in G;*
2. *The residual network G_f contains no augmenting paths; and*
3. *$|f| = c(S, T)$ for some cut (S, T) of G.*

Proof that 2 => 3:

Suppose G_f has no augmenting paths (so no paths from s to t). Consider the partition (S, T) where $S = \{v \in V \mid \exists \text{ path from s to v in } G_f\}$, and $T = V \setminus S$.

(S, T) is a cut because $s \in S$ and $t \in T$.

Consider $u \in S$ and $v \in T$. Is $(u, v) \in E$, or is $(v, u) \in E$, or is neither in E?

Proof of Max-Flow Min-Cut Theorem [4]

If $(u, v) \in E$ then we must have $f(u, v) = c(u, v)$ since, otherwise, there would be residual capacity on the edge and (u, v) would be in E_f. That would place $v \in S$.

If $(v, u) \in E$ then we must have $f(v, u) = 0$ because, otherwise, $c_f(u, v) = f(v, u) > 0$ and we would have $(u, v) \in E_f$. That also places $v \in S$.

If neither is in E then $f(u, v) = f(v, u) = 0$.

$f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in S} \sum_{v \in T} f(v, u) = \sum_{u \in S} \sum_{v \in T} c(u, v) - 0 = c(S, T)$

We know that $|f| = f(S, T) = c(S, T)$, which proves statement 3.

Proof of Max-Flow Min-Cut Theorem [5]

1. *f is a maximum flow in G;*
2. *The residual network G_f contains no augmenting paths; and*
3. *$|f| = c(S, T)$ for some cut (S, T) of G.*

We have proven that 1 => 2 and that 2 => 3 and that 3 => 1, which suffices to show the equivalence of all three statements in the Max-Flow Min-Cut Theorem.

Basic FORD-FULKERSON(G, s, t)

```
1   for (u, v) in G.E    (u, v).f = 0
2   while there exists a path p from s to t in G_f
3       c_f(p) = min{c_f(u, v) | (u, v) is in p}
4       for (u, v) in p
5           if (u, v) ∈ G.E
6               (u, v).f = (u, v).f + c_f(p)
7           else
8               (v, u).f = (v, u).f - c_f(p)
```

Termination and Analysis

Interestingly, Ford-Fulkerson can fail to terminate if the edge capacities are irrational numbers: augmenting paths can add tiny amounts of additional flow in a series that is not convergent.

If all the capacities are integers, this cannot occur. We can find augmenting paths using breadth-first search or depth first search, costing $O(|E_f|)$ each time, which is $O(|E|)$ each time.

With integral capacities, the flow must increase by at least 1 each iteration so the cost of FORD-FULKERSON is $O(|E|\ |f^*|)$, where f^* is the maximum flow.

Optimisation: EDMUNDS-KARP(G, s, t)

We find *shortest* augmenting paths using breadth-first search on the residual network but with edge weights all set to 1.

It can be shown that this algorithm has $O(|V| \, |E|^2)$ running time.

(Proof is in CLRS3, pp 728–729.)

Maximum Bipartite Matchings

Given an undirected graph G = (V, E), a **matching** M $\subseteq$ E contains at most one edge that is incident at each vertex v $\in$ V.

A vertex v $\in$ V is **matched** if some edge in M is incident on v.
Other vertices are **unmatched**.

A **maximum matching** is a matching of maximum cardinality.

We are most interested in finding matchings within *bipartite graphs*.

Bipartite Graphs

An undirected bipartite graph $G = (V, E)$ where $V = V_1 \cup V_2$ and $E \subseteq V_1 \times V_2$.

We also assume that every vertex has at least one incident edge.

Example: your college's undergraduate rooms ballot can be modelled as a bipartite graph where the vertex set is $\{u \in \text{undergraduates}\} \cup \{r \in \text{rooms}\}$, and the edges represent the students' choices for which rooms they might like to live in next year.

Maximum Bipartite Matching Problem

Input: an undirected bipartite graph G = (V, E) where $V = V_1 \cup V_2$ and $E \subseteq V_1 \times V_2$.

Output: a matching $M \subseteq E$ with maximum cardinality.

We can use Ford-Fulkerson to find a maximum matching by transforming G into a flow network. Set the weight of every edge to 1 and add vertices {s, t} with edges {s, u} for $u \in V_1$ and {v, t} for $v \in V_2$ (these edges have infinite capacity). The solution requires O(|V||E|) time but the proof of correctness requires the Integrality Theorem.

It's worth converting the algorithm rather than the data…

Augmenting Paths in Unweighted Bipartite Graphs

An **Augmenting Path** with respect to a matching M in a bipartite graph G is an alternating path that starts at an unmatched vertex in V_1 and ends at unmatched vertex in V_2.

An **Alternating Path** with respect to a matching M in a bipartite graph G is a sequence of edges that are all in the graph's edge set and are alternately in the matching, not-in, in, not-in, …

Maximum Matchings in Unweighted Bipartite Graphs

MAXIMUM-MATCHING(G):

```
1   let M = ∅
2   do
3       let a = FIND-AUGMENTING-PATH(G, M)
4       M = M ⊕ a
5   while (a != NULL)
6   return M
```

Proof: ∃ augmenting path until M is maximum [1]

Let M’ be a maximum matching. M is the matching we have at the moment.

Consider the symmetric difference of M’ and M: every edge that is in M’ or M, but not in both. You can think of this as the XOR of the edge sets.

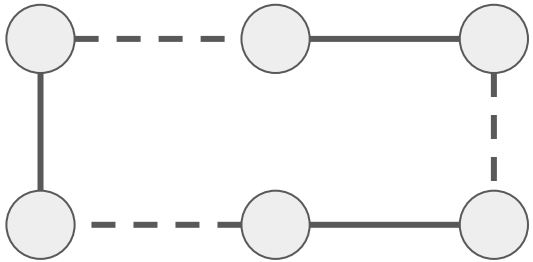
Notice that we cannot have two edges from M’ meeting at a vertex, nor two from M, since M’ and M are both matchings.

What structures can you find in the symmetric difference?

Proof: ∃ augmenting path until M is maximum [2]

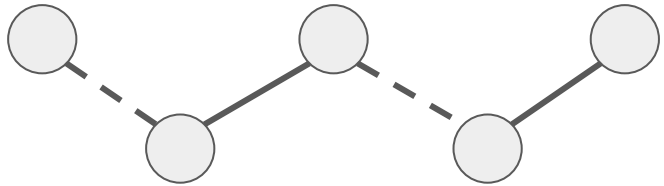
We can find isolated vertices.

We can find closed loops. Any closed loops can have any even length because, otherwise, we would have two edges from the same matching sharing a vertex.

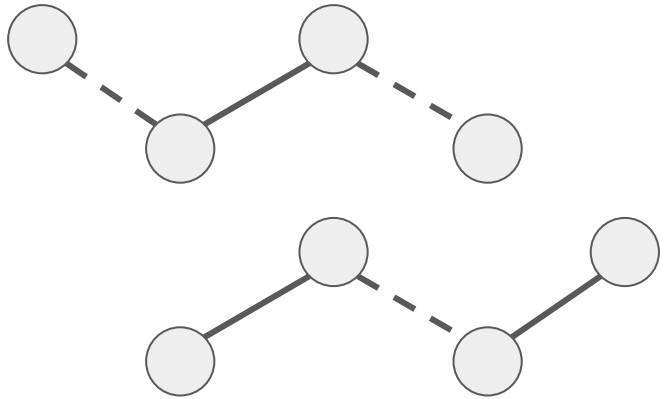


Proof: ∃ augmenting path until M is maximum [3]

We can have chains of even length.



We can have chains of odd length, in two ways: one more edge from M’ or one more edge from M.



Proof: ∃ augmenting path until M is maximum [4]

There are no other options because the maximum degree of any vertex is two: if we had three or more incident edges at any vertex, at least two would need to come from M or M', which is impossible because both are matchings.

We know that |M'| > |M| if M is not maximum, and in that case there must be at least one more edge from M' than from M in the symmetric difference.

The loops and even chains use the same number of edges from M' and M so there must be at least one odd-length chain with one more edge from M' than M.

That odd-length chain is an augmenting path with respect to M for G!

Finding Augmenting Paths

A simple method is to run a variant of BFS or DFS starting from each unmatched vertex in whichever of V_1 and V_2 has fewer unmatched vertices.

The search is constrained to taking edges $(u, v) \notin M$ for its first step.

If v is unmatched then we have an augmenting path, otherwise, we follow the edge $(v, w) \in M$ and allow the search to explore edges $(w, x) \notin M$ as the next step.

Repeat until either an augmenting path is found or the search gets stuck with no further edges to follow. If so, start a new search from the next unmatched starting vertex. If no search finds an augmenting path, there is none to be found.

Cost of Finding Augmenting Paths

The algorithm can find at most |V|/2 augmenting paths.

Each search costs O(|V| + |E|) = O(|E|) here (because the graph is connected).

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

We can do better…

HOPCROFT-KARP(G)

```
1   let M = ∅
2   do
3       a[] = ALL-VERTEX-DISJOINT-SHORTEST-AUGMENTING-PATHS(G, M)
4       M = M ⊕ a_1 ⊕ a_2 ⊕ … ⊕ a_a.length
5   while (a.length != 0)
6   return M
```

ALL-VERTEX-DISJOINT-SHORTEST-AUGMENTING-PATHS(G, M)

These are minimum length augmenting paths for M, with no common vertices.

Find them using a combination of a depth-first and breadth-first search, marking nodes on augmenting paths as they are found (to avoid finding multiple augmenting paths using the same vertices).

It can be shown that the WHILE loop requires only $\sqrt{|V|}$ iterations (by considering the maximum number of augmenting paths that can be found in each iteration).

HOPCROFT-KARP(G) requires $O(|E| \sqrt{|V|})$ running time!

Beware: Maximum and Maximal Matchings

A *maximum* matching is what we have been finding: a largest cardinality subset of non-adjacent edges in an input graph.

A *maximal* matching is a subset of non-adjacent edges that cannot be extended.

These are not the same!

Example: by making poor choices about the edges to include, it might be impossible to add more non-adjacent edges, but if you removed some and added different edges, it might be possible to get more in total.

Minimum Spanning Trees (MSTs)

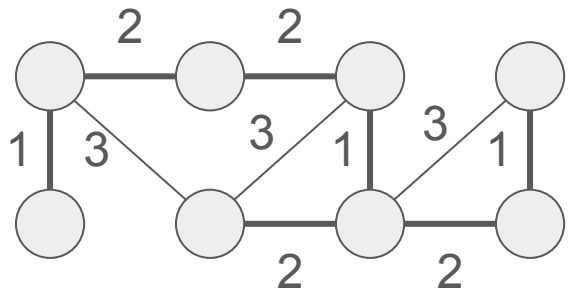
Input: a connected, undirected graph G = (V, E) with weight function w: E→ℝ.

Output: an acyclic subset T ⊆ E that connects all the vertices and whose total weight w(t) is minimal, where $w(t) = \sum_{(u,v) \in T} w(u,v)$.

Because T *does* connect all the vertices and T *is acyclic*, it must be that the edges in T form a tree. Any such tree is a **spanning tree**. T is not (necessarily) rooted.

A **minimum spanning tree** is a spanning tree with minimum total edge weights, and need not be unique.

Minimum Spanning Tree Example



Computing Minimum Spanning Trees

We will see two iterative, greedy algorithms that exploit *safe edges*.

Both algorithms iteratively increase a set (w.r.t. subset inclusion) A of edges, maintaining the property that $A \subseteq T$, for *some* T that is a minimum spanning tree.

As the algorithms run, edges $(u, v) \in E$ are added to A, always preserving the property that $A \subseteq T$, for *some* T that is a minimum spanning tree. A **safe edge** is one that can be added without violating the property.

Iteration continues until there are no more safe edges, at which point, A = T.

Cut, Cross, Respect, and Light

A **cut** $(S, V \setminus S)$ of an undirected graph $G = (V, E)$ is a partition of V.

An edge $(u,v) \in E$ **crosses** the cut $(S, V \setminus S)$ if $u \in S$ and $v \in V \setminus S$.

A cut **respects** a set A of edges if no edge in A crosses the cut.

An edge crossing a cut is a **light edge** if its weight is minimum of any edge crossing the cut. The minimum weight crossing the cut is unique but light edges are not necessarily unique: multiple crossing edges might have the same weight.

Safe Edge Theorem

Let G = (V, E) be a connected, undirected graph with real-valued weight function w defined on E.

Let A be a subset of E that is included in some minimum spanning tree for G.

Let (S, V \ S) be *any* cut of G that respects A.

Let (u, v) $\in$ E be a light edge crossing (S, V \ S).

$\Rightarrow$ (u, v) is a safe edge for A.

Proof of the Safe Edge Theorem [1]

Let T be a minimum spanning tree that includes A.

If T contains (u, v) then we are done.

If T does not contain (u, v), we can show that another minimum spanning tree T' exists and includes $A \cup \{(u, v)\}$. This makes (u, v) a safe edge for A.

Proof of the Safe Edge Theorem [2]

Add (u, v) to T and note that this forms a cycle (since T is a spanning tree and must already contain some unique path p = u $\rightsquigarrow$ v).

(u, v) crosses the cut (S, V \ S), and there must be at least one edge in T, on the path p, that also crosses the cut (since T is connected).

Let (x, y) be such an edge. (x, y) is not in A because the cut respects A.

Remove (x, y) from T and add (u, v) instead: call this T'. T' must be connected and acyclic (a tree).

Proof of the Safe Edge Theorem [3]

Calculate a bound on the weight of edges in T':

$w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$

The final inequality is because (u, v) is a light edge crossing (S, V \ S), i.e. for any other edge (x, y) crossing the cut, $w(u, v) \leq w(x, y)$.

Since T was a minimum spanning tree, T' must also be a minimum spanning tree.

So why is (u, v) a safe edge for A? That's because $A \subseteq T'$ since $A \subseteq T$ and the removed edge $(x, y) \notin A$, so $A \cup \{(u, v)\} \subseteq T'$. Because T' is an MST, (u, v) is a safe edge for A.

Corollary

Let G = (V, E) be a connected, undirected graph with real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, and let C = (V_C, E_C) be a connected component (tree) in the forest G_A = (V, A). If (u, v) is a light edge connecting C to some other component in G_A then (u, v) is a safe edge for A.

Kruskal’s Algorithm

Kruskal’s algorithm finds safe edges to add to a growing forest of trees by finding least-weight edges that connect any two trees in the forest.

The corollary tells us that any such edge must be a safe edge (for either tree) because it is the lightest edge crossing the cut that separates that tree from the rest of the graph.

The algorithm resembles that used to find connected components.

MST-KRUSKAL(G, w)

```
1   A = ∅
2   S = new DisjointSet; for v in G.V   MAKE-SET(S, v)
3   MERGE-SORT(G.E)          // Or any other non-decreasing sort
4   for (u, v) in G.E        // Can also stop if |A|=|V|-1
5       if !IN-SAME-SET(S, u, v)
6           A = A ∪ {(u, v)}
7           UNION(S, u, v)
8   return A
```

Analysis of MST-KRUSKAL(G, w)

Creating a disjoint set with |V| separate sets costs Θ(|V|).

In the worst case, the FOR loop runs to completion: |E| iterations performing 1 IN-SAME-SET check each, and |V|-1 calls to UNION across all the iterations.

The total cost is ~O(|E| + |V|) since both disjoint-set representation operations cost ~O(1).

The sort costs O(|E| lg |E|), which is O(|E| lg |V|) since G is connected.

The cost of sorting dominates and we state that MST-KRUSKAL costs O(|E| lg |V|).

Prim's Algorithm

Prim's algorithm maintains that A is a single tree (not a forest), and adds safe edges between the tree and an isolated vertex, to increase the size of the tree until |A| = |V|. Prim's algorithm starts from an arbitrary vertex $r \in V$.

The corollary tells us that that any such edge must be a safe edge because they are the lightest edges crossing the cut that separates the tree from the rest of the graph.

The algorithm resembles Dijkstra's algorithm, used to find single-source shortest paths.

MST-PRIM(G, w, r)

```
1   Q = new PriorityQueue
2   for v in G.V    v.key = ∞; v.π = NIL; PQ-ENQUEUE(Q, v)
3   PQ-DECREASE-KEY(Q, r, 0)
4   while !PQ-IS-EMPTY(Q)
5       u = PQ-EXTRACT-MIN(Q)
6       for v in G.E.adj[u]
7           if v ∈ Q && w(u, v) < v.key
8               v.π=u; v.key=w(u,v); PQ-DECREASE-KEY(Q,v,v.key)
```

Analysis of MST-PRIM(G, w, r) [1]

If we use a Fibonacci Heap as the implementation of the Priority Queue ADT then the |V| calls to PQ-ENQUEUE cost O(1) amortised each (FH-INSERT).

The WHILE loop executes |V| times and each call to FH-EXTRACT-MIN costs O(lg V) time so the total time is O(|V| lg |V|). (We should sum the costs as the size of the PQ decreases but this over-approximation turns out to be asymptotically accurate.)

Analysis of MST-PRIM(G, w, r) [2]

Across all iterations of the WHILE loop, the FOR loop covers every edge exactly twice (once in each direction).

We need to test for membership of the Priority Queue, which is not a supported operation in the ADT. We can implement this with a bit string: one bit per vertex, initialise to 11..1 and set bits to zero when extracted; test membership looks at the corresponding bit. This test for membership becomes O(1) time, and the updates do not add to the corresponding big-O costs because they are O(1) per vertex.

The calls PQ-DECREASE-KEY, cost O(1) amortised for the Fibonacci Heap implementation.

Analysis of MST-PRIM(G, w, r) [3]

The total cost of MST-PRIM(G, w, r) is O(|E| + |V| lg |V|) amortised, which is better than MST-KRUSKAL.

Either term could be dominant, depending on the size of the edge set.

If we used a binary heap, MST-PRIM(G, w, r) would cost O(|E| lg |V| + |V| lg |V|), which is O(|E| lg |V|) if G is connected so |E| > |V|-1.

Algorithms 2

Section 3: Advanced Data Structures

Amortised Analysis

Sometimes, a worst-case analysis is too pessimistic.

For example, consider a **vector**: an array that grows when necessary by allocating an array of twice the size and copying existing elements into the new array. The worst case cost of INSERT would assume that resizing is necessary.

Three common methods can be used to give more representative cost estimates:

1. Aggregate Analysis
2. The Accounting Method
3. The Potential Method

Aggregate Cost of Vector Insert

Let's start with an array of 16 items. The first 16 inserts take O(1) time. The 17th insert allocates an array of size 32, copies 16 items in O(n) time since n=16 at that point, then inserts one more item in O(1) time. The next 15 inserts take O(1) time and the next uses O(n) time again.

If we perform $N=2^k$ inserts, the total cost is:

$16 + (16+1) + 15 + (32+1) + 31 + (64+1) + 63 + \ldots$

…which has sum $\in O(N)$. Dividing by N inserts, we conclude that the typical cost per insert is $O(N)/N = O(1)$ amortised, per item.

💡 This proves a result we assumed in Algorithms 1 (see Stacks)

Accounting Method

The accounting method is more sophisticated.

We declare the amortised cost for each operation as the amount we charge our customer. Amortised costs might exceed the actual costs, with the excess going into a ‘credit’ account. When an amortised cost is less than the actual cost, the ‘credit’ pays for the shortfall.

The accounting method yields a valid set of amortised costs provided for *any sequence of operations*, the total amortised cost is an upper bound for the actual cost, and the credit never goes negative.

Potential Method

The potential method is similar but does not attribute ‘credit’ to particular operations or items within the data structure.

Instead, we measure the potential of the whole data structure.

$\phi(d_i)$ is the potential of the data structure in each state, i, it can get into through sequences of the supported operations. We require that $\phi(\text{initial}) = 0$ and that $\phi(d_i) \geq 0$ for all states, i.

Each operation’s amortised cost is the sum of the actual cost and the change in potential caused by the operation.

Example of the Potential Method [1]

Suppose we have a binary counter stored as a list of bits. We can use the potential function to calculate the amortised cost of INCREMENT, which adds one to the current value represented in binary by the string of bits.

We can use the number of 1s in the list of bits, b_i, after the i^{th} increment as the potential function mapping any state of the list of bits to potential ≥ 0.

The initial state (counter=0) has no 1s in its binary representation so $\phi = 0$: this meets the requirement that the potential of the initial state is zero.

Example of the Potential Method [2]

If the i^{th} increment operation, resets r_i bits from 1 to 0, the total actual cost is at most $r_i + 1$: from the least significant bit, we walk the string of bits either setting a 0 to a 1 and terminating, or setting a 1 to a 0 and rippling to the next bit.

The difference in potential before and after the increment is:

$\phi(d_i) - \phi(d_{i-1}) \leq (b_{i-1} - r_i + 1) - b_{i-1} = 1 - r_i$

The amortised cost is (actual + ϕ change) = $(r_i + 1) + (1 - r_i) = 2 \in O(1)$.

The total amortised cost for any sequence is an upper bound for the actual costs. All checks pass so n INCREMENT operations have amortised O(n) cost: O(1) each.

Abstract Data Types (ADTs)

We used the acronym ADT (three times) in Algorithms 1 but have yet to properly define it.

An **abstract data type** is to data structures what a Java Interface is to an algorithm: a list of the operations that must be supported (names, inputs/outputs, semantics), but without a specific implementation.

We have seen some examples already: a stack is an ADT and our implementation using an array is a data structure that implements the interface.

One ADT can extend another, adding further operations.

Binomial Heaps

Binomial Heaps implement the *Mergeable* Priority Queue ADT:

- CREATE(): creates a new, empty Binomial Heap.
- INSERT(bh, (k,p)): insert key/payload into a Binomial Heap.
- PEEK-MIN(bh): returns without removing the min key and its payload.
- EXTRACT-MIN(bh): returns and removes the min key and its payload.
- DECREASE-KEY(bh, ptr_k, nk): decreases key k (in node ptr_k) to nk.
- DESTRUCTIVE-UNION(bh1, bh2): merges two Binomial Heaps.
- COUNT(bh): returns the number of keys present.

DELETE(bh, ptr_k) = {DECREASE-KEY(bh, ptr_k, -∞); EXTRACT-MIN(bh);}

Binomial Heaps vs ordinary Heaps

The heaps we saw in Algorithms 1 perform all these operations in O(lg n) time or better *except for* DESTRUCTIVE-UNION.

The best implementation of DESTRUCTIVE-UNION on ordinary heaps would be to copy the two arrays into a single, larger array and call FULL-HEAPIFY. This would cost $\Theta(n_1 + n_2)$ when two heaps with those sizes are merged.

Binomial Heaps can perform DESTRUCTIVE-UNION in $O(\lg (n_1+n_2))$ time.

Binomial Heaps (like Heaps) do not provide a SEARCH operation. DECREASE-KEY and DELETE require the caller to be able to provide a pointer to a node.

Binomial Trees [1]

A Binomial Heap is a collection of Binomial Trees.

In a Binomial Tree, each node keeps its children in a strictly ordered list: these are not *binary* trees.

A Binomial Tree, B_k, is formed by linking two B_{k-1} trees together such that the root of one is the leftmost child of the other. B_0 is a single node.

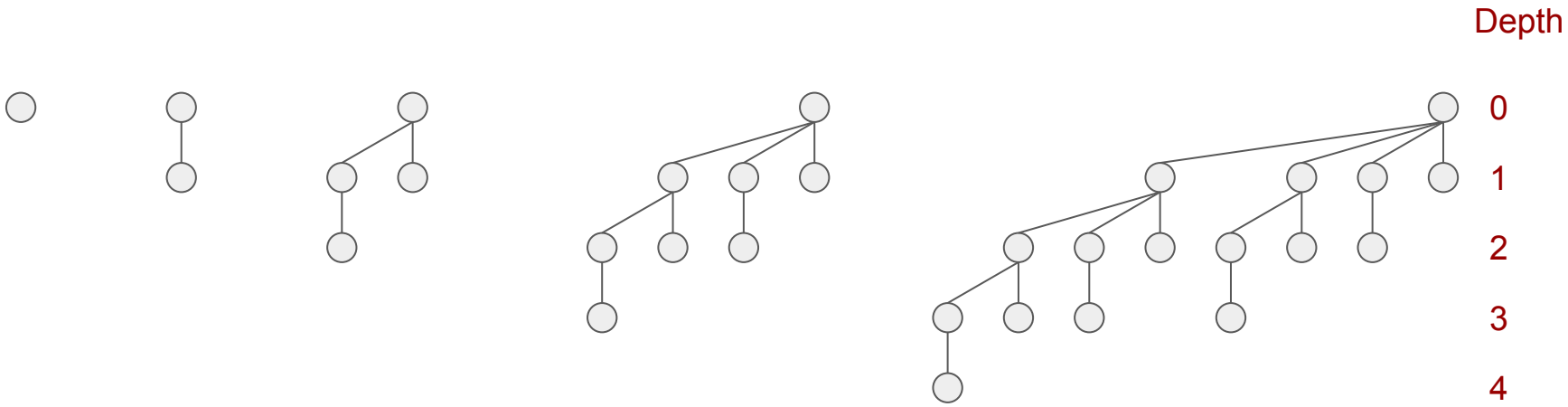
Binomial Trees [2]

These characteristics follow from the recursive definition of Binomial Tree, B_k:

1. There are 2^k nodes in the tree (note that these are not *binary* trees!)
2. The height of the tree is k
3. There are exactly ${}^kC_i$ nodes at depth i, for i = 0, 1, .. k
4. The root has degree k, which is greater than that of any other node
5. The children of the root are ordered: k-1, k-2, .. 0 and child i is the root of a subtree B_i obeying these defining characteristics.

All can trivially be proven by induction.
The maximum degree of any node is lg n (follows from 1 & 4).

Binomial Trees [3]



B_0 B_1 B_2 B_3

B_4 : a Binomial Tree with depth 4

Binomial Heaps

We build a Binomial Heap, H, out of Binomial Trees, as follows.

- Each Binomial Tree in H obeys the min-heap property: each node's key is greater than or equal to that of its parent.
- For any non-negative integer k, there is at most one Binomial Tree in H with root node having degree k.

Notice this means that the overall minimum key must be one of the roots of the Binomial Trees.

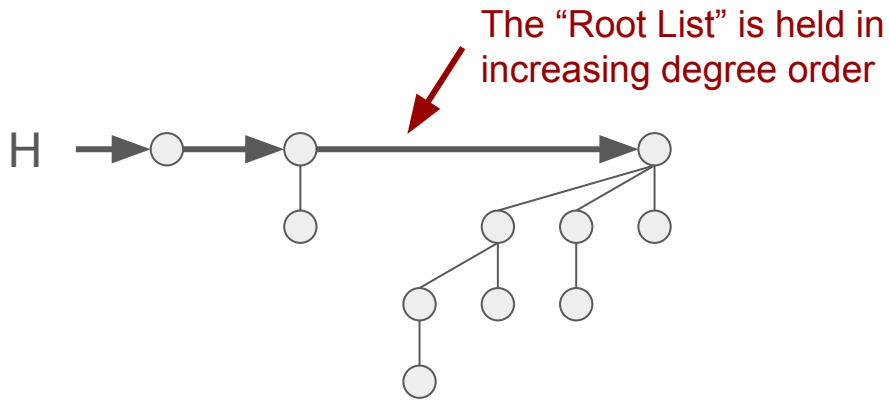
An n-node Binomial Heap contains at most $\lfloor \lg n \rfloor + 1$ Binomial Trees.

Binary Structure

Because the Binomial Tree B_i has 2^i nodes, it follows that a Binomial Heap with n nodes must contain trees B_i corresponding to the 1s in the binary representation of n.

$11 = 8 + 2 + 1 = 2^3 + 2^1 + 2^0$

$11 = 1011_2$

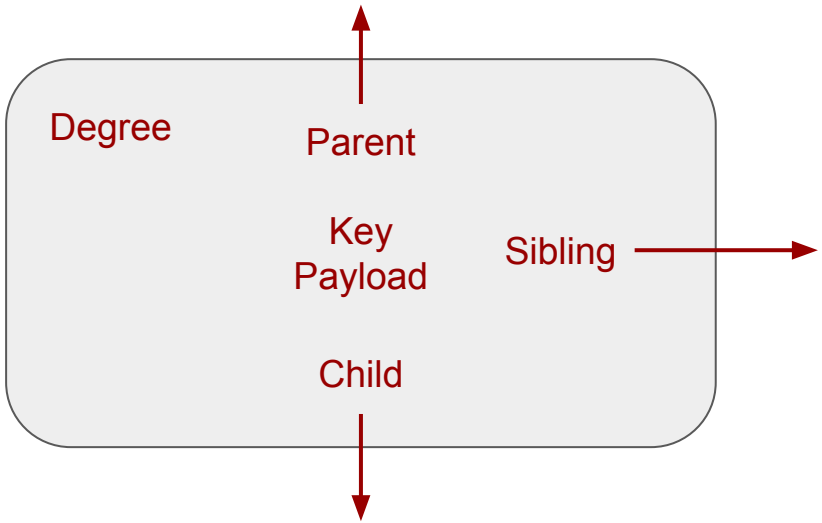


We repurpose the sibling pointers of Binomial Tree root nodes to construct the root list.

Binomial Heap Data Structure [1]

To represent this structure, we need six attributes in each node:

1. Key
2. Payload
3. Next sibling pointer
4. Parent pointer
5. Child pointer (to ONE child)
6. Degree (number of children)



💡 Duplicate keys are permitted.
⚠️ Degree is the number of immediate children, not the number of descendants!

Binomial Heap Data Structure [2]

A reference to the root list of a Binomial Heap is a pointer to the root of the first (lowest degree) Binomial Tree in the root list.

It is permitted (indeed, *required*) to keep pointers to nodes within the heap structure.

```
BH-CREATE()
    return NIL
```

Create is clearly O(1).

BH-PEEK-MIN(bh)

The minimum key has to be one of the roots.

We perform a sequential scan through the root list to find the minimum.

The root list contains at most $\lfloor \lg n \rfloor + 1$ Binomial Trees so this is $O(\lfloor \lg n \rfloor)$.

Note that is BH-PEEK-MIN: it does not BH-EXTRACT-MIN!

BH-DESTRUCTIVE-UNION(bh1, bh2) [1]

First consider the task of merging two Binomial Trees of the *same degree*.

BH-MERGE(bt1, bt2) makes bt2 become the first child of bt1 (increasing bt1.degree in the process). This is achieved by setting bt2.sibling = bt1.child and then bt1.child = bt2, and setting bt2.parent = bt1.

This is O(1) and maintains the order of the child list (characteristic #5 of Binomial Heaps): descending order of degree.

BH-DESTRUCTIVE-UNION(bh1, bh2) [2]

Now we can merge two Binomial Heaps.

bh1 and bh2 each has a root list that is sorted by increasing order of degree. We merge these in order, using BH-MERGE when we encounter two degrees of the same degree (smaller key remains in the root list). This ensures that the resulting Binomial Heap has at most one Binomial Tree of each degree and preserves the property that the root list contains at most ⌊lg n⌋ + 1 Binomial Trees.

Because BT-MERGE is O(1), the running time of the operation to merge the two root lists is O(⌊lg n1⌋ + ⌊lg n2⌋) and this is O(⌊lg n⌋), where n is the total number of nodes in the merged Binomial Heap.

BH-INSERT(bh, (key, payload))

The process to insert one new (key, payload) pair is to:

1. Create a new node, n, containing the (key, payload).
   n.child = NIL, n.parent = NIL, n.sibling = NIL, n.degree = 0.
2. A pointer to this node, p, is itself a Binomial Heap so we can return the result of a call to BH-DESTRUCTIVE-UNION(bh, p).

The running time is O(lg n), dominated by the BH-DESTRUCTIVE-UNION.

BH-EXTRACT-MIN(bh)

This is also straightforward!

1. Cut the Binomial Tree containing the old minimum out of the root list
   a. Use BH-PEEK-MIN to find the minimum if you don't have a pointer to it already.
2. Reverse the list of the old minimum's child list
3. BH-DESTRUCTIVE-UNION the (reversed) child list and the root list

All three steps can be achieved in O(lg n) time since that dominates both the length of the root list and the largest degree (child list length) of any node.

Note that we do not need to find the new minimum because the BH-PEEK-MIN operations searches for it each time.

BH-DECREASE-KEY(bh, ptr_k, nk)

ptr_k is a pointer to the node containing the key we wish to decrease.

Remember that this node is a node in a Binomial Tree, which is min-heap ordered!

We decrease the key using the same method as on a Min-Heap, in O(lg n) time:

1. Decrease the key to nk (it's an error if nk > ptr_k.key)
2. If ptr_k.parent != NIL, access the parent and swap the keys (and payloads) if the new child key compares as smaller in the key sort order
3. Recurse up the tree until either the bubbling stops or we attempt to go to root's parent (identified by parent = NIL). Max height is O(lg n), hence cost.

Fibonacci Heaps

Fibonacci Heaps implement the *Mergeable* Priority Queue ADT:

- CREATE(): creates a new, empty Fibonacci Heap. O(1)
- INSERT(fh, (k,p)): insert key/payload into a Fibonacci Heap. O(1) amortised
- PEEK-MIN(fh): returns without removing the min key and its payload. O(1)
- EXTRACT-MIN(fh): returns and removes the min key and its payload. O(lg n)
- DECREASE-KEY(fh, ptr_k, nk): decreases key k to nk. O(1) amortised
- DESTRUCTIVE-UNION(fh1, fh2): merges two Fibonacci Heaps. O(1) amortised
- COUNT(fh): returns the number of keys present. O(1)

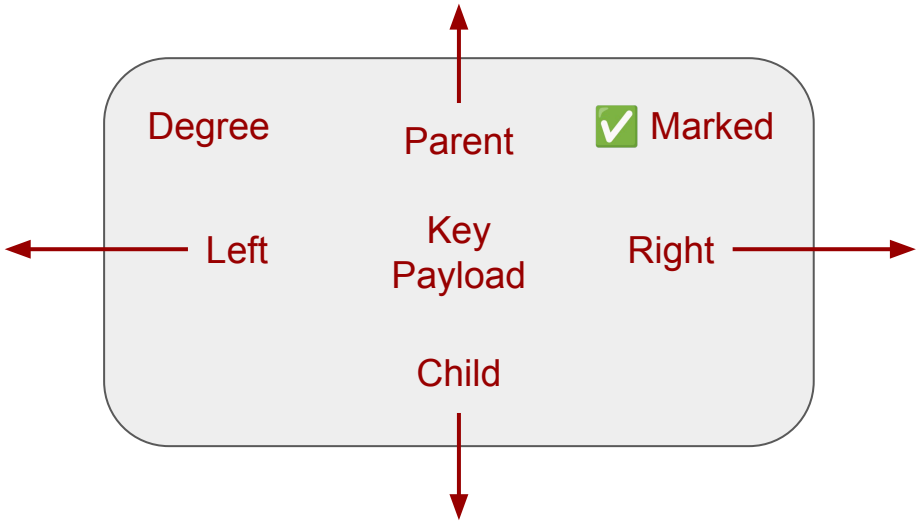
The low costs are what make Fibonacci Heaps special. Let's see how it's done!

💡 DELETE(fh, ptr_k) is DECREASE-KEY(fh, ptr_k, -∞); EXTRACT-MIN(fh). O(lg n) amortised
💡 INCREASE-KEY(fh, ptr_k, nk) is DELETE(fh, ptr_k); INSERT(fh,(ptr_k.key,ptr_k.payload)). O(lg n) amortised

Fibonacci Heap Data Structure [1]

Fibonacci Heap nodes have eight attributes:

1. Key
2. Payload
3. Left sibling pointer
4. Right sibling pointer
5. Parent pointer
6. Child pointer (to ONE child)
7. Degree (number of children)
8. Marked flag (a boolean)



💡 Duplicate keys are permitted.
⚠️ Degree is the number of immediate children, not the number of descendants!
💡 Marked: has this node lost a child since it became a child of its current parent?

Fibonacci Heap Data Structure [2]

A reference to the root of a Fibonacci Heap is a 2-tuple: fh = (r, n).

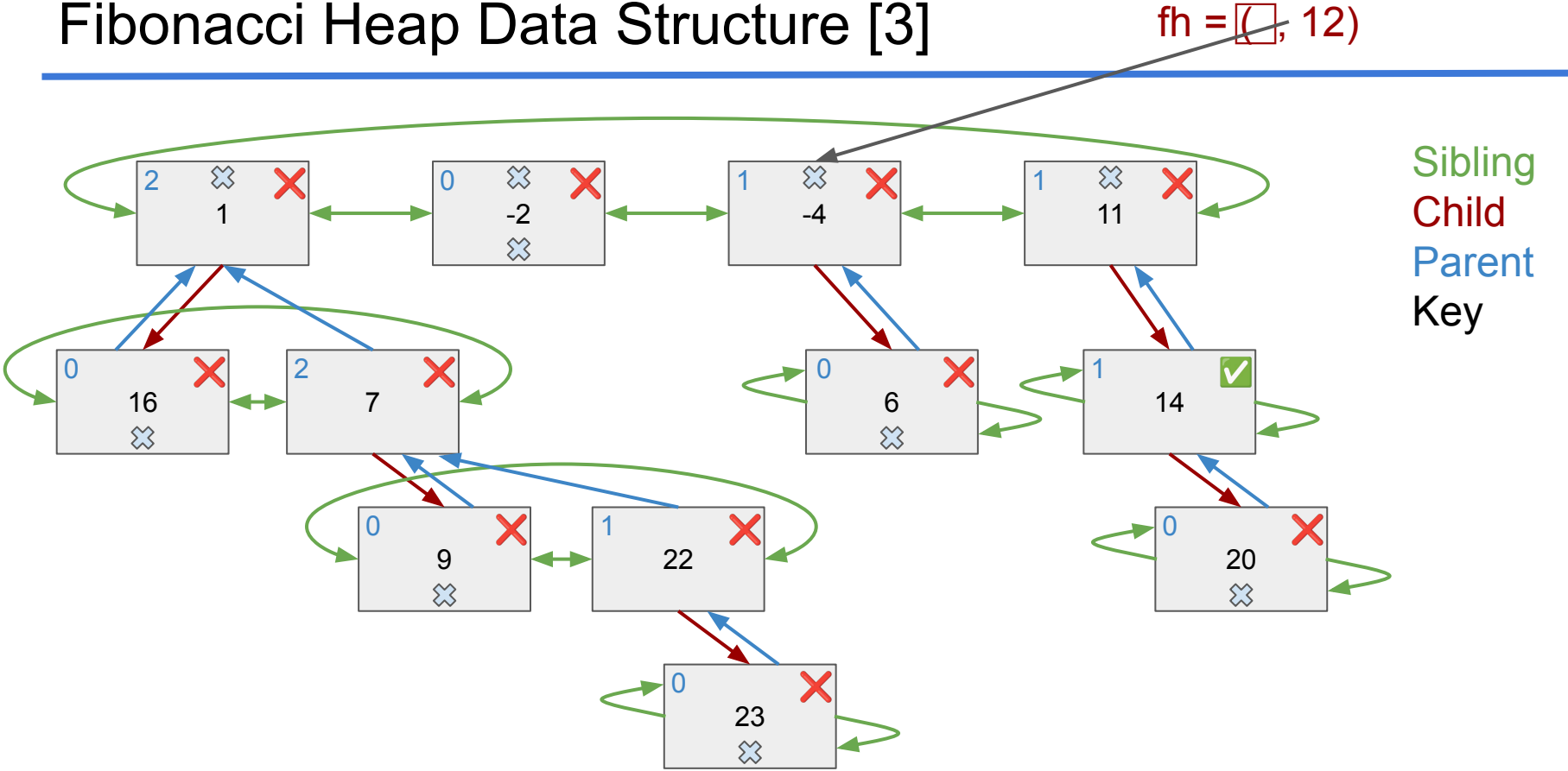
- r is a pointer to the node containing a current minimum key
- n is the number of keys currently present in the Fibonacci Heap

It is permitted (indeed, *required*) to keep pointers to nodes within the heap structure.

```
FH-CREATE()
    return (NIL, 0)
```

Create is clearly O(1).

Fibonacci Heap Data Structure [3]



Fibonacci Heap Data Structure [4]

A collection of binomial min-heaps, held unordered in a doubly linked cyclic list.

The children of every node are held in unordered doubly linked cyclic lists.

Nodes in the root list are never marked.

If a node's key is decreased and becomes smaller than the parent's key then it violates the heap property and cannot remain in its current place in the heap:

- Move it into the root list
- Mark the parent (unless the parent is in the root list) but if the parent was already marked, move the parent to root list and recurse on its parent.

New FibHeapNode(k, p)

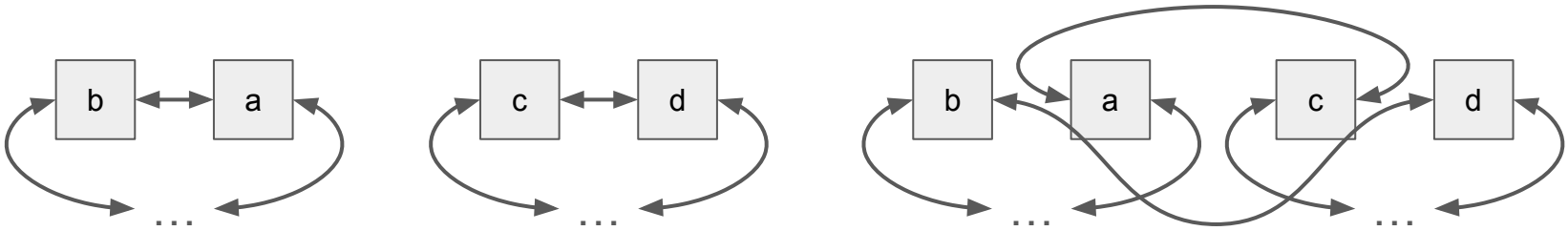
We initialise the 8 fields to create a valid 1-item Fibonacci Heap:

- Key = k
- Payload = p
- Left = <pointer to itself>
- Right = <pointer to itself>
- Parent = NIL
- Child = NIL
- Marked = false
- Degree = 0

FH-DESTRUCTIVE-UNION(fh1, fh2) DLL-SPLICE(a, b, c, d)

```
1   let (p1, n1) = fh1, (p2, n2) = fh2
2   if (p1 == NIL) return fh2
3   if (p2 == NIL) return fh1
4   DLL-SPLICE(p1, p1.left, p2, p2.right)
5   if (p1.key ≤ p2.key) return (p1, n1+n2)
6   return (p2, n1+n2)
```

```
1   a.left = c
2   c.right = a
3   b.right = d
4   d.left = b
```



FH-INSERT(fh, (k, p))

```
1   let fh2 = new FibHeapNode(k, p)
2   return FH-DESTRUCTIVE-UNION(fh, fh2)
```

Notice that this *does* handle the case where fh is an empty Fibonacci Heap (see line 2 of FH-DESTRUCTIVE-UNION).

Notice that this does not put the new key into the correct place in a binomial heap structure. Instead, it “dumps” the new key into the root list.

FH-PEEK-MIN(fh)

```
1   let (p, _) = fh
2   if p == NIL
3       return NIL
4   else
5       return (p.key, p.payload)
```

FH-COUNT(fh)

```
1   let (_, n) = fh
2   return n
```

FH-DECREASE-KEY(fh, ptr_k, nk)

```
1   if (ptr_k.key < nk) error "New key is not smaller!"
2   ptr_k.key = nk; ptr_k_orig = ptr_k
3   if ptr_k.parent != NIL && ptr_k.key < ptr_k.parent.key
4       do if (!ptr_k.parent.marked)
5               CHOP-OUT(fh, ptr_k); break
6           else CHOP-OUT(fh, ptr_k); ptr_k = ptr_k.parent
7       while ptr_k.parent != NIL
8   if (fh.p.key > nk) fh.p = ptr_k_orig
```

Private helper function CHOP-OUT(fh, ptr_k)

```
1   if (ptr_k.parent.degree == 1) ptr_k.parent.child = NIL
2   else if (ptr_k.parent.child == ptr_k)
3       ptr_k.parent.child = ptr_k.left
4   ptr_k.parent.degree = ptr_k.parent.degree - 1
5   if ptr_k.parent.parent != NIL ptr_k.parent.marked = true
6   ptr_k.left.right = ptr.right; ptr.right.left = ptr.left
7   ptr_k.parent = NIL
8   ptr_k.left = ptr_k.right = ptr_k
9   ptr_k.marked = false
10  DLL-SPLICE(fh.p, fh.p.left, ptr_k, ptr_k.right)
```

Only child of its parent

The child that the parent pointed to

Parent has one fewer children

Parent has lost a child

Cut out of parent's child list

Prepare this node to enter the root list

Splice into the root list

⚠️ Line 5 is careful to avoid marking nodes in the root list!

FH-EXTRACT-MIN(fh) [1]

This is where the magic happens.

Let (p, n) = fh.

- If n==1 then this is the last node in the Fibonacci Heap so return (NIL, 0).

The new minimum key has to be one of the children of the old minimum, or one of the other keys in the root list. We begin by dropping the current minimum's children into the root list:

- If p.child != NIL then set v.parent=NIL and v.marked=false for all nodes, v, in the p.child list; and then call DLL-SPLICE(p, p.left, p.child, p.child.right)

FH-EXTRACT-MIN(fh) [2]

Now we can cut the old minimum out of the root list:

- p.left.right = p.right
- p.right.left = p.left
- p = p.left
- n = n - 1

We need to walk around the root list looking for the new minimum key. Because each entry in the root list is a min-heap, we know the overall minimum cannot be deep into any of the heaps, but it could be any of the roots as there is no ordering between them.

FH-EXTRACT-MIN(fh) [3]

- let start = p, t = p.right
- while t != start
  - if (t.key < p.key) then p = t
  - t = t.next

p and n are now set correctly so we could say that we're done and return (p, n). Although that would implement the operations correctly, it would not achieve the asymptotic costs we claimed.

It turns out that all we need to do is clean up the heap at this point.

FH-EXTRACT-MIN(fh) [4]

We are going to need an array with $D(n) + 1 = \lfloor \log_{\varphi} n \rfloor + 1$ elements, initialised to NIL ('n' is the node count *after* removing the old minimum). $\varphi = (1+\sqrt{5})/2$. D(n) is the maximum degree of any node in a Fibonacci Heap with n nodes.

It will be convenient to index this array from 0, not from 1.

- let A = new Array[$\lfloor \log_{\varphi} n \rfloor + 1$]
- for i = 0 to A.length-1
  - A[i] = NIL

While we are walking around the root list, we combine heaps with the same degree, using the array to remember where we last saw a heap with each degree.

⚠️ The array length must be 1 + max_degree. We will prove this value later.

FH-EXTRACT-MIN(fh) [5]

When considering node ‘t’ in the root list we…

- if A[t.degree] == NIL
  - A[t.degree] = t
- else
  - old_start = start
  - old_start_right = start.right
  - merge(t, A[t.degree])
  - if (old_start.parent != NIL) start = old_start_right

What does merge do? Well…

FH-EXTRACT-MIN(fh) [6]

MERGE(a, b):

- if a.key >= b.key
  - A[b.degree] = NIL
  - b.degree = b.degree + 1
  - a.left.right = a.right; a.right.left = a.left
  - a.left = a.right = a
  - if (b.degree == 1)
    - b.child = a
  - else
    - DLL-SPLICE(b.child, b.child.left, a, a.right)
  - if (A[b.degree] != NIL) MERGE(b, A[b.degree])
  - else A[b.degree] = b

FH-EXTRACT-MIN(fh) [7]

- else
  - A[a.degree] = NIL
  - a.degree = a.degree + 1
  - b.left.right = b.right; b.right.left = b.left
  - b.left = b.right = b
  - if (a.degree == 1)
    - a.child = b
  - else
    - DLL-SPLICE(a.child, a.child.left, b, b.right)
  - if (A[a.degree] != NIL) MERGE(a, A[a.degree])
  - else A[a.degree] = a

Intuition behind Fibonacci Heaps

Before we prove the asymptotic running times for the key operations, let's get an intuition for why they're so cheap.

To get started, let's consider only the Priority Queue ADT operations operations: CREATE, INSERT, PEEK-MIN, EXTRACT-MIN.

It's obvious that our implementations of CREATE and PEEK-MIN are O(1): the code only performs a fixed number of fixed-time operations. No loops in either case.

Intuition behind Fibonacci Heap INSERT

It's also obvious that INSERT does a constant amount of work when it is called but we might consider that it is only doing part of the job: it is putting the item into the data structure but not into the correct place in the data structure.

There are two problems with only doing half a job:

1. You have to come back and do the rest of the work later; and
2. There is a price to pay for putting "spurious" items into the root list: it costs more time to run extract-min.

When we consider EXTRACT-MIN, we see that neither is asymptotically important.

Intuition behind Fibonacci Heap EXTRACT-MIN [1]

EXTRACT-MIN:

- Drops the old minimum's children into the root list – O(1)
  - Because the lists are cyclic, we do not need to walk to the start/end of either to append lists
  - Because the lists are unordered, it does not matter where we join the two lists together
  - We do have to set the parent pointers to NIL and the marked flags to false: come back to this!
- Cuts the old minimum out of the root list – O(1)
  - We do not need to search for the node containing the minimum (we have a pointer to it)
  - The root list is doubly linked so we can delete in O(1) time
- Walks around the root list looking for the new minimum – O(r), r: len root list
  - It's O(1) work per item to compare it to the running minimum
  - It's O(1) per item to set parent=NIL and marked=false so we can absorb the earlier costs!

Intuition behind Fibonacci Heap EXTRACT-MIN [2]

EXTRACT-MIN is doing O(r) work but we only charge the customer O(lg n), so there is a shortfall to explain with an amortised analysis.

Suppose, for every O(1) Insert also put O(1) money into a bank account, and that money can be used to pay for work that is done later. O(1) money can be used to pay for any constant amount of work.

We need to spend O(r - lg n) money from the bank account to balance the books for EXTRACT-MIN.

We can only spend the money once so what about the second EXTRACT-MIN?

Intuition behind Fibonacci Heap EXTRACT-MIN [3]

The reason that r > k lg n is because there is “junk” in the root list that shouldn’t be there: all the keys we inserted cheekily in the wrong place!

It’s OK to use the bank account to pay for scanning through those keys once to find the new minimum but we have to make sure those keys do not need to be scanned the next time we run extract-min.

This is exactly achieved by combining of roots of the same degree: trees begin as single nodes and are combined into 2s, 4s, 8s, 16s, etc. so if we have n keys then we have at most 1 + lg n min-heaps in the root list, i.e. the root list shrunk from r to ~lg n, and r - k lg n is exactly the correct amount of money to balance the books!

⚠️ This informal intuition does not account for DECREASE-KEY. (We use “lg n”, not “$\log_\varphi$ n”.)

Intuition behind Fibonacci Heap DECREASE-KEY [1]

To account for decrease key, we note that it takes O(1) time to replace the key, compare it to the parent (thanks to the parent pointer) and, if necessary to cut it out of the parent's child list and splice it into the root list (thanks to both being doubly linked). Even if we cut the parent out of its sibling list as well, that's still only O(1) work.

The problem comes when the parent's parent is already marked, and its parent, and so forth – we do an amount of work that is proportional to the height of that min-heap and this is not O(1).

Same trick! Actual cost is O(h); customer pays O(1); bank funds O(h - 1). How?

Intuition behind Fibonacci Heap DECREASE-KEY [2]

Every time we remove a child and mark its parent node (the parent *not* already being marked), we put 2x O(1) amounts of money into our bank account. DECREASE-KEY costs O(1) so this does not change its asymptotic cost.

When that marked node loses another child, one of those O(1) amounts of money can pay for its removal and splicing into the root list. If its parent is also marked then it, too, has 2x O(1) amounts of money in the bank, one of which can be used to pay for it to fall into the root list. This continues and since each level in the min-heap pays for its own O(1) work, the total paid is the O(h - 1) shortfall.

BUT… don't all those items in the root list add to the cost of EXTRACT-MIN?!

Intuition behind Fibonacci Heap DECREASE-KEY [3]

Yes, all those decreased-keys in the root list do increase the cost of EXTRACT-MIN but we have that second O(1) amount of money that we haven't spent yet.

The second O(1) amount of money is what pays for the additional costs of scanning the root list during the next EXTRACT-MIN operation. Since that recombines trees leaving O(lg n) items in the root list, it only needs to be spent once.

The decreased keys and marked parents that fell into the root list have a corresponding O(1) amount of money in the bank, mirroring the O(1) money contributed by INSERT for new keys, and paying for their clean-up.

Formal analysis: amortised analysis

We need a potential function that is non-negative, zero for the empty data structure, and sufficient to “pay for” the expensive steps in the algorithms such that we can claim amortised costs:

- CREATE(), COUNT(fh), PEEK-MIN(fh) : O(1)
- INSERT(fh, (k,p)): O(1) amortised
- EXTRACT-MIN(fh): O(lg n) amortised
- DECREASE-KEY(fh, ptr_k, nk): O(1) amortised
- DESTRUCTIVE-UNION(fh1, fh2): O(1) amortised

Coming up with a potential function

- It often helps to compare the ideal state of your data structure with the actual state, which might be “damaged” by the cheeky operations that did something cheaply but imperfectly.
  - The potential needs to be (at least) what it would cost to fix the damage.
- Consider the cost actual cost of the operations you need to perform and what you want to charge for them, since the difference is what you need the potential to cover. Which other operations lead to these operations being more expensive than they might be, and could you get them to pay for the clean-up in advance?

Potential for a Fibonacci Heap

It turns out that a good potential function is $\phi = r + 2m$, where r is the number of items in the root list and m is the number of marked nodes.

Check: $\phi = 0$ for an empty Fibonacci heap?

- Yes, because CREATE returns (NIL, 0): the root list is empty and there are no marked nodes (no nodes at all).

We proceed to check the other operations…

Amortised analysis of FH-INSERT(fh, (k,p))

Start with any Fibonacci Heap, fh, (including the empty case), with potential $\phi_1 = r + 2m$.

FH-INSERT adds (via a call to the destructive union operation) one item to the root list. The new node is never marked. Existing marked nodes remain marked; existing unmarked nodes remain unmarked. **r increases by 1; m is unchanged.**

The potential after insert, $\phi_2 = (r+1) + 2m = \phi_1 + 1$.

The amortised cost of insert is {cost of immediate work} + {change in potential} $= k + (\phi_2 - \phi_1) = k + 1 \in O(1)$ ✅ this is as claimed.

Amortised analysis of FH-DESTRUCTIVE-UNION(fh1, fh2)

Start with Fibonacci Heaps, fh1 and fh2, with potentials
$\phi_1 = r_1 + 2m_1$ and $\phi_2 = r_2 + 2m_2$.

FH-DESTRUCTIVE-UNION splices the roots lists together, adds n1 and n2, and compares the min keys to return a value. No marked flags are changed. The potential of the combined Fibonacci Heap is $\phi_3 = (r_1 + r_2) + 2(m_1 + m_2)$.

The amortised cost of destructive-union is {cost of immediate work} + {change in potential}
$= k + (\phi_3 - (\phi_1 + \phi_2)) = k \in O(1)$ ✅ this is as claimed.

⚠️ The input heaps cannot be used after the call to FH-DESTRUCTIVE-UNION so they do not need their potential to "repair" them: that potential can be repurposed to repair the damage on the combined heap!

Amortised analysis of FH-DECREASE-KEY(fh, ptr_k, nk) [1]

Start with any Fibonacci Heap, fh, with potential $\phi_1 = r + 2m$.

After decreasing a key, we might be in a variety of states.

If we decreased the key but it remains greater than its parent, then no changes are made to the root list, nor to any marked flags.

The potential is unchanged so the amortised cost is the actual cost in this case, and is clearly $\in O(1)$.

Amortised analysis of FH-DECREASE-KEY(fh, ptr_k, nk) [2]

If the ptr_k.key becomes smaller than its parent, but the parent is not marked then the decreased key falls into the root list and the parent becomes marked.

If the node pointed to by ptr_k was marked before, it becomes unmarked when it falls into the root list.

If !ptr_k.marked beforehand, then the change in potential is (r+1)-r + 2(m+1-m) = 3

If ptr_k.marked beforehand, then the change in potential is (r+1)-r + 2(m-m) = 1

In both cases, the work done immediately is constant and the contribution to the potential is constant so the amortised cost is $\in O(1)$.

Amortised analysis of FH-DECREASE-KEY(fh, ptr_k, nk) [3]

If the ptr_k.key becomes smaller than its parent, and the parent is already marked then the parent falls into the root list (and becomes unmarked), and its parent might do the same. Suppose that, in total, 'a' ancestor nodes fall into the root list.

The total work done is $k_1 + a.k_2$.

The potential afterwards is (r+1+a) + 2(m-a) so the change is 1+a-2a = 1-a.

The sum is $k_1 + a.k_2 + (1-a) = k_3$ (choosing $k_2 = 1$) so the amortised cost is $\in$ O(1).

In all cases, FH-DECREASE-KEY has amortised cost O(1) ✅ this is as claimed.

⚠️ We should also consider the two cases of ptr_k being marked and not before the decrease-key.

⚠️ There is really another case split here! The final ancestor might be in the root list (so does not get marked) or not in the root list (does get marked). This slide shows the second (worst case).

Amortised analysis of FH-EXTRACT-MIN(fh) [1]

- We start with r nodes in the root list and m marked nodes.
- We add the old minimum's children into the root list: at most D(n) children where D(n) is the maximum degree of a node in a Fibonacci Heap with n nodes.
- We remove one item from the root list (the minimum).
- When we scan the root list, looking for the new minimum, there are at most r + D(n) - 1 nodes in the root list.
- The cost of finding the new minimum $\in O(r + D(n))$

Amortised analysis of FH-EXTRACT-MIN(fh) [2]

- When we combine nodes of the same degree, we loop through each node in the root list:
  - If we do not remove a node from the root list (now or through later merges with it), we do constant work on it because we put it in the array and do not remove it.
  - If we do remove a node from the root list, we only compare it to one other before doing so: one is found in O(1) time using the array, and we can only remove a node once.
- The total work to merge nodes is O(r + D(n))
- This leaves at most D(n)+1 nodes in the root list because, if there were more, there would be two items in the same array position and we would have merged them. (Remember the array length was D(n)+1.)

Amortised analysis of FH-EXTRACT-MIN(fh) [3]

Now we can analyse the change in potential. Note that no nodes changed their marked flag during the merges.

Potential before = r + 2m

Potential afterwards (worst case) = (D(n) + 1) + 2m
Change in potential = (D(n) + 1) + 2m - (r + 2m) = D(n) + 1 - r
Amortised cost = (r + D(n)) + (D(n) + 1 - r) $\in$ O(D(n))

⚠️ Worst case because fewer items in the root list would release more potential to pay for work. Also, if any of the old minimum's children were marked, unmarking them would release potential.

Amortised analysis of FH-EXTRACT-MIN(fh) [4]

To show that the cost of EXTRACT-MIN is O(lg n), we need to show that D(n) is bounded from above by k.lg n.

We will show that $D(n) \leq \lfloor \log_{\varphi} n \rfloor$ where $\varphi = (1+\sqrt{5})/2$ is the golden ratio.

For any node x, define size(x) to be the total number of nodes in the heap rooted at node x, including node x itself. (Node x need not be in the root list.)

Amortised analysis of FH-EXTRACT-MIN(fh) [5]

Lemma 1: let x be a node in Fibonacci Heap; if x has degree k then let its children be c_1, c_2, .. c_k in the order they were added as children of x; we have that c_1.degree ≥ 0 and c_i.degree ≥ i-2 for i=2..k.

Proof

c_1's degree must be at least zero because any node's degree is non-negative.

x and c_i had the same degree when they were merged, and x had i-1 children at that point. Since then, c_i can have lost at most one child (since losing a second would have removed it from x's parentage) so c_i.degree ≥ i-2.

Amortised analysis of FH-EXTRACT-MIN(fh) [6]

Fibonacci numbers, indexed as the 0th, 1st, … are defined by

```
let rec fib(k) = if (k < 2) then k else fib(k-1) + fib(k-2)
```

0, 1, 1, 2, 3, 5, 8, 13, 21, … correspond to k=0, 1, 2, 3, 4, 5, …

Notice that

$fib(k+2) = 1 + \Sigma^{k}_{i=0} fib(i)$

(Trivially proved by induction.)

Amortised analysis of FH-EXTRACT-MIN(fh) [7]

Notice also that the $(k+2)^{th}$ Fibonacci number, $fib(k+2) \geq \varphi^k$.

Proof: by induction.

Base case k=0: $fib(0+2) = 1 = \varphi^0$.
Base case k=1: $fib(1+2) = 2 > \varphi^1 = 1.619\ldots$

Inductive step uses strong induction: assume that $fib(i+2) \geq \varphi^i$ for all i = 2..k-1 and prove $fib(k+2) \geq \varphi^k$ for k ≥ 2.

$$fib(k+2) = fib(k+1) + fib(k) \geq \underbrace{\varphi^{k-1} + \varphi^{k-2}}_{\text{By inductive hypothesis}} = \underbrace{\varphi^{k-2}(\varphi + 1)}_{\text{Because } \varphi \text{ is the positive root of } x^2 = x+1} = \varphi^{k-2}\varphi^2 = \varphi^k$$

Amortised analysis of FH-EXTRACT-MIN(fh) [8]

Lemma 2: let x be any node in a Fibonacci Heap and let k = x.degree; then size(x) ≥ fib(k+2) ≥ φ^k.

Proof

Denote the minimum possible size of any node of degree k as s_k.

$s_0 = 1$ and $s_1 = 2$, and considering a node x with degree k, $s_k \leq$ size(x).

Note that s_k increases monotonically with k (adding children cannot decrease the minimum size).

Amortised analysis of FH-EXTRACT-MIN(fh) [9]

Now consider some node z with degree k and size(z) = s_k (i.e. minimum size).

$s_k \leq$ size(x) so a lower bound on s_k is a lower bound on size(x).

Consider the children c_1, c_2, .. c_k of z in the order they were added.

size(x) $\geq s_k \geq$

1 (for z itself)

+ 1 (for c_1, also a zero-degree node when merged with z, or now a larger child if the original first child was removed)

+ $\sum_{i=2}^{k} s_{ci.degree}$

$\geq 2 + \sum_{i=2}^{k} s_{i-2}$ // using Lemma 1 and monotonicity

Amortised analysis of FH-EXTRACT-MIN(fh) [10]

Next, we show that $s_k \geq fib(k+2)$ for $k \geq 0$, using induction.

Bases cases k = 0 and k = 1 follow immediately from the definitions of s_k and fib(k+2): $s_0 = 1 = fib(0+2)$ and $s_1 = 2 = fib(1+2)$.

Inductive step: k ≥ 2. The induction hypothesis gives us that $s_i \geq fib(i+2)$ for i = 0..k-1 and we seek to prove this property for i = k.

$$
\begin{aligned}
s_k &\geq 2 + \sum_{i=2}^{k} s_{i-2} \\
&\geq 2 + \sum_{i=2}^{k} fib(i) \\
&= 1 + \sum_{i=0}^{k} fib(i) \\
&= fib(k+2) \geq \varphi^k \qquad \text{using the properties of Fibonacci numbers}
\end{aligned}
$$

Amortised analysis of FH-EXTRACT-MIN(fh) [11]

If x is any node in a Fibonacci Heap and has k = x.degree, then we know that $n \geq size(x) \geq \varphi^k$.

Taking logs to base φ, we have that $k \leq \log_\varphi n$.

Since k must be an integer, we have $k \leq \lfloor \log_\varphi n \rfloor$.

Because this is true for any node, we have that the maximum degree of any node in a Fibonacci Heap with n nodes is $D(n) \leq \lfloor \log_\varphi n \rfloor \in O(\lg n)$.

Hence the amortised cost of EXTRACT-MIN $\in O(\lg n)$. ✅ this is as claimed.

Uses for Fibonacci Heaps

Fibonacci Heaps used to be used in the Linux Kernel as the priority queue of processes, waiting to be chosen to run by the Process Scheduler.

It was replaced with a Red-Black tree that, although it has larger asymptotic costs, runs faster on the typical size of problem instance, due to (much) lower constant factors.

Fibonacci Heaps in Dijkstra's Algorithm

We said that in the worst case, Dijkstra's algorithm will call CREATE once, INSERT O(|V|) times, EXTRACT-MIN O(|V|) times, and DECREASE-KEY O(|E|) times.

Priority Queue	**CREATE**	**INSERT**	**EXTRACT-MIN**	**DECREASE-KEY**	**Total**
Sorted Linked List	O(1)	O(|V|)	O(1)	O(|V|)	$O(\|V\|^2 + \|V\|\|E\|)$
Sorted Array	O(|V|)	O(|V|)	O(|V|)	O(|V|)	$O(\|V\|^2 + \|V\|\|E\|)$
Heap	O(1)	O(lg |V|)	O(lg |V|)	O(lg |V|)	O(|V| lg |V| + |E| lg |V|)
Fibonacci Heap	O(1)	O(1) amortised	O(lg |V|) amortised	O(1) amortised	O(|V| lg |V| + |E|) amortised

Are there any better mergeable priority queues?

Actually, there are two!

1. 1996: Gerth Stølting Brodal (Aarhus University, Denmark) invented **Brodal Heaps**, which achieve *actual* O(1) worst case running time for all operations except EXTRACT-MIN, which is O(lg n). Actual means “not amortised”.
2. 2012: Gerth S. Brodal, George Lagogiannis, and Robert E. Tarjan invented **Strict Fibonacci Heaps**, which achieve the same *actual* asymptotic bounds and a simpler set of algorithms.

These are asymptotically optimal (on conventional hardware). Proof: we could do comparison sorts in less than O(n lg n) time with better data structure/algorithms.

Disjoint Sets

The **Disjoint Set ADT** can be implemented in many ways, including by the use of a data structure that is based on an amortised cost analysis.

The Disjoint Set ADT is initialised with a collection of n distinct keys. Each key is placed into its own set. The data structure supports two operations:

1. UNION(s_1, s_2): combine two disjoint sets, s_1 and s_2, into a single set
2. IN-SAME-SET(k_1, k_2): report whether keys k_1 and k_2 are currently in the same set (return true) or different sets (return false)

Disjoint Sets using Doubly Linked Lists

CREATE: for n provided keys, create n linked lists, each of length 1, holding their corresponding key.

UNION(a, b): walk forwards along a's list until you reach the end, and along b's list until you reach the beginning. Change the list pointers to join the end of the 'a' list to the start of the 'b' list.

IN-SAME-SET(k_1, k_2): from the node holding k_1, walk in both directions until you reach a NIL pointer. If a node containing k_2 is found, return true; else return false.

CREATE: O(n) UNION: O(n) IN-SAME-SET: O(n)

Disjoint Sets using Cyclic Doubly Linked Lists

CREATE: for n provided keys, create n cyclic linked lists, each of length 1, holding their corresponding key.

UNION(a, b): splice a's list and b's list together. As the lists are unordered, we can splice at the positions pointed to by a and b (which do not require any searching to find).

IN-SAME-SET(k_1, k_2): from the node holding k_1, walk around until you find k_2 or get back to k_1. If a node containing k_2 is found, return true; else return false.

CREATE: O(n) UNION: O(1) IN-SAME-SET: O(n)

Disjoint Sets using Hash Tables

CREATE: create a hash table and insert the (key, payload) pairs (k_i, i). This represents the starting point that key k_i is in set i.

UNION(a, b): scan through every record in the hash table; if some key maps to payload b, change the payload to a.

IN-SAME-SET(k_1, k_2): return HT-SEARCH(k_1) == HT-SEARCH(k_2)

CREATE: O(n) UNION: O(n) IN-SAME-SET: O(1)

Disjoint Sets with Path Compression & Union by Rank [1]

CREATE: create a tree node for each key. This yields n separate trees. The data stored in each tree node is a pointer to another tree node, initialised to NIL. Each node also contains an integer estimating (upper-bounding) the depth of the subtree rooted at itself, initialised to 0.

CHASE(k): starting from the node for key k, follow the pointers until you reach the root, r, of its tree (where pointer == NIL). Change the pointer of each node you went through to 'r'. This ensures that the next time we CHASE(k), or we chase any descendant of k, we jump straight from k to r. The cost of walking the path from k to r is only paid once. This is called **path compression**.

CREATE: O(n) UNION: ~O(1) amortised IN-SAME-SET: ~O(1) amortised

Disjoint Sets with Path Compression & Union by Rank [2]

UNION(a, b): let r_a = CHASE(a), r_b = CHASE(b). If the estimated depth of r_a is strictly greater than that of r_b, then change r_b's pointer to r_a. If r_b is deeper than r_a then change r_a's pointer to r_b. If both depths are equal, make either point to the other and increment the estimated depth of the root (the one pointed *to*). This is called **union-by-rank**.

IN-SAME-SET(k_1, k_2): return CHASE(a) == CHASE(b). If the roots of the trees containing the two keys are the same then the keys are in the same set.

CREATE: O(n) UNION: ~O(1) amortised IN-SAME-SET: ~O(1) amortised

Kruskal's Algorithm using Disjoint Sets

In the worst case, Kruskal's Algorithm CREATEs a disjoint set representation exactly once, calls UNION |V|-1 times and calls IN-SAME-SET |E| times.

Kruskal also sorts the edges: O(|E| lg |E|) = O(|E| lg |V|) since |E| is at most $|V|^2$.

Disjoint Set	**Sort**	**CREATE**	**UNION**	**IN-SAME-SET**	**Total**
DLLs	O(|E| lg |V|)	O(|V|)	O(|V|)	O(|V|)	$O(\|V\|^2 + \|V\|\|E\|)$
Cyclic DLLs	O(|E| lg |V|)	O(|V|)	O(1)	O(|V|)	O(|V| + |V||E|)
Hash table	O(|E| lg |V|)	O(|V|)	O(|V|)	O(1)	$O(\|V\|^2 + \|E\| \lg \|V\|)$
Trees with PC & UbR	O(|E| lg |V|)	O(|V|)	~O(1) amortised	~O(1) amortised	~O(|V| + |E| lg |V|) amortised

Algorithms 2

Section 4: Geometric Algorithms

Polygons

Polygons are an ordered list of vertices.

Vertices are points (vectors) in some kind of 2D vector space.

We are mostly interested in planar, closed, simple polygons.

Our first problem is to work out whether a point is on the “inside” of a polygon.

Planar Polygons [1]

If the space in which the polygon exists is not planar, it can be tricky or impossible to define “inside” and “outside”.

Example: the Earth’s surface is (roughly) spherical; is Cambridge (assumed to be a point) “inside” the UK mainland (represented as a polygon), or “outside”? Neither label makes sense because the polygon boundary divides two finite areas and we could label either as “inside”.

Note that we cannot say the “smaller” area is “inside”: ask whether a container ship’s position is “inside” the ocean polygon or is on land.

Planar Polygons [2]

A planar space is 2D, flat, and infinite in the “horizontal” and “vertical” directions.

A polygon drawn on a planar surface separates a finite area from an infinite area: we refer to the finite area as “inside” and the infinite area as “outside”.

Closed Polygons

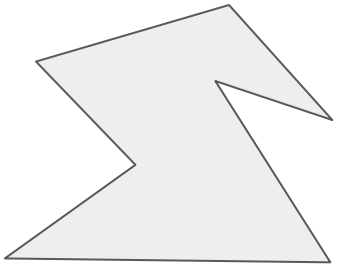
A closed polygon is one where there is an edge from its last vertex back to its first.

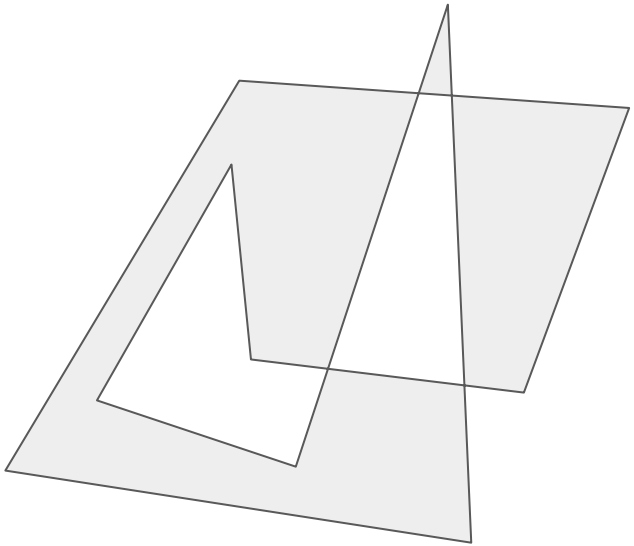
An open polygon does not (necessarily) enclose any area so we cannot define inside and outside.

Example: can you be “inside” the letter O? What about letter C?

Simple Polygons [1]

Simple polygons do not overlap themselves.

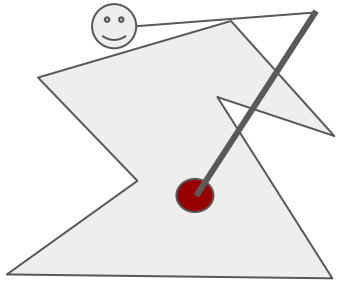


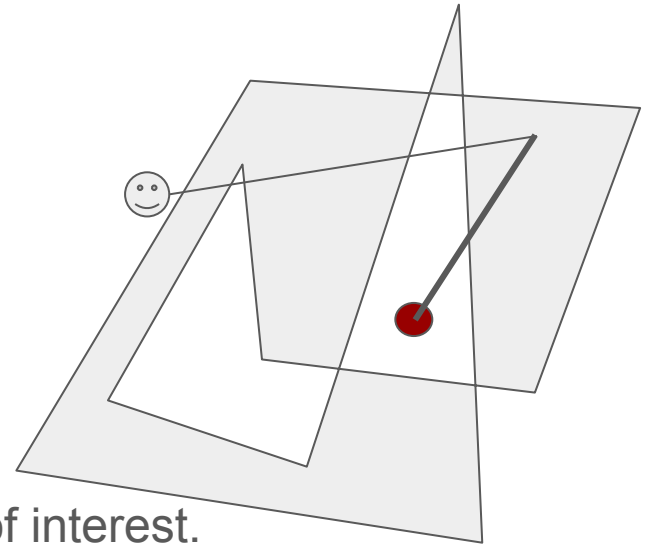


Winding Numbers [1]

How do we define what is “inside” a complex polygon?

One way is with the **winding number**.

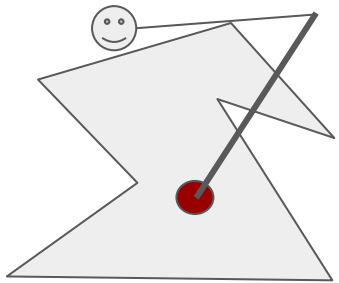


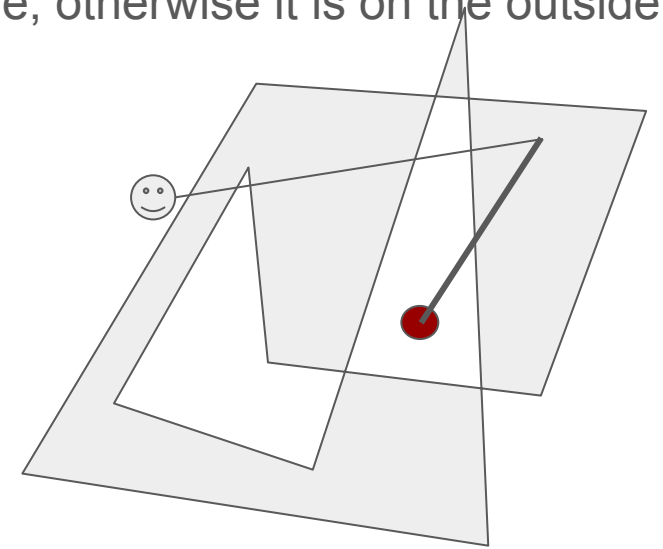


Walk around the perimeter with a piece of string attached to a post at the point of interest.

Winding Numbers [2]

When you get back to the start, if the string is wound around the post an odd number of times, the post is on the inside; otherwise it is on the outside.





Winding Numbers [3]

We can implement this algorithm on a computer:

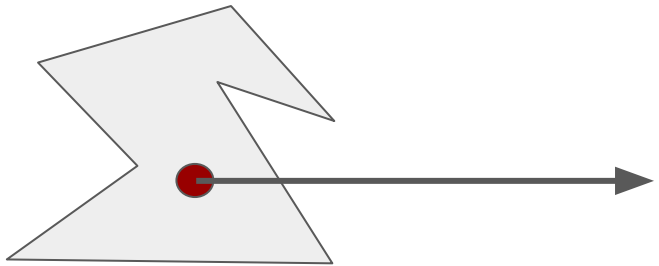
1. calculate angles subtended at the post by the two ends of each edge;
2. sum the angles
3. divide by 2π to get the winding number.

Problems: floating point inaccuracy; slow trigonometric functions.

Inclusion within Simple, Planar, Closed Polygons

Add a semi-line from the point of interest P, in any direction.

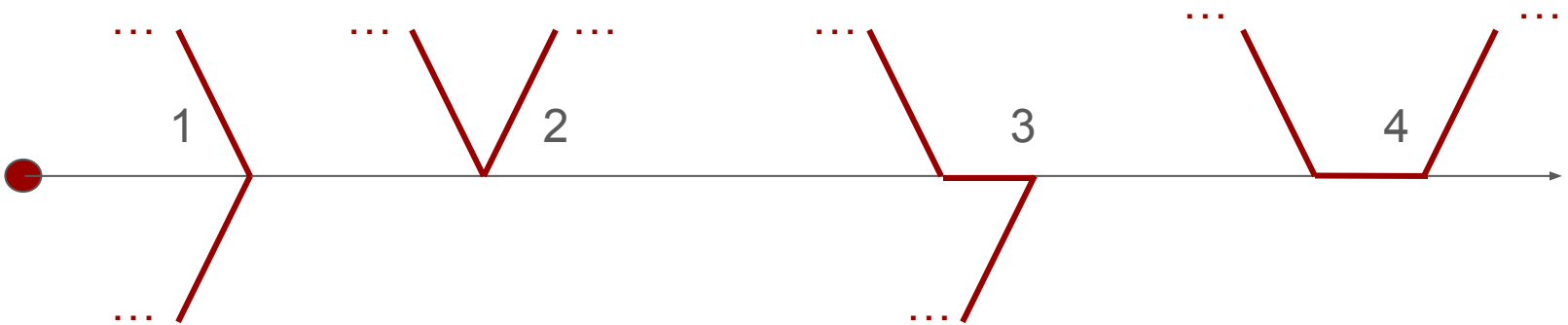
A semi-line is infinite in one direction. Because the coordinates of any vertex are finite values, a point at infinity must be on the “outside”. Because the polygon is simple, planar and closed, each edge separates a a region of “inside” from a region of “outside” so we can count edge crossings.



Awkward cases

If the ray goes through a vertex, we could discard the ray and send one in a different direction; keep retrying until it doesn't hit any vertices.

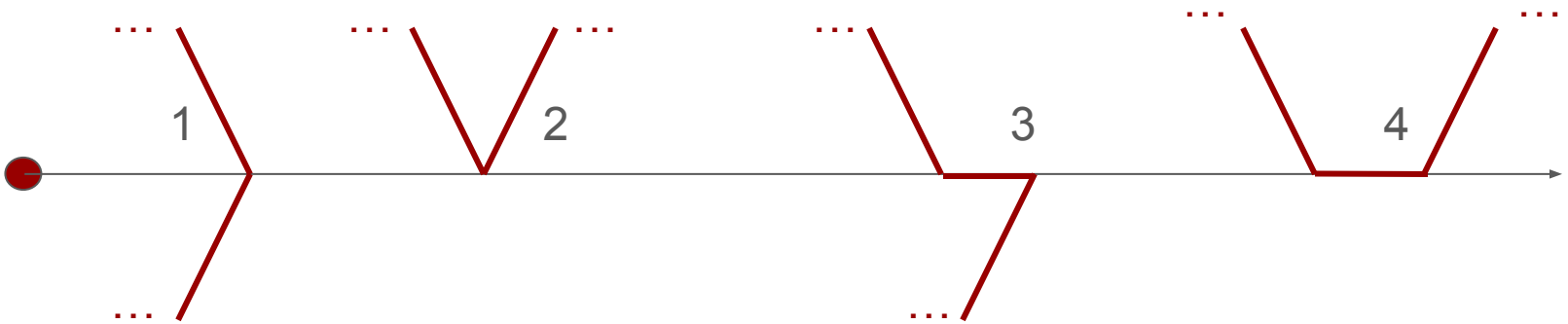
The horizontal ray avoids floating point error in calculations of whether we hit the vertex, were slightly above or were slightly below because (non-NaN) floats are totally ordered.



Handling the Awkward cases

If a vertex is on the ray, look at the neighbouring vertices. If they're on the same side (both above / both below) then the polygon's edge was *not* crossed (case 2); if they are on opposite sides then the edge was crossed (case 1).

If either neighbour is also on the ray, replace it with the next neighbour in the same direction around the polygon boundary.



Line segments

A **line segment** p_1p_2 is a straight line between two points p_1 and p_2. We say that p_1 and p_2 are the **endpoints** and, if the line has a direction then we have a **directed segment** $p_1 \rightarrow p_2$.

These points might be adjacent vertices in a polygon or the test point and a point "at infinity".

Convex combinations

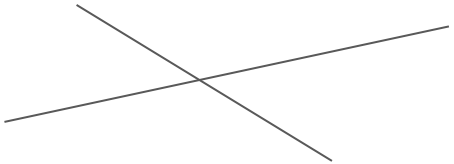
If $p_1 = (x_1,y_1)$ and $p_2 = (x_2,y_2)$, then we say that $p_3 = (x_3,y_3)$ is a **convex combination** of p_1 and p_2 if p_3 is on the line segment between p_1 and p_2 (including the endpoints).

Mathematically, $x_3 = \alpha x_1 + (1-\alpha)\, x_2$ and $y_3 = \alpha y_1 + (1-\alpha)\, y_2$. This is often written as the vector equation $p_3 = \alpha p_1 + (1-\alpha)p_2$. We require $0 \leq \alpha \leq 1$, to place p_3 between p_1 and p_2 inclusive of the endpoints.

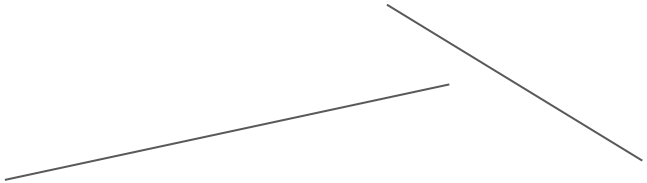
Intersection Determination Problem

Input: two line segments p_1p_2 and p_3p_4.

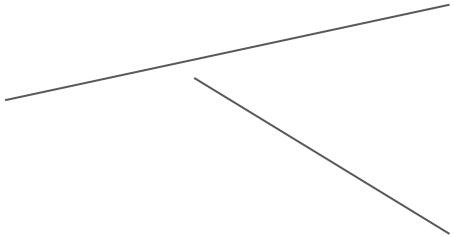
Output: true if the line segments intersect; false otherwise.



→ True



→ False

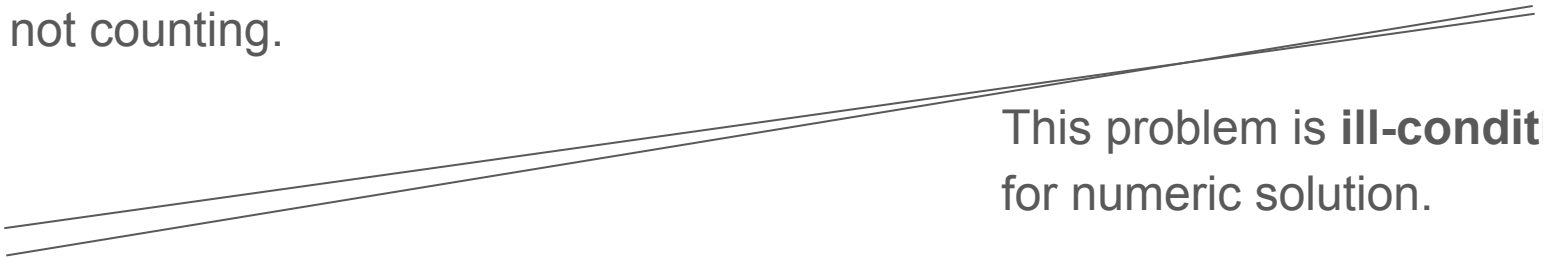


→ False

Intersection Determination

We would like to avoid trigonometry (slow).

The “high school maths” approach based on two equations of the form y = mx + c leads to divisions, which are slow in floating point, and introduce error that cannot be managed as effectively as with addition and multiplication (a concept known as *infinite precision*). This can lead to incorrect answers: small floating point errors can lead to the intersection of these two lines being “off the end” of the segments so not counting.

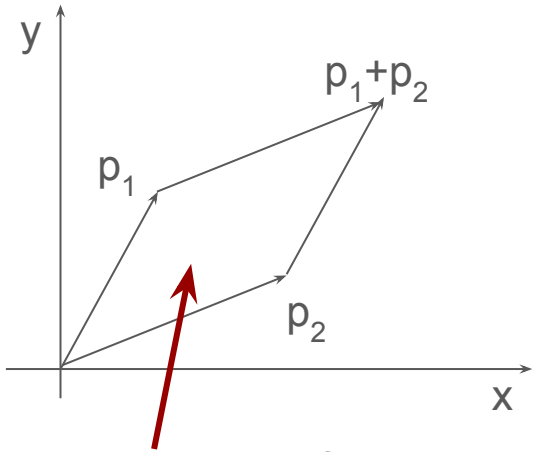


This problem is **ill-conditioned** for numeric solution.

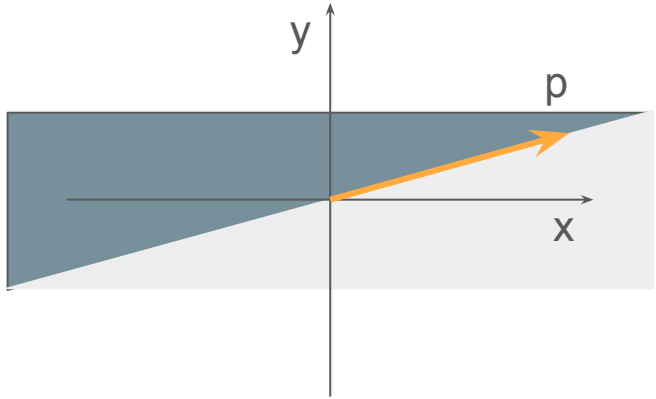
💡 Definition of an ill-conditioned problem: a small change in the input data can result in a large change in the output.

Cross Products

The vector cross product turns out to be very useful.



The cross product of p_1 and p_2 can be thought of as the signed area of the parallelogram.



The darker regions contains position vectors that are anticlockwise from p; the lighter region contains vectors that are clockwise from p.

💡 This is Figure 33.1 from CLRS3.

Matrix Determinants

$$p_1 \times p_2 = \det \begin{bmatrix} x1 & x2 \\ y1 & y2 \end{bmatrix}$$
$$= x_1y_2 - x_2y_1$$
$$= -p_2 \times p_1$$

If $p_1 \times p_2 > 0$ then p_1 is clockwise from p_2 with respect to the origin.
If $p_1 \times p_2 < 0$ then p_1 is anticlockwise from p_2 with respect to the origin.
If $p_1 \times p_2 = 0$ then p_1 and p_2 are collinear (parallel or antiparallel).

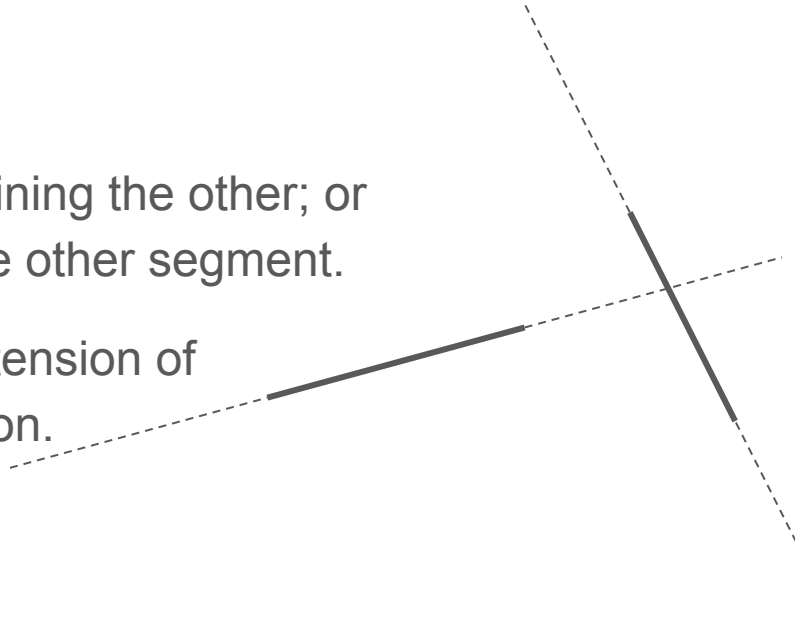
Line Segment Intersection

Check whether each line segment **straddles** the extension of the other. The **extension** of a line segment is the (infinite) line containing its two endpoints, i.e. drop the constraint that $0 \leq \alpha \leq 1$.

Two line segments intersect if and only if

- each segment straddles the line containing the other; or
- an endpoint of one segment lies on the other segment.

In this example, one segment cross the extension of the other, but *not vice-versa*. No intersection.



SEGMENTS-INTERSECT(p_1,p_2,p_3,p_4) [1]

```
1   d1 = DIRECTION(p3,p4,p1)        // Relative orientation of
2   d2 = DIRECTION(p3,p4,p2)        // each endpoint w.r.t. the
3   d3 = DIRECTION(p1,p2,p3)        // other segment
4   d4 = DIRECTION(p1,p2,p4)
5   if ((d1>0 && d2<0) || (d1<0 && d2>0)) &&
       ((d3>0 && d4<0) || (d3<0 && d4>0))
6      return true
```

💡 If p3→p1 and p3→p2 have opposite directions w.r.t. p3→p4 then p1p2 straddles p3p4.

💡 If p1→p3 and p1→p4 have opposite directions w.r.t. p1→p2 then p3p4 straddles p1p2.

SEGMENTS-INTERSECT(p_1,p_2,p_3,p_4) [2]

```
7   else if d1==0 && ON-SEGMENT(p3,p4,p1)  return true
8   else if d2==0 && ON-SEGMENT(p3,p4,p2)  return true
9   else if d3==0 && ON-SEGMENT(p1,p2,p3)  return true
10  else if d4==0 && ON-SEGMENT(p1,p2,p4)  return true
11  return false
```

```
DIRECTION(pi,pj,pk) = (pk-pi) x (pj-pi)
```

```
ON-SEGMENT(pi,pj,pk) = (min(xi,xj) ≤ xk ≤ max(xi,xj)) &&
                       (min(yi,yj) ≤ yk ≤ max(yi,yj))
```

💡 If p1 or p2 is on p3p4 then the segments intersect if that point is within the limits of the segment (L7,8).

💡 If p3 or p4 is on p1p2 then the segments intersect if that point is within the limits of the segment (L9,10).

n-Segment Intersection Problem

Input: n line segments, each specified as pairs of endpoints, p_i for $1 \leq i \leq n$.

Output: true if any pair intersects; false otherwise.

Obvious solution: solve the segment intersection problem for all pairs, $O(n^2)$.

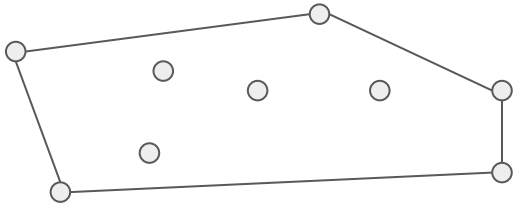
There is a smarter solution called **sweeping** with running time O(n lg n) that exploits the geometry of lines in a plane to constrain the cases that must be considered. Supervision exercise!

Convex Hull Problem

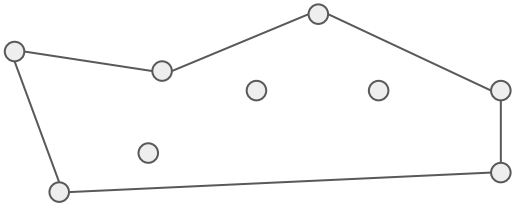
Input: a set of n>2 points p_i for $1 \leq i \leq n$. At least 3 points are not collinear (so the polygon is not a zero-area line).

Output: an ordered list of points forming a convex hull for the input points.

The furthest-apart of a set of points in a plane are both on the convex hull. The convex hull of a set of points is a minimal subset that forms a convex polygon with none of the points outside the polygon (i.e. either inside or on the edge).



✅ Convex Hull



❌ Not convex!

Five Solutions

1. Rotational Sweeps
   a. Graham's Scan — O(n lg n)
   b. Jarvis's March — O(n h)
2. Incremental — O(n lg n)
3. Divide and Conquer — O(n lg n)
4. Prune and Search — O(n lg h)

n: number of points in the input data

h: number of points on the convex hull produced

Prune-and-Search is asymptotically fastest since $h \leq n$.

Graham's Scan [1]

- Start at the left-most of the bottom-most points.
- Sort the points by increasing polar angle relative to a horizontal line through this point.
  - Resolve tie-breaks by retaining only the point farthest from the start point.
- Push the first three points onto an initially empty stack.
- For each of the other points, p, taken in the sorted order:
  - Pop off the stack until the directed segment from the next-to-top vertex on the stack to the top vertex on the stack forms a (strictly) left turn with the directed segment from top vertex to p
  - Push p onto the stack.
- The points on the stack are the convex hull.

Graham's Scan [2]

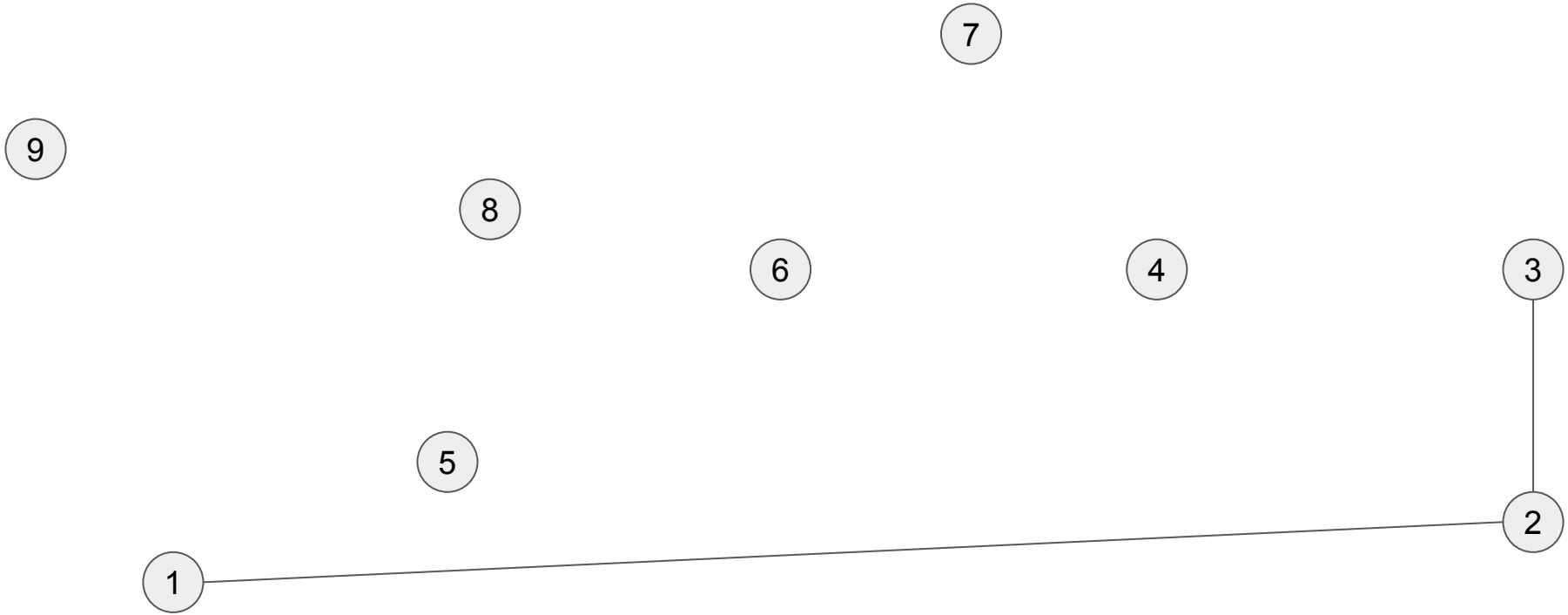
To sort by polar angle, we do not need to compute the angles!

The cross product $a \times b = |a|\,|b| \sin \theta$, where θ is the angle between the vectors a and b.

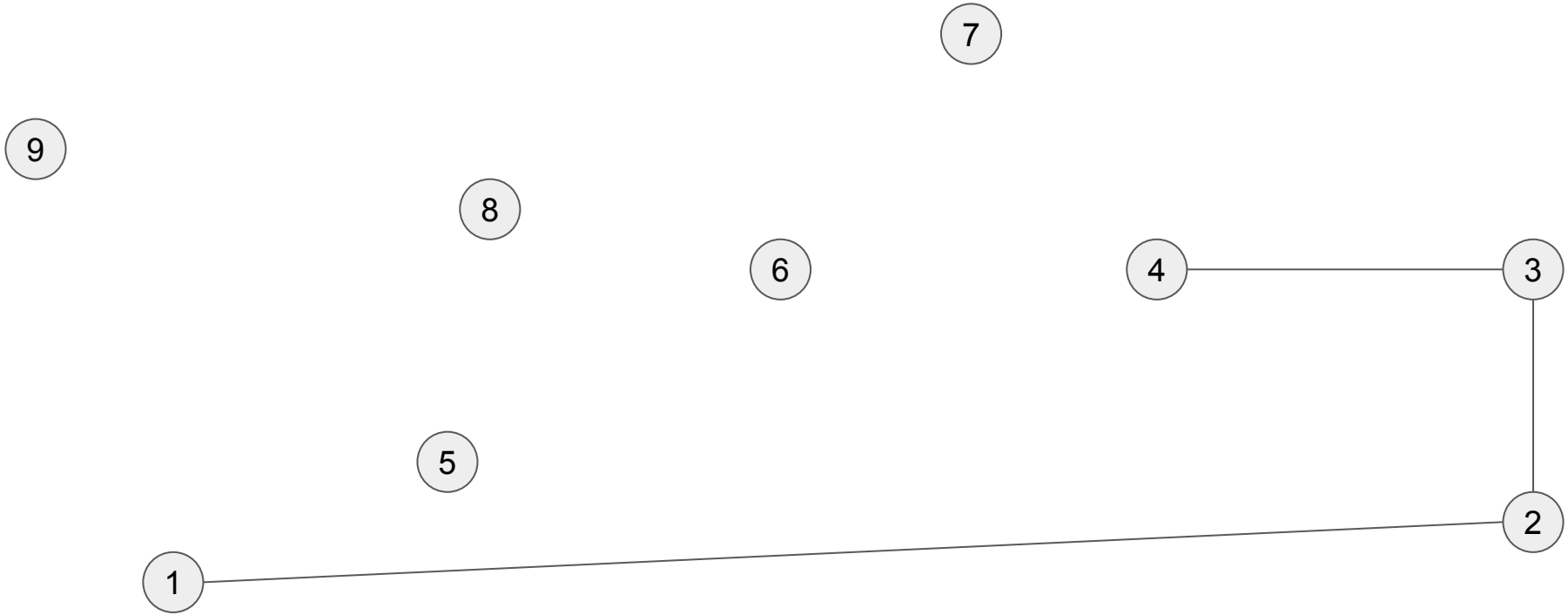
If a and b are unit vectors, sorting by the value of the cross product is the same as a sort by θ because $\sin \theta$ is monotonic with θ for $-\pi/2 \leq \theta < \pi/2$.

Normalising the vectors is often quicker than trigonometry.

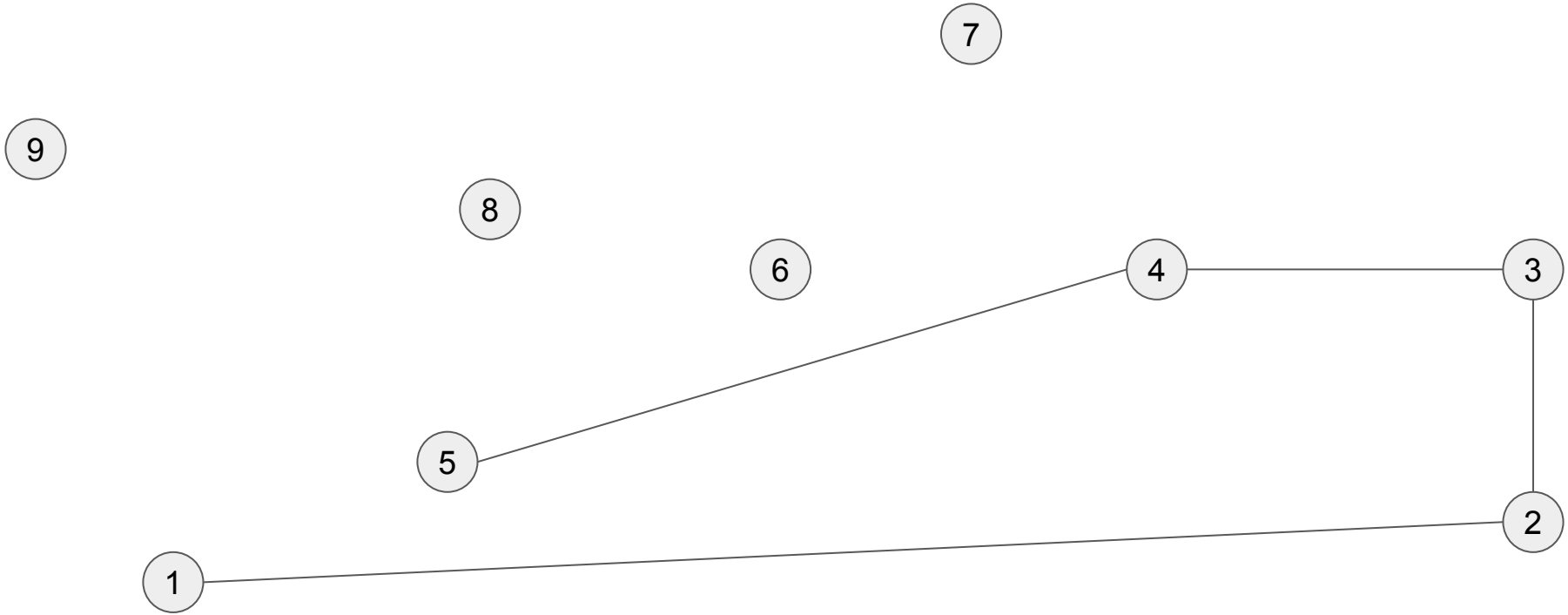
Graham’s Scan [3]



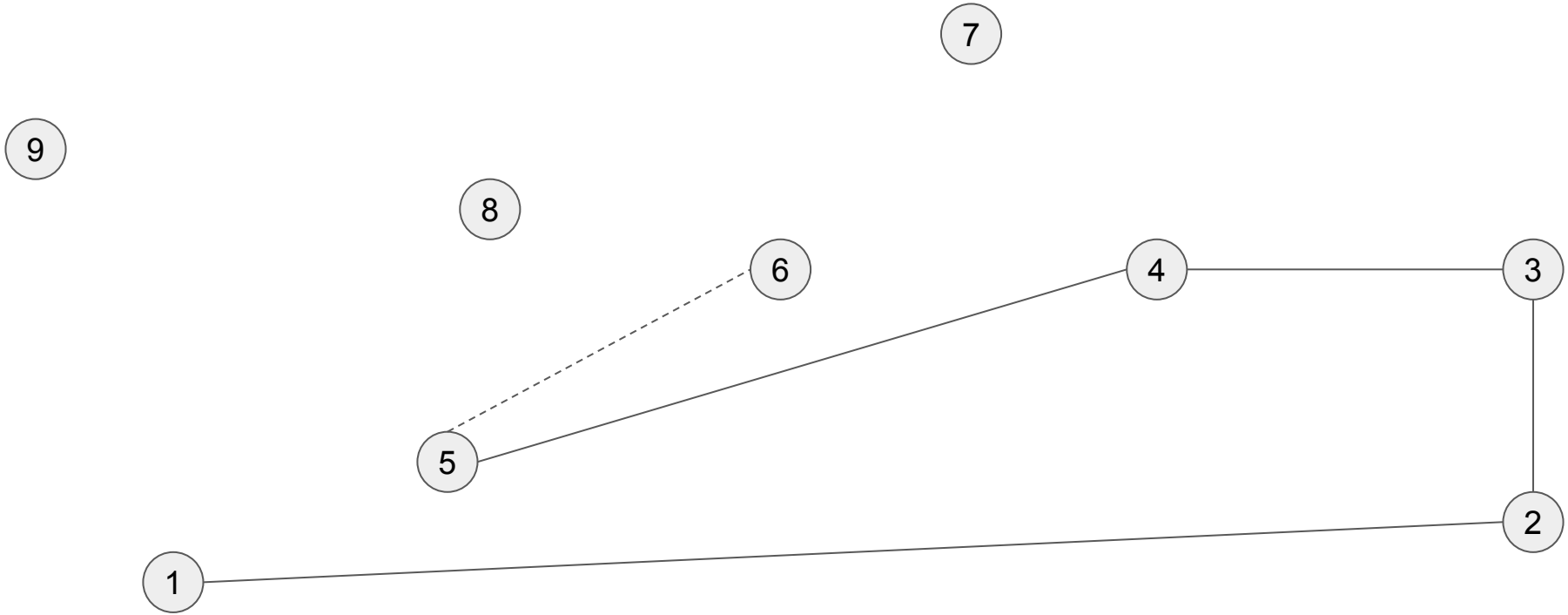
Graham’s Scan [4]



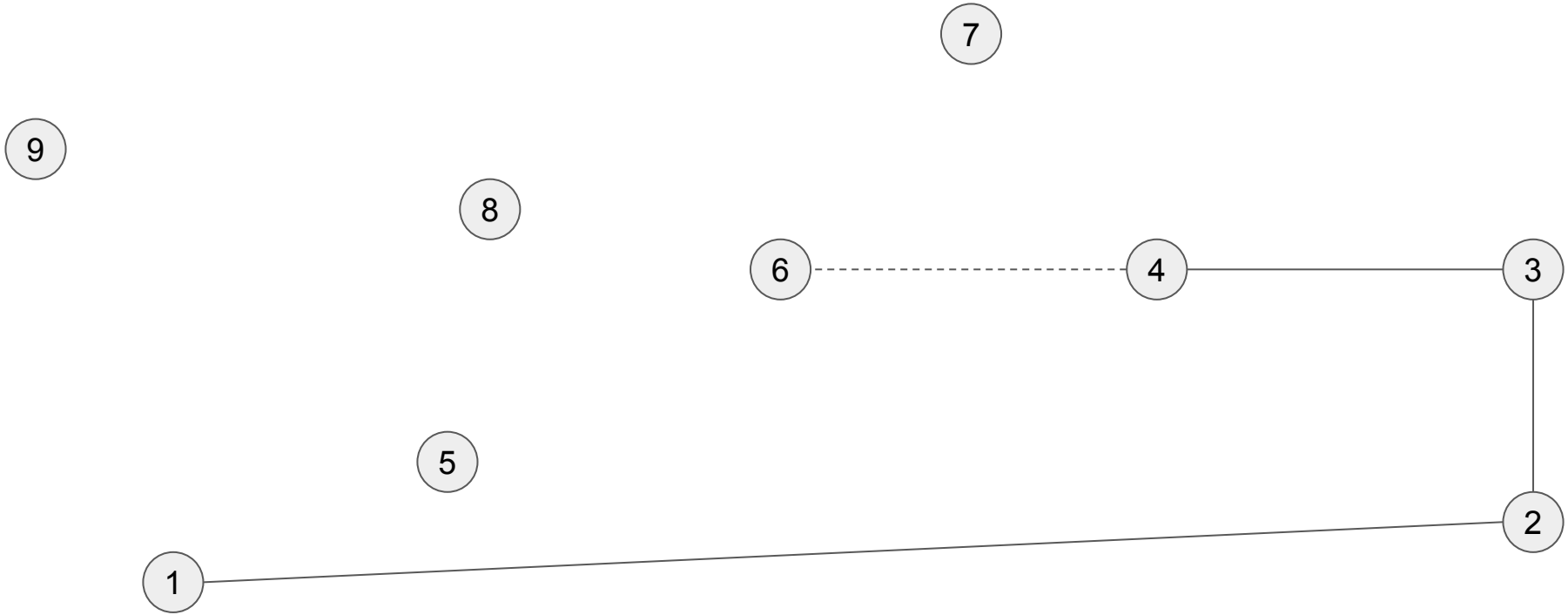
Graham’s Scan [5]



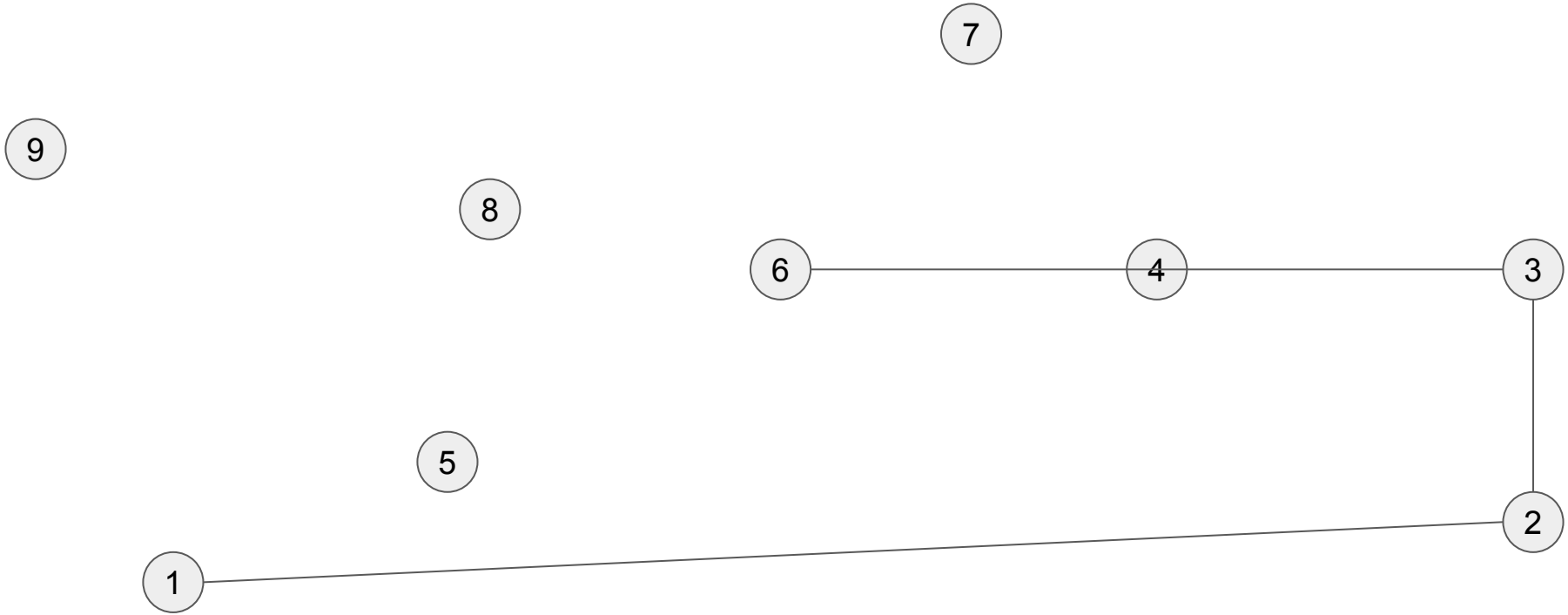
Graham’s Scan [6]



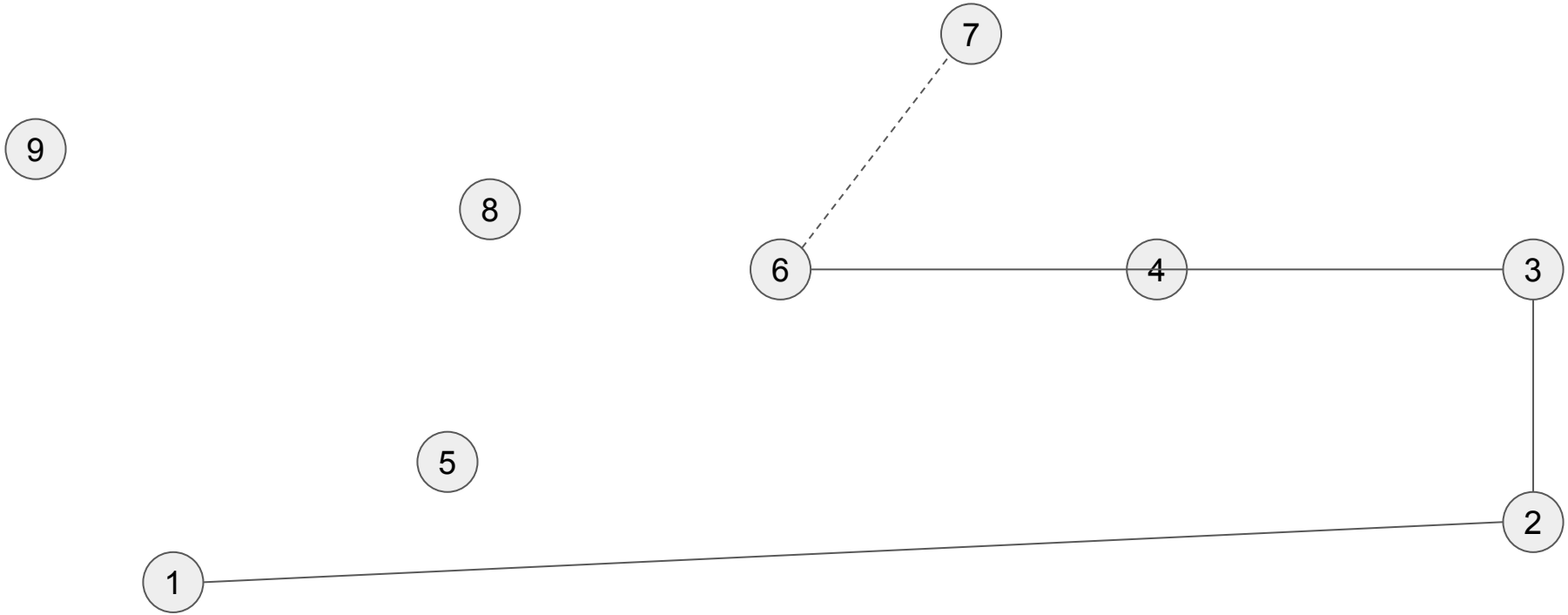
Graham’s Scan [7]



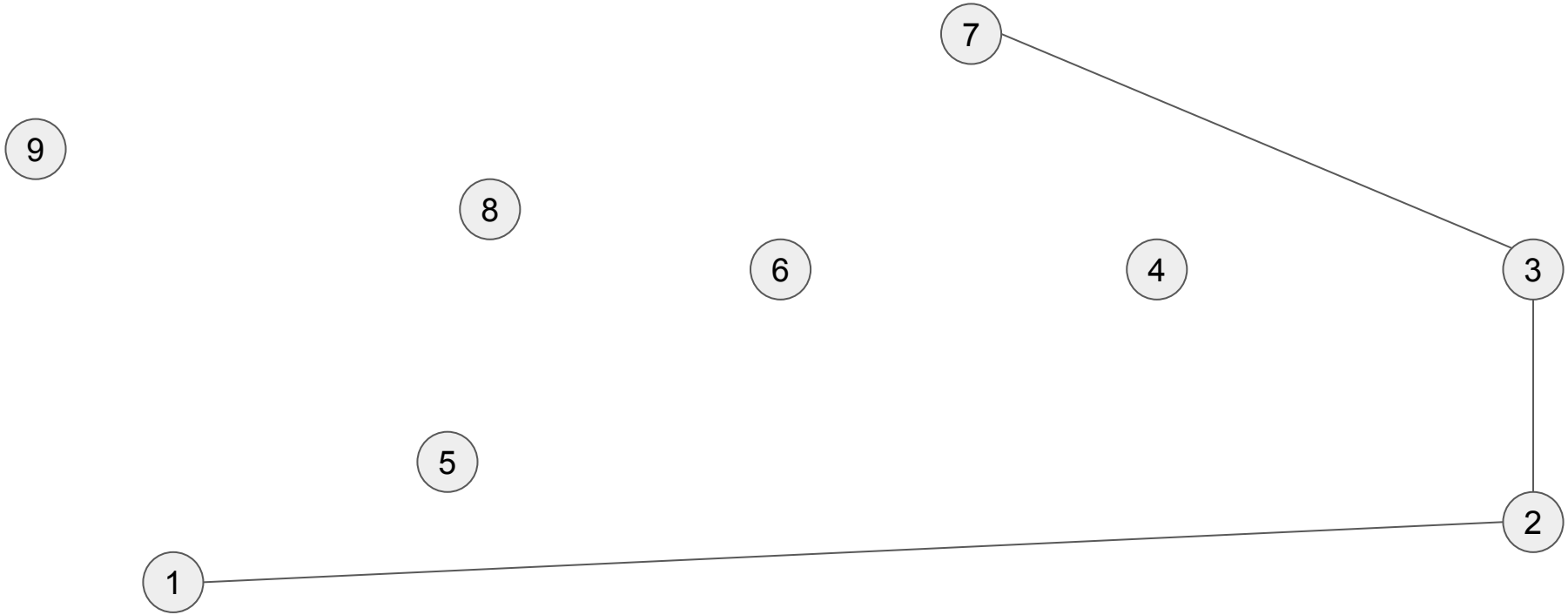
Graham’s Scan [8]



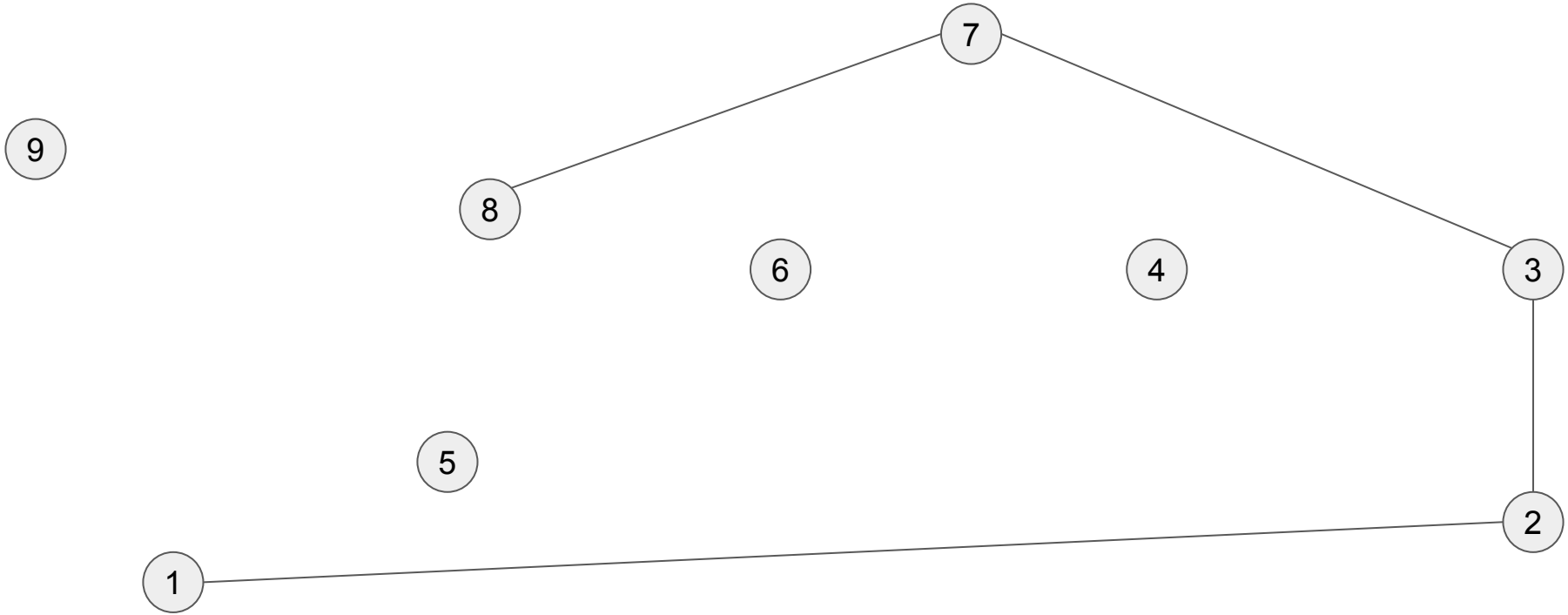
Graham’s Scan [9]



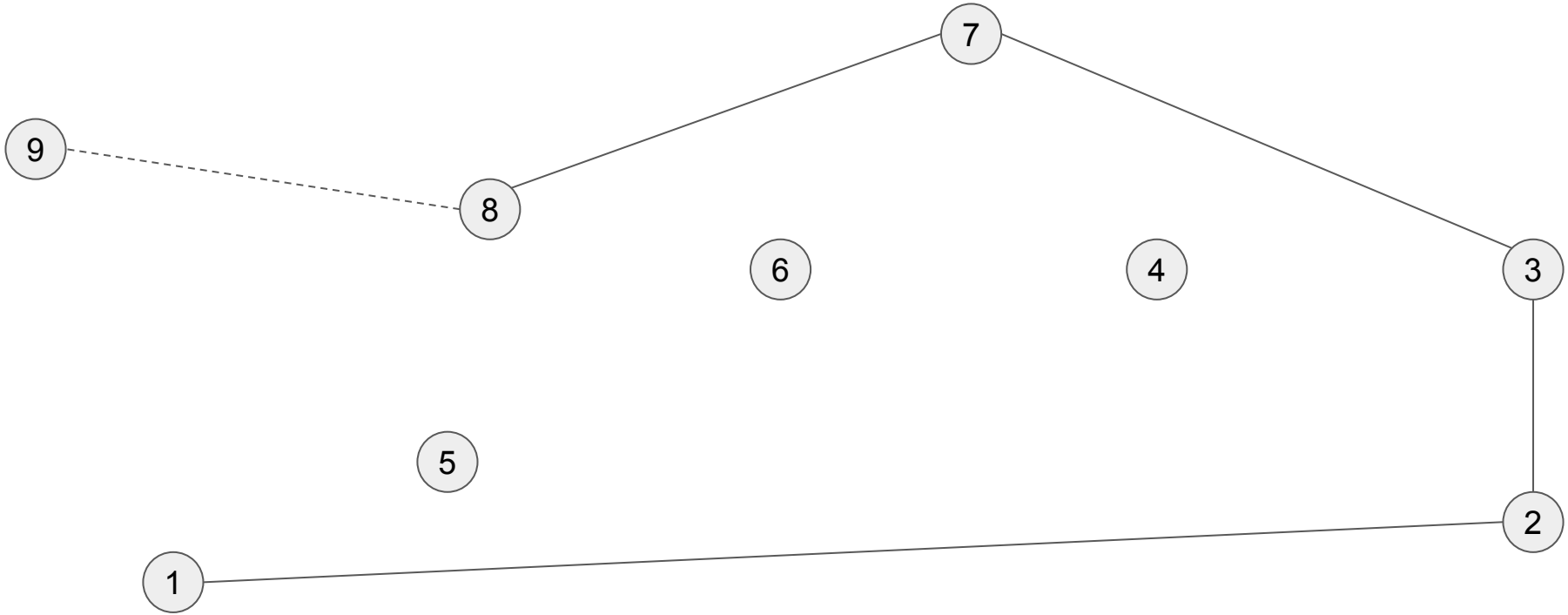
Graham’s Scan [10]



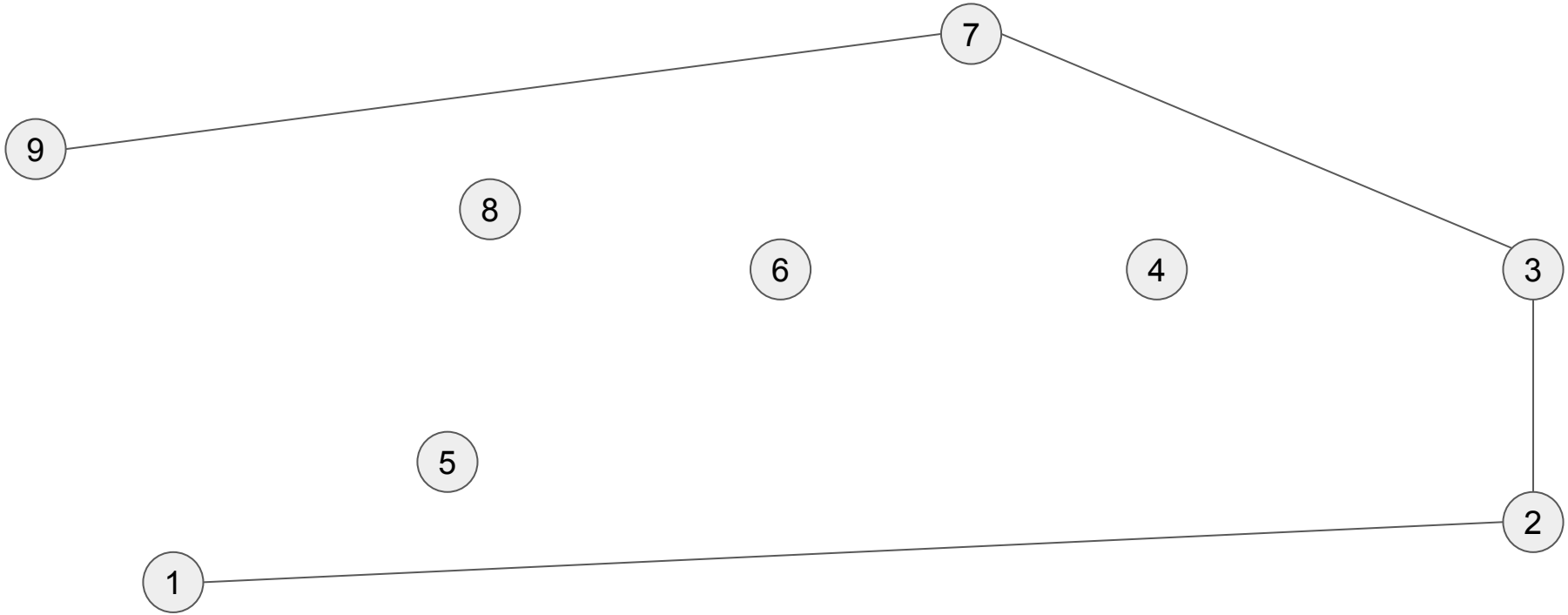
Graham’s Scan [11]



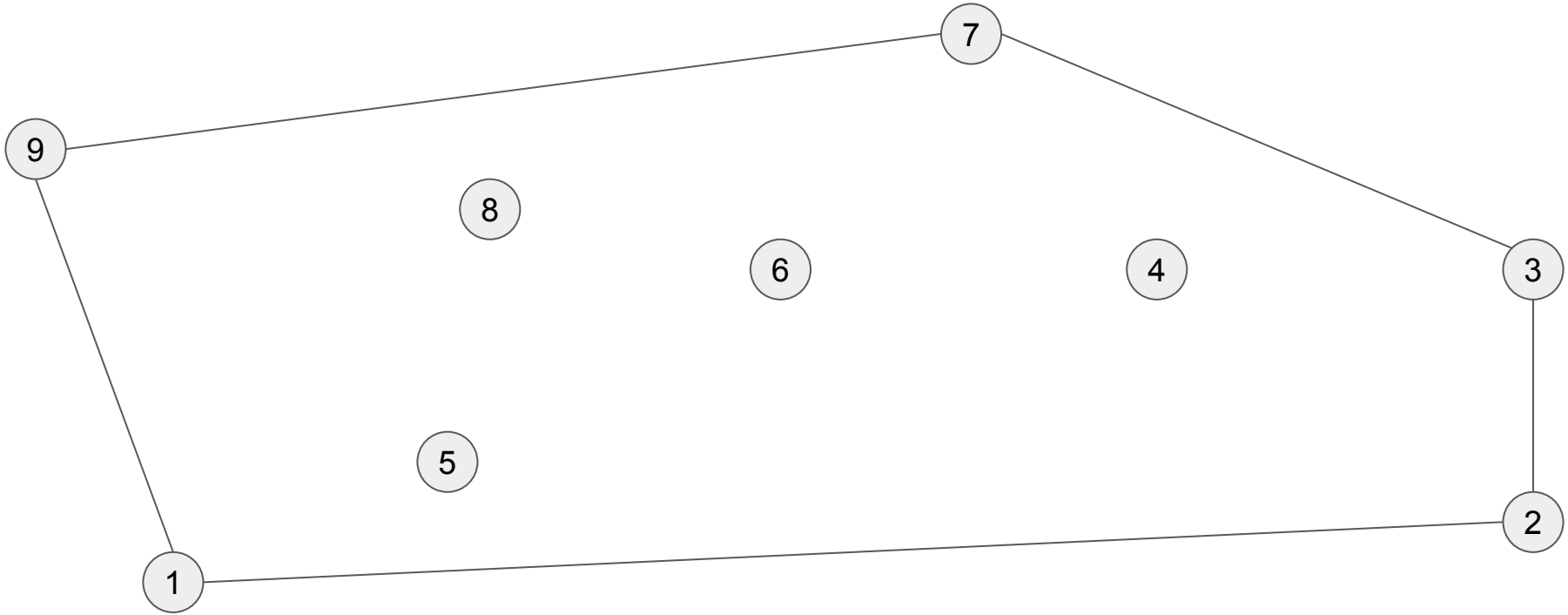
Graham’s Scan [12]



Graham’s Scan [13]



Graham’s Scan [14]



Analysis of Graham's Scan

Calculating one polar angle is O(1). Calculating n of them is O(n).

Sorting n polar angles is O(n lg n), with any sensible comparison-based sort (including the tie-break logic to discard points with sub-maximal distance).

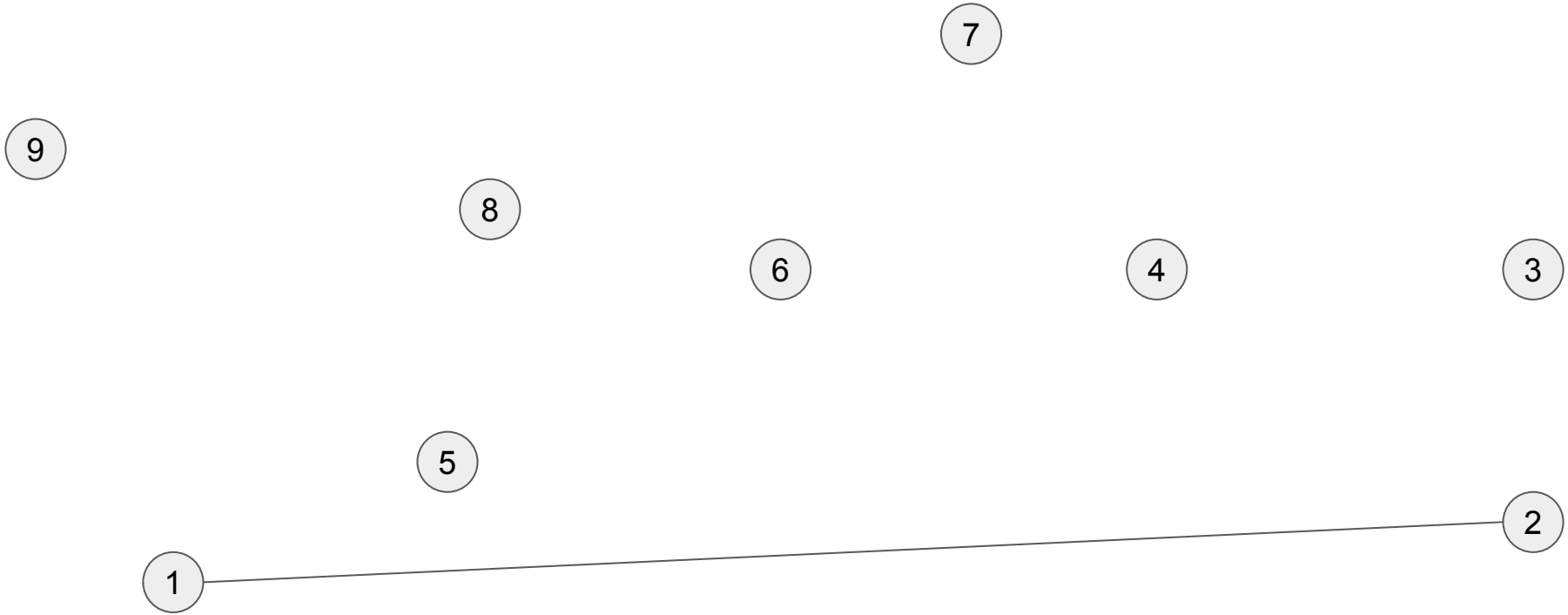
As we walk around the hull, each point is only pushed to the stack at most once and is removed at most once. Every comparison either adds a point to the stack or removes a point from the stack. Hence the walk is O(n).

Graham's Scan costs O(n lg n), dominated by the sorting step.

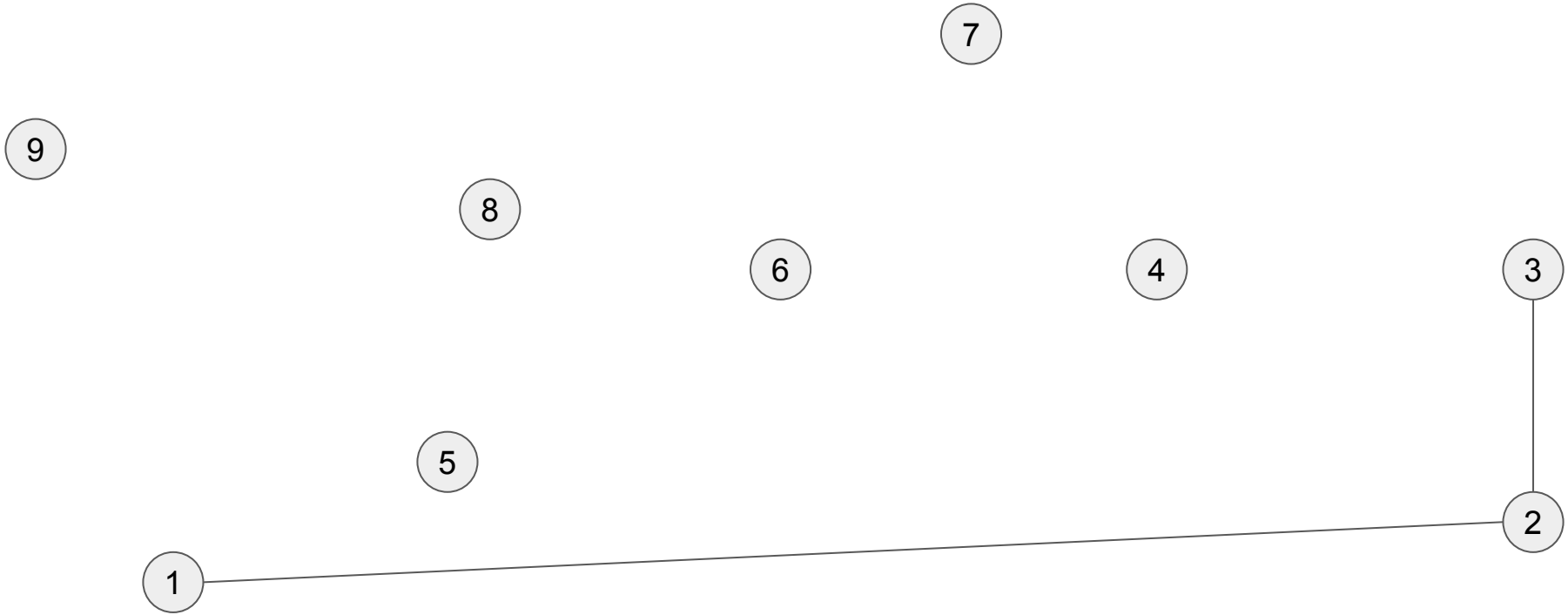
Jarvis's March [1]

- Start with the left-most of the bottom-most points, p_1, which is on the hull.
- Find the point p_2 with the least polar angle relative to a horizontal line through p_1. p_2 is also on the hull.
- Repeatedly find the point p_{i+1} with the least polar angle relative to the line through p_{i-1} and p_i. p_{i+1} is on the hull. The p_i form the **right chain**.
  - The repetition continues until a top-most point is reached (might not be unique).
- Repeat the previous two bullets to find the **left chain** using greatest polar angles.
- Join the right chain and left chain to get the convex hull.

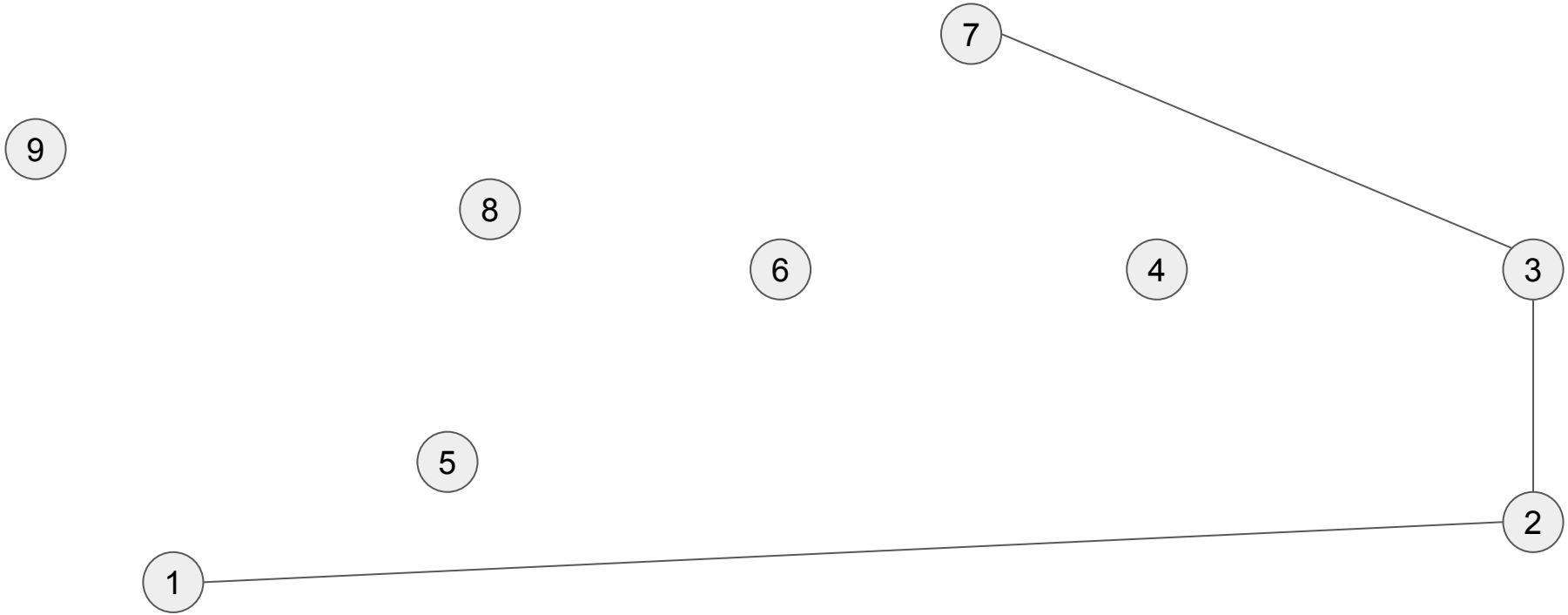
Jarvis’s March [2]



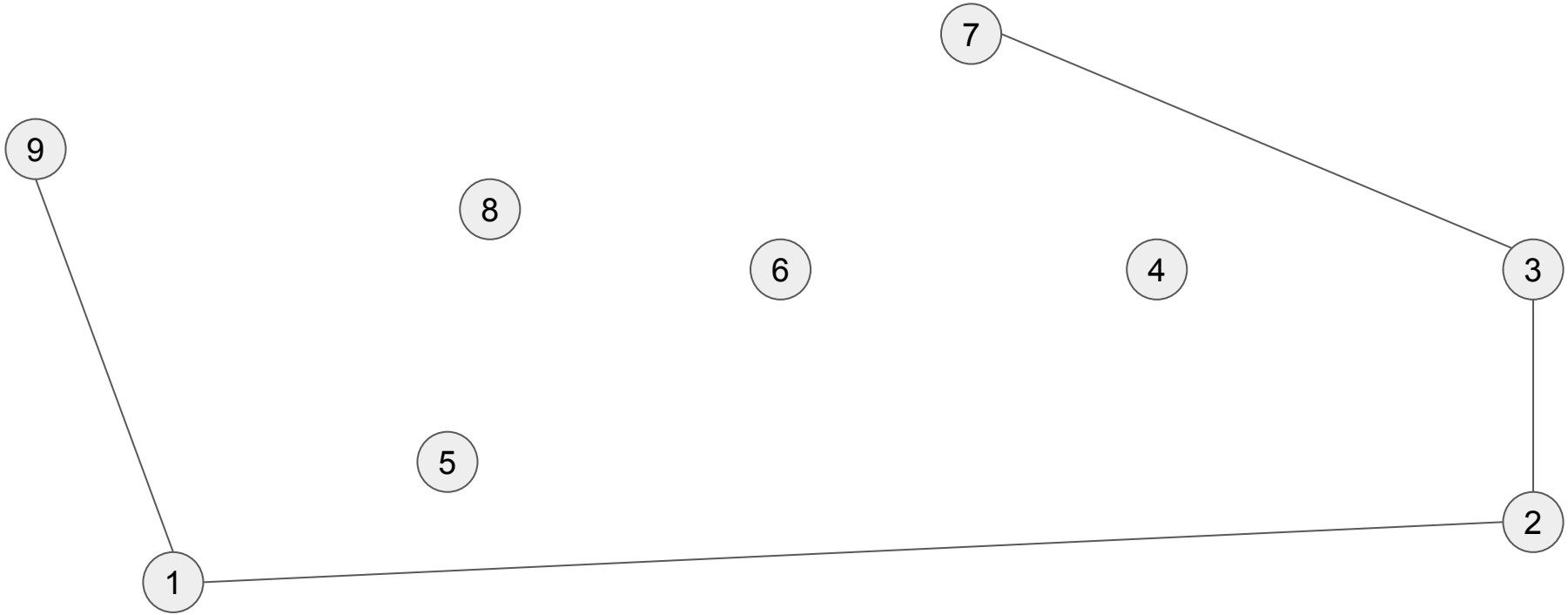
Jarvis’s March [3]



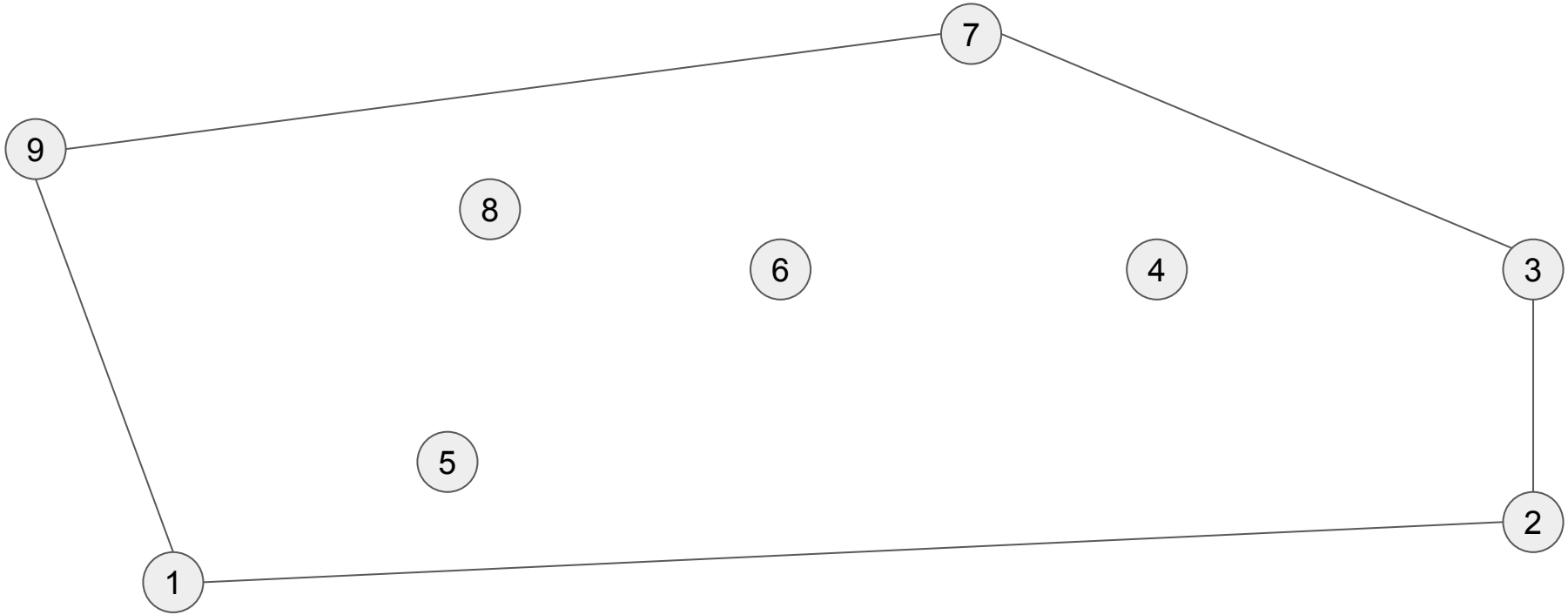
Jarvis’s March [4]



Jarvis’s March [5]



Jarvis’s March [6]



Analysis of Jarvis's March

Calculating one polar angle is O(1). Calculating n of them is O(n).

Finding the minimum of n numbers is O(n).

Repeating that h times is O(n h).

The right/left chain allows us to exploit the cross product trick for comparisons because the polar angles, θ, we handle are always in the range $-\pi/2 \leq \theta < \pi/2$.

Revision Guide / Summary of Algorithms 2 [1]

- Graphs
  - Representing the edge set with adjacency lists and adjacency matrices
  - Terminology
- Graph colouring problems: vertex, edge, face colouring
- Breadth-first search
  - With the concept of ‘depth’ to solve vertex colouring
  - Subgraph induced by the predecessors: breadth first tree
- Depth-first search
  - Discovery time and finish time for each vertex
  - Topological sort
- Edge classification: tree edge, back edge, forward edge, cross edge

Revision Guide / Summary of Algorithms 2 [2]

- Strongly connected components
  - Two DFSs and the transpose graph
- Shortest path problems:
  - Single-source shortest paths
  - Single-destination shortest paths
  - Single-pair shortest path
  - All-pairs shortest paths
- Complications caused by negative edges, negative cycles, zero-weight cycles
- Bellman-Ford
  - Introduced the concept of edge relaxation
  - Special case for directed acyclic graphs with lower costs

Revision Guide / Summary of Algorithms 2 [3]

- Optimal substructure led to Dijkstra's algorithm
  - Unable to handle negative edge weights
  - Proof of correctness using the convergence lemma
- Matrix multiplication methods for all-pairs shortest paths
  - Mapping domain-specific problems to other theory, to pull in speed-ups from other research
  - Repeated squaring
  - Floyd-Warshall
- Johnson's algorithm
  - Introduced the concept of reweighting

Revision Guide / Summary of Algorithms 2 [4]

- Flow networks
  - Capacity
  - Max-Flow Min-Cut Theorem
  - Ford-Fulkerson (Edmunds-Karp as optimisation)
  - Augmenting paths, flow cancellation
- Bipartite matchings
  - Maximum bipartite matchings (Hopcroft-Karp as an optimisation)
  - Maximum and maximal matchings
- Minimum spanning trees
  - Safe edge theorem
  - Kruskal’s algorithm
  - Prim’s algorithm

Revision Guide / Summary of Algorithms 2 [5]

- Amortised analysis
  - Aggregate method
  - Accounting method
  - Potential method
- Mergeable Priority Queues
  - Binomial Heaps
  - Fibonacci Heaps, golden ratio, peculiar property giving them their name
- Disjoint set representations
  - Path compression and union-by-rank

Revision Guide / Summary of Algorithms 2 [6]

- Geometric algorithms
  - Simple, planar and closed polygons
  - Defining the inside and outside
  - Winding numbers
  - Line segment intersection problems
  - Cross-product tricks for numerical stability and performance
- Convex Hulls
  - Graham’s scan
  - Jarvis’s March
  - … and I tantalised you with the “Search and Prune” asymptotically optimal method!

Thank you for listening!

I hope you enjoyed the course.
Please fill in the lecture feedback forms.
Good luck in the exams!