CS2040S Part 2

W9L1 Graphs

Basic Terminologies

Graphs consists of 2 types of elements

- Nodes (or vertices): ≥ 1
- (except for vacuously true case: 0 node)

Hypergraph

- Nodes (same as ←)
- Edge: each edge connects ≥ 2 nodes; each edge is unique

Multigraph

- Nodes (same ←)
- Edges: 2 nodes may be connected by > 1 edge

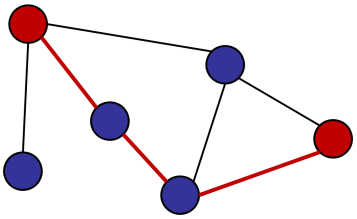
- Edges (or arcs): each edge connects 2 nodes; each edge is unique

Graph G = <V, E>

- V: set of nodes |V| > 0
- E: set of edges
  - $E \subseteq (v, w) : \{(v \in V), (w \in V)\}$
  - $e = (v, w)$
  - For all $e_1, e_2 \in E : e_1 \neq e_2$

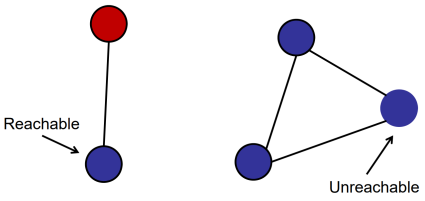
Connected:

– Every pair of nodes is connected by a path.



Disconnected:

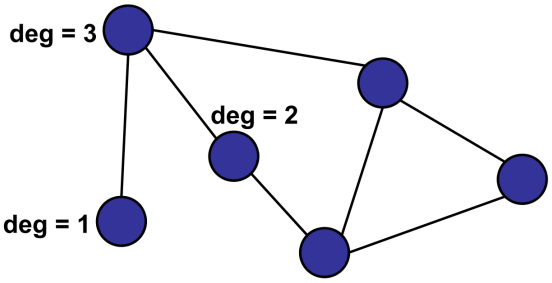
– Some pair of nodes is not connected by a path.



– Two **connected components**.

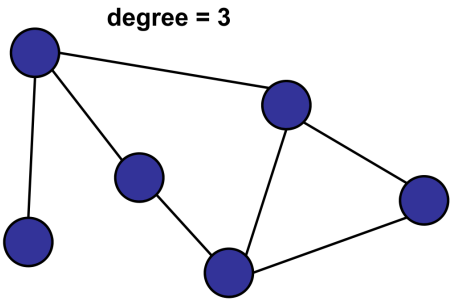
Degree of a node:

– Number of **adjacent** edges.



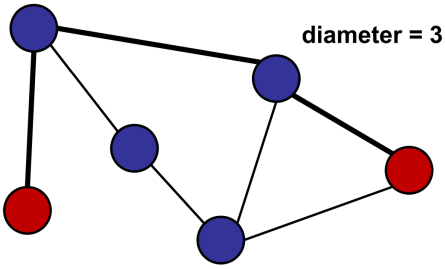
Degree of a graph:

– Maximum number of **adjacent** edges.



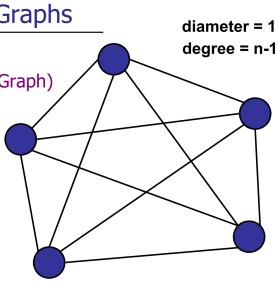
Diameter:

– Maximum distance between two nodes, following the shortest path.



Special Graphs

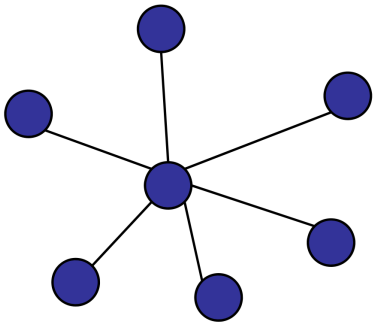
Clique
(Complete Graph)



All pairs connected by edges.

Special Graphs

Star



One central node, all edges connect center to edges.

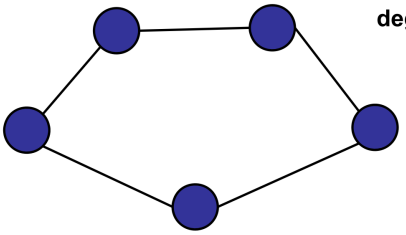
deg = n - 1; diameter = 2S

Special Graphs

Cycle

diameter = n/2
or
diameter = n/2-1

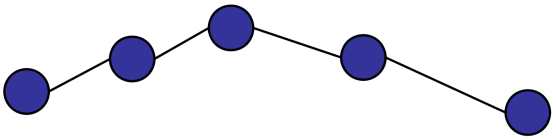
degree = 2

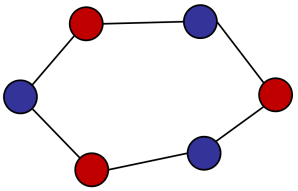


Special Graphs

Line (or path)

diameter = n-1
degree = 2

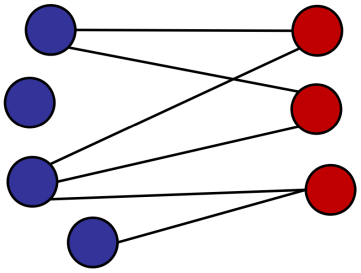




another eg of bipartite graphs; all even cycles are bipartite graphs

Special Graphs

Bipartite Graph



Nodes divided into two sets with no edges between nodes in the same set.

diameter = n - 1

- star, trees, grid graphs are bipartite

Practical Application:

Sliding Puzzle

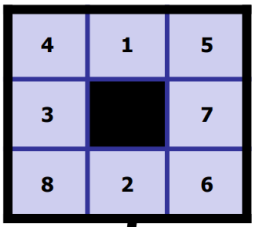
Nodes:

- State of the puzzle
- Permutation of 9 titles:

Edges:

- 2 states are edges if they differ by only 1 move
- Max degree = 4: middle empty, 4 possibilities in 1 move

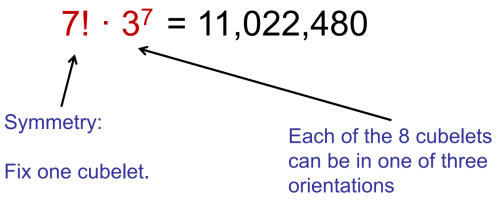
$N_{moves} \leq diameter$



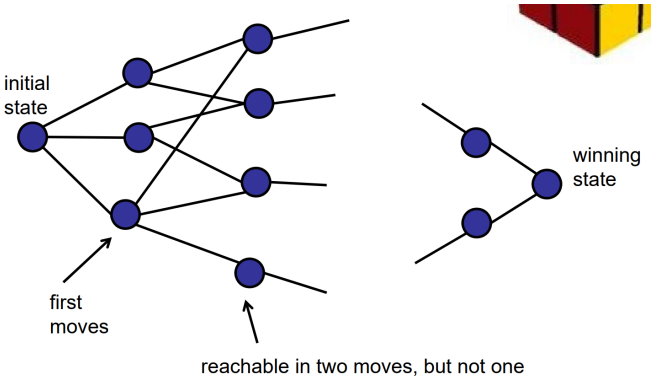
- $|E| < 4 \cdot 9! < 1,451,520$

Rubik's Cube

No. of vertices (2x2x2)



- $diameter = \Theta(n^2/log(n))$

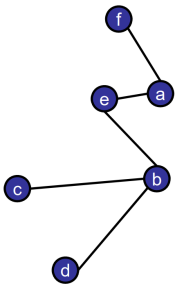


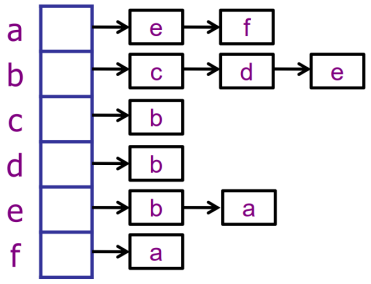
Graph Representations

Adjacency List

Graph consists of:

- nodes: stored in an array
- edges: linked list per node (neighbou)





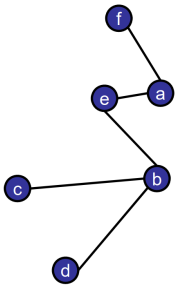
⌨️ Pseudo-Code for Adjacency List

```
class NeighborList extends LinkedList<Integer> {
}

class Node {
  int key;
  NeighborList nbrs;
}

class Graph {
```

```
  Node[] nodeList;
}

/* Alternative: using a LL in another LL (less preferred) */
class Graph {
  List<List<Integer>> nodes;
}
```

Adjacency Matrix

- Edges = pairs of nodes
- $A[v][w] = 1 \iff (v, w) \in E$
- paths of length n = A^n



	a	b	c	d	e	f
a	0	0	0	0	1	1
b	0	0	1	1	1	0
c	0	1	0	0	0	0
d	0	1	0	0	0	0
e	1	1	0	0	0	0
f	1	0	0	0	0	0

⌨️ Pseudo-Code for Adjacency Matrix

```
class Graph {
  // boolean[][] adjMatrix;
  Node[][] adjMatrix;
}
```

Adjacency List v.s. Adjacency Matrix

Aa G = (V, E)	Memory usage	Cycle	Clique	Basic Rule	Query: v & w neighbours?	Query: find any neighbour	Enumerate all neighbours
Adjacency List	$O(V+E)$	$O(V)$	$O(V^2)$	use for sparse (|E| < |V|) graphs	O(min(|V| |E|)) Slow	Fast	Fast
Adjacency Matrix	$O(V^2)$	$O(V^2)$	$O(V^2)$	use for dense ($\|E\| = \Theta(V^2)$) graphs	Fast	Slow	Slow
Untitled							

Directed Graphs (Digraphs)

Graphs consists of 2 types of elements

- Nodes (or vertices): ≥ 1 (except for vacuously true case: 0 node)
- Edges (or arcs): each edge connects 2 nodes; each edge is unique; each edge is **directed**

Graph G = <V, E>

- V: set of nodes |V| > 0

- E: set of edges
  - $E \subseteq (v, w) : \{(v \in V), (w \in V)\}$
  - $e = (v, w)$ ← order matters!
  - For all $e_1, e_2 \in E : e_1 \neq e_2$

Basic Terminologies

- In-degree: no. of incoming edges
- Out-degree: no. of outgoing edges

⌨️ Pseudo-Code for Adj List

```
class NeighborList extends ArrayList<Integer> {
}

class Node {
  int key;
  NeighborList nbrs;
}

class Graph {
  Node[] nodeList;
}
```

Graph Searching

BFS

- Search level-by-evel
- follow outgoing edges
- ignore incoming edges

DFS

- search recursively
- follow outgoing edges
- backtrack (through incoming edges)

📁 Space Complexity of Digraph

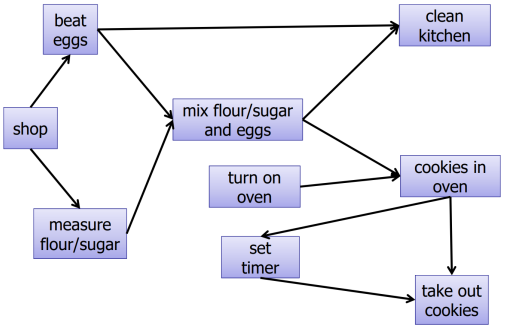
- adj list: $S(n) = O(V + E)$
  - nodes: stored in an array
  - outgoing edges: linked list per node
- adj matrix: $S(n) = O(V^2)$

Practical Application

Friendships

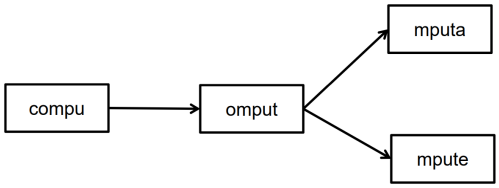
- Nodes: people
- Edge: friendship
- not always bidirectional (e.g. wechat v.s. ins)

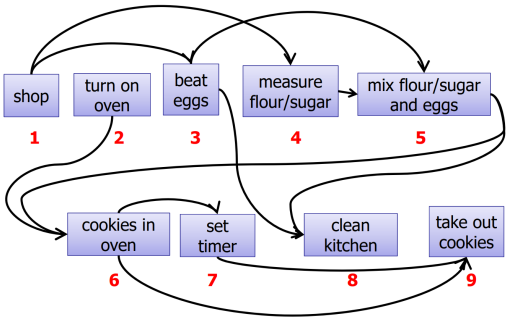
Scheduling



Markov text generation

- Nodes: `kgram`
- Edge: one `kgram` follows another





Topological order

Searching a Graph

Goal:

- start at some vertex s = start
- find some other vertex f = finish
  - or: visit all the nodes in the graph

2 techniques:

1. Breadth-First Search (BFS)
2. Depth-First Search (DFS)

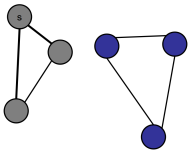
Graph Rep: Adjacency List

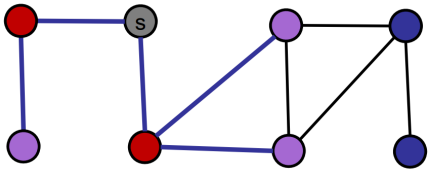
Breadth-First Search (BFS)

- Explore level by level
- `Frontier` : current level
- initially `{s}`
- advance frontier
- never goes backwards
- finds the shortest path

- build levels
- calculate `level[i]` from `level[i - 1]`
- skip already visited nodes

- NOTE: basic algo FAILS to visit every node in graphs with ≥ 2 components (disconnected)





Red = active frontier
Purple = next
Gray = visited
Blue = unvisited

⌨️ Pseudo-Code for BFS

```
BFS(Node[] nodeList, int startId) {
  boolean[] visited = new boolean[nodeList.length];
  Arrays.fill(visited, false);

  int[] parent = new int[nodelist.length];
  Arrays.fill(parent, -1);

  // the following 2 lines works for connected graphs
  // but fails for graphs with > 1 component
  Collection<Integer> frontier = new Collection<Integer>;
  frontier.add(startId);

  // main code
  while (!frontier.isEmpty()){
    Collection<Integer> nextFrontier = new … ;
    for (Integer v : frontier) {
      for (Integer w : nodeList[v].nbrList) {
        if (!visited[w]) {
          visited[w] = true;
          parent[w] = v;
          nextFrontier.add(w);
        }
      }
    }
  frontier = nextFrontier;
  }

  // the following lines ensure that the algo also works for graphs w > 1 components
  for (int start = 0; start < nodeList.length; start++) {
    if (!visited[start]){
      Bag<Integer> frontier = new Bag<Integer>;
      frontier.add(startId);
      // Main code to be included below
    }
  }


}
```

⌛Time Complexity

BFS (using adj list): $T(n) = O(V + E)$

- Vertex v = "start" once
- Vertex v added to `nextFrontier` (and frontier) once [O(V)]
  - after visited, never re-added
- Each `v.nbrlist` is enumerated once [O(E)]
  - when v is removed from frontier

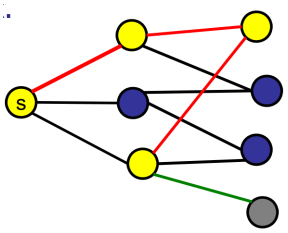
BFS (using adj matrix): $T(n) = O(V^2)$

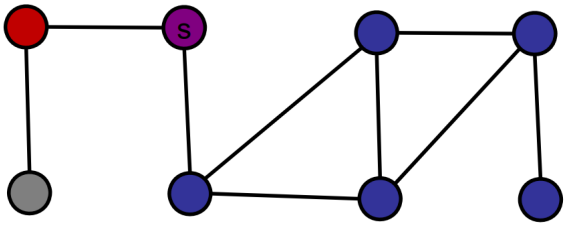
📁 Space Complexity

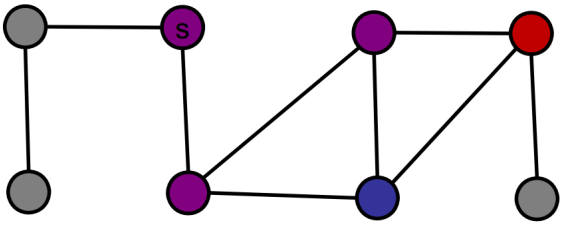
- adj list: $S(n) = O(V + E)$
- adj matrix: $S(n) = O(V^2)$

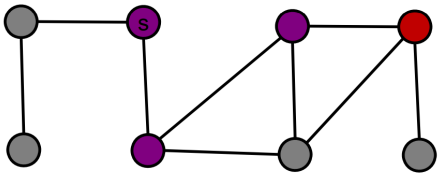
Depth-First Search (DFS)

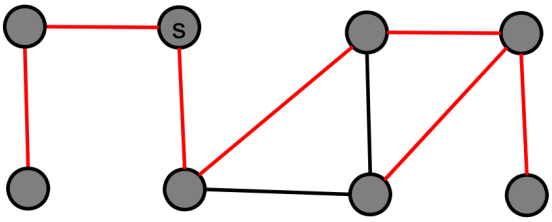
- follow path until you get stuck
- backtrack until you find a new edge
- recursively explore the graph
- never repeat a vertex











⌨️ Pseudo-Code for DFS

```
DFS(Node[] nodeList) {
  boolean[] visited = new boolean[nodeList.length];
  Arrays.fill(visited, false);
    for (start = i; start<nodeList.length; start++) {
      if (!visited[start]){
        visited[start] = true;
        // ProcessNode(v) - pre-order
        DFS-visit(nodeList, visited, start);
      }
  }
}

DFS-visit(Node[] nodeList, boolean[] visited, int startId){
  for (Integer v : nodeList[startId].nbrList) {
    if (!visited[v]) {
      visited[v] = true;
      DFS-visit(nodeList, visited, v);
    }
  }
}
```

⌛ Time Complexity of DFS

DFS (using adj list): $T(n) = O(V + E)$

- DFS-visit called only once per node - O(V)
  - never call DFS-visit again after visited
- In DFS-visit, each neighbour is enumerated - O(E)

DFS (using adj matrix): $T(n) = O(V^2)$

📁 Space Complexity

- adj list: $S(n) = O(V + E)$
- adj matrix: $S(n) = O(V^2)$

Implementation of Graph Search

Aa Name	Properties of parent edges	T(n)	S(n)	DS	How it works?	Details	Note
BFS	- parent edges form a tree (no cycles) - parent edges are the shortest paths from s (start node) - possibly high degree and/or diameter	- adj list: $O(V + E)$ - adj matrix: $O(V^2)$	- adj list: $O(V + E)$ - adj matrix: $O(V^2)$	Queue	Every time you visit a node, add all unvisited neighbours to the queue.	Add start-node to queue. Repeat until queue is empty: • Remove node v from the front of the queue. • Visit v. • Explore all outgoing edges of v. • Add all unvisited neighbours of v to the queue.	visit every node/edge in the graph, but not every path - takes exponential time to explore all paths
DFS	- DFS parent graph is a tree - DFS parent graphs does not contain the shortest paths	- adj list: $O(V + E)$ - adj matrix: $O(V^2)$	- adj list: $O(V + E)$ - adj matrix: $O(V^2)$	Stack	Every time you visit a node, add all unvisited neighbours to the stack.	Add start-node to stack. Repeat until stack is empty: • Pop node v from the front of the stack. • Visit v. • Explore all outgoing edges of v. • Push all unvisited neighbours of v on the front of the stack.	visit every node/edge in the graph, but not every path
Untitled							

W10L1 Weighted Graphs & SSSP

Weighted Graphs

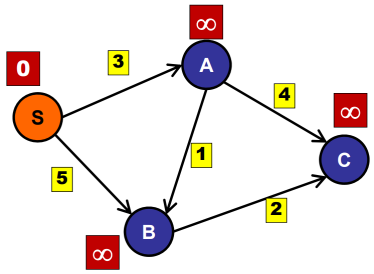
w(e): E → R

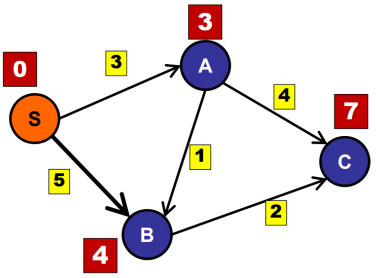
- weight = distance: δ(u, v) = distance from u to v
- Note: If all edges have the same weight ⇒ use BFS

Bellman-Ford Algorithm

Key idea: triangle inequality

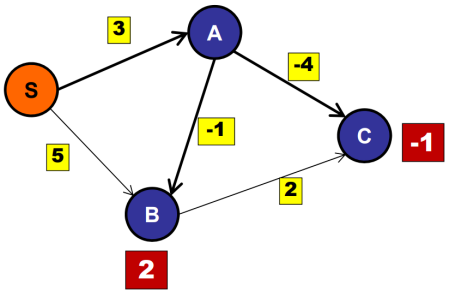
- δ(S, C) ≤ δ(S, A) + δ(A, C) where S and C are not directed connected
- maintain estimate for each distance
- Repeat |V| times: relax every edge
  - |V-1| is enough if there's no cycle
- Stop when “converges”: when an entire sequence of |E| relax operations have no effect



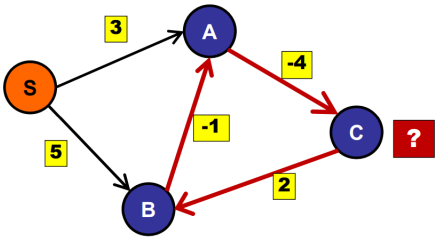


Works for graphs with negative weights as long as there is no cycle

- Run Bellman-Ford for |V| iterations to detect negative weight cycles



No problem!



d(S,C) is infinitely negative!

⌨️ Pseudo-Code for Bellman-Ford

```
int[] dist = new int[V.length];
Arrays.fill(dist, INFTY);
dist[start] = 0;

relax(int u, int v){
  if (dist[v] > dist[u] + weight(u,v))
    dist[v] = dist[u] + weight(u,v);
}

Bellam-Ford
n = V.length; // store nodes in a list/array
for (i=0; i<n; i++)
  for (Edge e : graph)
    relax(e)

// terminate early when an entire sequence of |E| relax operations have no effect

#Example Sequence
// relax(S, A)
// relax(A, C)
// relax(A, B)
// relax(S, B)
```

✳️ Invariant

- estimate ≥ distance
- If P is the shortest path from S to D, and if P goes through X, then P is also the shortest path from S to X (and from X to D)
- After k iterations of the outer loop, ~~every~~ at least k nodes within k hops of the source has a correct estimate.
- After i-th iteration, the i-th node of the shortest path must have its distance estimate set to the correct value. As the path is at most |V|−1 edges long, |V|−1 rounds suffices to find this shortest path. If the V-th iteration still relaxed some edges →

negative cycle.

⌛ Time Complexity

$T(n) = O(VE)$

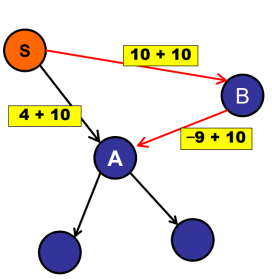
Dijkstra's Algorithm

Key idea:

- Maintain distance estimate for every node.
- Begin with empty shortest-path-tree.
- Repeat:
  - Find unfinished vertex with minimum estimate.
  - Add vertex to shortest-path-tree (priority queue e.g. AVL) - Mark vertex as finished
  - Relax all outgoing edges.
- only relax each edge once: O(E) cost for relaxation step.
- Source-to-Destination: can terminate as soon as we dequeue the destination node
  - a vertex is "finished" when it is dequeued
  - if the destination is finished, then stop

Assumptions:

- all edges weight ≥ 0
  - cannot reweigh graph (to offset negatives) overcounting



Path S-B-A: 21

Path S-A: 14

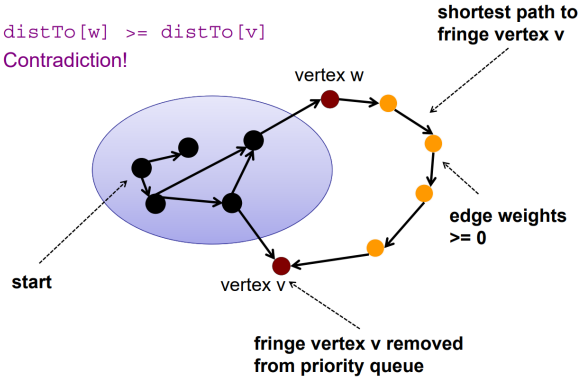
- extending a path does not make it shorter
- can be applied to negative weight with augmentation

✳️ Loop Invariant

- At the start of each iteration of the while loop, d[v] is the shortest path from s to v for each vertex v in S.
- i.e. Every “finished” vertex has correct estimate.

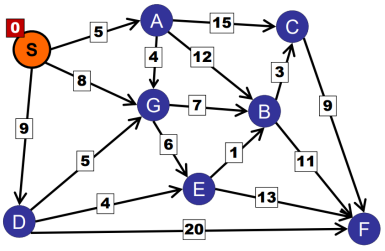
Proof by Induction:

- Base step: the only "finished" vertex is the start vertex
- Inductive step:
  - Remove vertex from priority queue.
  - Relax its edges.
  - Add it to finished.
  - Claim: it has a correct estimate → prove by contradiction



Dijkstra's Algorithm

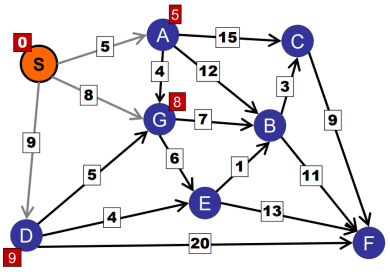
Step 1: Add source



Vertex	Dist.
S	0

Dijkstra's Algorithm

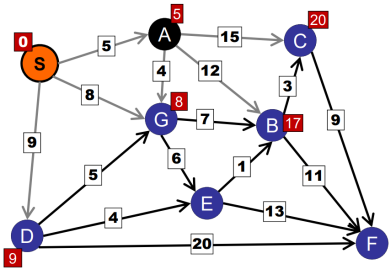
Step 2: Remove S and relax.



Vertex	Dist.
A	**5**
G	**8**
D	**9**

Dijkstra's Algorithm

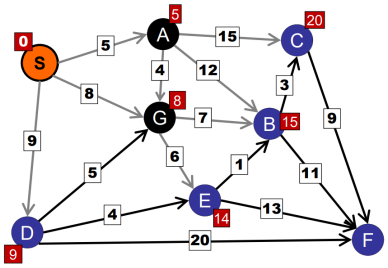
Step 3: Remove A and relax.



Vertex	Dist.
G	**8**
D	9
B	**17**
C	**20**

Dijkstra's Algorithm

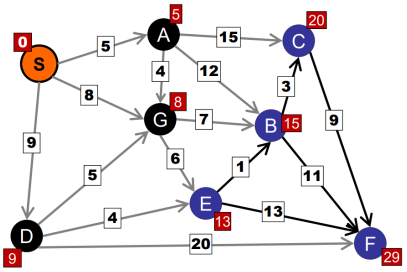
Step 4: Remove G and relax.



Vertex	Dist.
D	9
E	**14**
B	**15**
C	20

Dijkstra's Algorithm

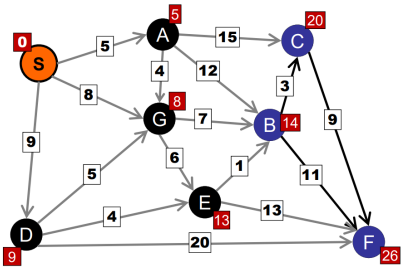
Step 5: Remove D and relax.



Vertex	Dist.
E	**13**
B	15
C	20
F	**29**

Dijkstra's Algorithm

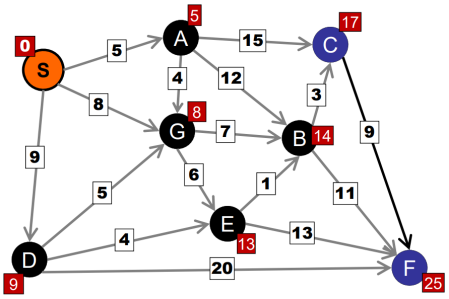
Step 5: Remove E and relax.



Vertex	Dist.
B	**14**
C	20
F	**26**

Dijkstra's Algorithm

Step 5: Remove B and relax.

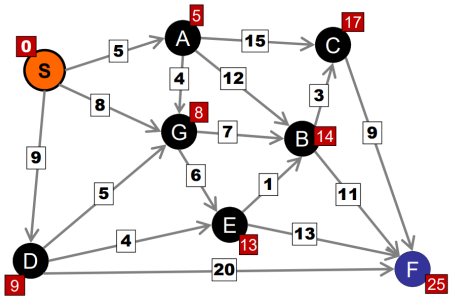


Vertex	Dist.
C	**20**
F	**25**

Dijkstra's Algorithm

Step 5: Remove C and relax.

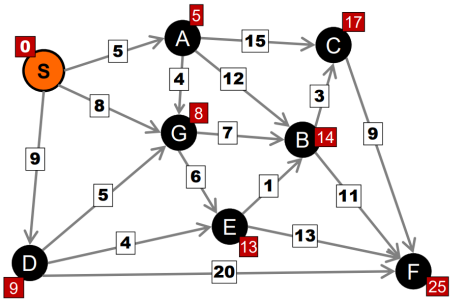
Vertex	Dist.
F	25



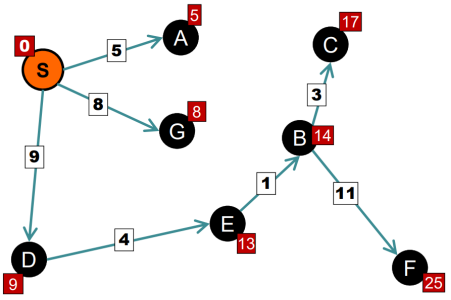
Dijkstra's Algorithm

Step 5: Remove F and relax.

Vertex	Dist.



Shortest Path Tree



⌨️ Pseudo-Code for Dijkstra's Algorithm

```
public Dijkstra{
  private Graph G;
  private IPriorityQueue pq = new PriQueue();
  private double[] distTo;

  searchPath(int start) {
    pq.insert(start, 0.0); // insert: |V| times - each node added once
    distTo = new double[G.size()];
    Arrays.fill(distTo, INFTY);
    distTo[start] = 0;

    while (!pq.isEmpty()) {
      int w = pq.deleteMin(); // deleteMin: |V| times
      for (Edge e : G[w].nbrList)
        relax(e); // relax: |E| times - each edge relax once
    }
  }

  relax(Edge e) {
    int v = e.from();
```

```
    int w = e.to();
    double weight = e.weight();
    if (distTo[w] > distTo[v] + weight) {
      distTo[w] = distTo[v] + weight;
      parent[w] = v;

      if (pq.contains(w))
        pq.decreaseKey(w, distTo[w]); // decreasesKey: |E| times
      else
        pq.insert(w, distTo[w]);
  }
}
```

AVL Tree

- Indexed by priority
- Existing operations: `deleteMin()`, `insert(key, priority)`
- Other operations: `contains()`, `decreaseKey(key, priority)`
- Find vertex using Hash Table:
  - map keys to locations in the binary tree
  - update hash table whenever binary tree changes

DS Performance

Aa PQ Implementation	insert	deleteMin	decreaseKey	containsKey	Total
Unsorted Array	1	V	1		$O(V^2)$
Sorted Array	V	1	V		$O(EV)$
AVL Tree	$log(V)$	$log(V)$	$log(V)$	1	O((V+E)log(V)) = $O(E \cdot log(V))$
Fibonacci Heap	1	$log(V)$	1		$O(E + V \cdot log(V))$
d-way Heap	$d \cdot log_d V$	$d \cdot log_d V$	$d \cdot log_d V$		$O(E \cdot log_{E/V} V)$
Untitled					

⌛ Time Complexity

- `insert` / `deleteMin`: |V| times each
  - each node added to priority queue once
- `relax` / `decreaseKey`: |E| times
  - each edge is relaxed once
- Priority queue operations: $O(log(V))$
- $T(n) = O((V + E) \cdot log(V)) = O(E \cdot log(V))$

Algorithm Comparison

Similarities:

- Maintain a set of explored vertices
- Add vertices to the explored set by following edges that go from a vertex in the explored set to a vertex outside the explored set

Differences:

BFS **DFS** **Dijkstra's**

- Take edge from vertex that was discovered **least** recently.
- Use queue

- Take edge from vertex that was discovered **most** recently.
- Use stack

- Take edge from vertex that is **closest** to source.
- Use priority queue

Directed Acyclic Graphs (DAG)

Topological Order

- not every directed graph has a topological ordering
- topological ordering is not unique

Properties:

1. Sequential total ordering of all nodes
2. Edges only point forward (no backtrack)

Topological Sort (Post-Order DFS)

- Process each node when it is last visited

⌨️ Pseudo-Code for Topological Sort

```
DFS(Node[] nodeList) {
  boolean[] visited = new boolean[nodeList.length];
  Arrays.fill(visited, false);

  for (start = i; start < nodeList.length; start++) {
    if (!visited[start]){
      visited[start] = true;
      DFS-visit(nodeList, visited, start);
      schedule.prepend(v);
    }
  }
}

DFS-visit(Node[] nodeList, boolean[] visited, int startId){
  for (Integer v : nodeList[startId].nbrList) {
    if (!visited[v]) {
      visited[v] = true;
      DFS-visit(nodeList, visited, v);
      schedule.prepend(v);
    }
  }
}


// input: directed acyclic graph (DAG)
// output: total ordering of nodes, where all edges point forwards
```

⌛ Time Complexity

$T(n) = O(V + E) = O(E)$

- E ≥ V

Kahn's Algorithm

Repeat:

- S = nodes in G that have no incoming edges.
- Add nodes in S to the topological order
- Remove all edges adjacent to nodes in S
- Remove nodes in S from the graph

Implementation ("augmented BFS")

- Maintain all nodes in priority queue (AVL tree)
- Keys are incoming edges.
- Remove min-degree node u.
- For each outgoing edge (u,v): *decrease-key of v by 1.*

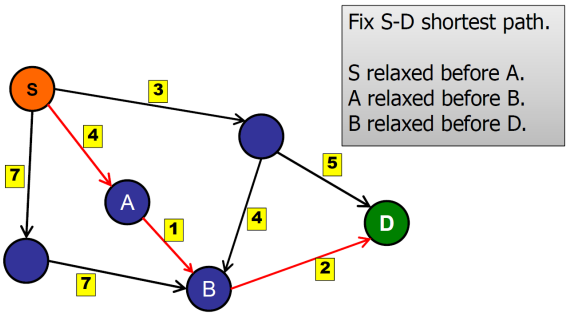
⌛ Time Complexity

$T(n) = O(E \cdot log(V))$

- can be implemented in O(V+E)

Shortest Path on a DAG

- relax the nodes in DFS post-order, i.e. topological order
- Path relaxed in-order, so distance is correct after relaxation.

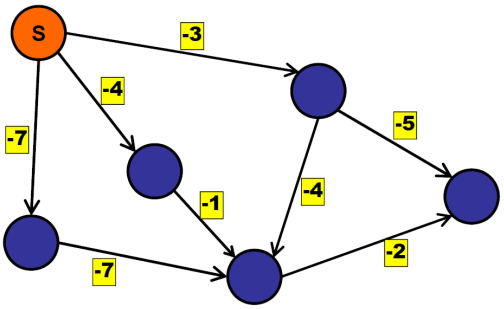


⌛ Time Complexity

$T(n) = O(E)$

Longest Paths

- Acyclic Graph: negate the edges
  - shortest path in negated = longest path in regular
  - or modify relax function
- General (cyclic graphs)
  - CANNOT negate the edges (may lead to negative cyclic paths)
  - NP-hard
  - Reduction from Hamiltonian Path:
    - If you could find the longest simple path, then you could decide if there is a path that visits every vertex.
    - Any polynomial time algorithm for longest path thus implies a polynomial time algorithm for HAMPATH



Longest path & Shortest path

- the longest path problem commonly means finding the longest simple path (no repeating vertices): points 3&4
- the shortest path problem focuses on finding the shortest (simple or non-simple)

1. longest path in a graph without cycles has optimal substructure and so does the shortest path
2. longest path without positive cycles has optimal substructure and so does a shortest path without negative cycles
3. longest path with positive cycles and shortest path with negative cycles do not exist due to infinite loops --> look at simple paths (no repeating vertices)
4. longest path with positive cycles and shortest path with negative cycles are both NP-hard (cannot be solved in polynomial time)

DP solves (1) in O(V+E).

Bellman-Ford solves (2) in O(VE)

Dijkstra/Topo sort help in certain subcases of (2), e.g. DAGs (just negate the edges and run Dijkstra)

Redefining Trees

(Undirected) Tree

- A graph with no cycles is an (undirected) tree.

Rooted Tree

- A tree with a special designated root note.

Previous (recursive) definition of a tree

- A node with zero, one, or more sub-trees, i.e. a rooted tree

Shortest Path: Tree

Assumptions:

- weighted edges
- positive/negative edges
- undirected tree

Basic idea:

- DFS/BFS
- Relax each edge the first time you see it
- |E| in a tree = |V-1| = O(V)
- $T(n) = O(V)$

Shortest Path Summary

Aa Graph Type	Algorithm	T(n)
No negative weight cycles	Bellman-Ford	O(VE)
No negative edges	Dijkstra	O(E log V)

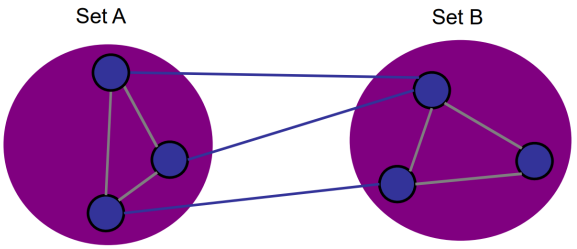
Graph Type	Algorithm	T(n)
No directed cycles	TopoSort + Relax	O(E)
No cycles (trees)	DFS + Relax	O(V)
...		
Untitled		

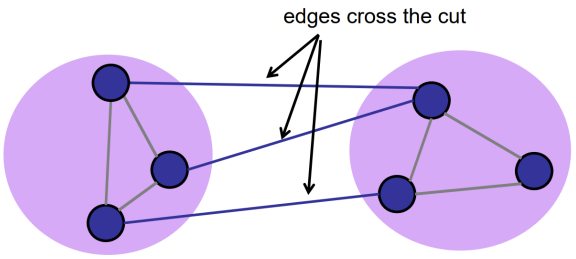
Minimum Spanning Tree (MST)

- weighted, undirected graph

Spanning Tree

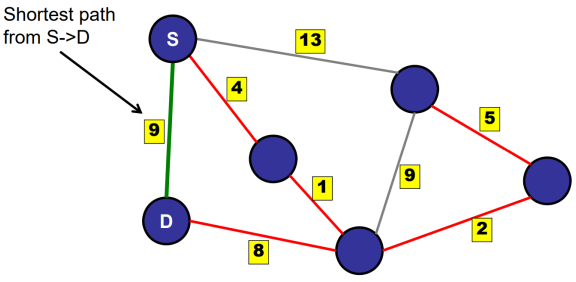
- def. acyclic subset of the edges that connects all nodes
- a *cut* of a graph G=(V, E) is a partition of the vertices V into 2 disjoint subsets.
- an edge *crosses a cut* if it has one vertex in each of the 2 sets.





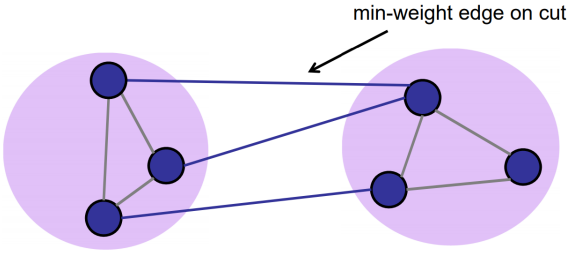
MST

- a spanning tree with min weight
- cannot use MST to find shortest path
- For every vertex, the minimum outgoing edge is always part of the MST.
- For every vertex, the max outgoing edge may or may not be part of the MST



Properties:

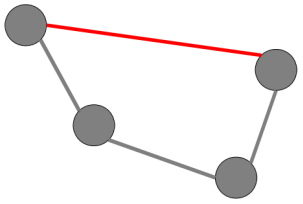
1. No cycles
2. If an MST is cut, the 2 pieces are both MSTs.
    - converse is false: joining 2 MST ≠ MST
3. Cycle Property:
    - For every cycle, the max weight edge is not in the MST.
    - False Cycle Property: For every cycle, the min weight edge may or may not be in the MST
4. Cut property
    - For every cut/partition of the nodes, the min weight edge that crosses the cut is in the MST.



Generic MST Algorithm

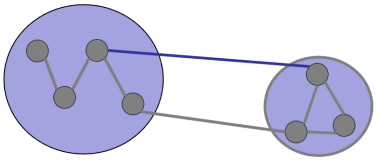
Red rule:

- If C is a cycle with no red arcs, then colour the max-weight edge in C red.



Blue rule:

- If D is a cut with no blue arcs, then colour the min-weight edge in D blue.



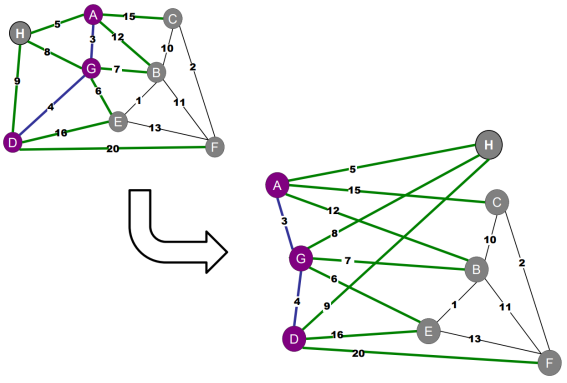
Greedy Algorithm

- Repeat:
  - apply red rule or blue rule to an arbitrary edge
  - until no more edges can be coloured
- On termination, the blue edges form a MST
  - every cycle has a red edge.
  - no blue cycles
  - blue edges form a tree (otherwise there is a cut with no blue edge)
  - every edge is coloured
  - every blue edge is in MST (property 4)

Prim's Algorithm

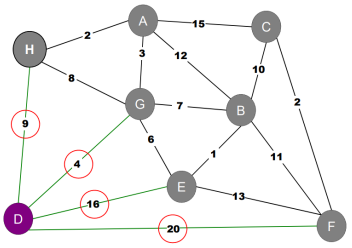
Basic idea:

- S: set of nodes connected by blue edges
- Initially: S= {A}
- Repeat:
  - identify cut: {S, V - S}
  - find min weight edge on a cut using Priority queue (blue rule)
  - add new node to S



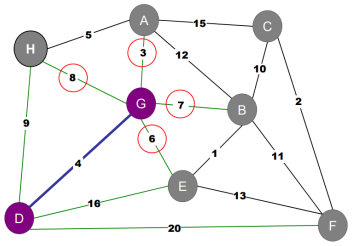
Prim's Example

Vertex	Weight
G	4
H	9
E	16
F	20



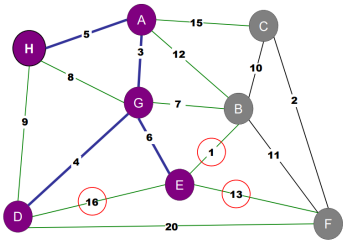
Prim's Example

Vertex	Weight
A	3
E	16->6
B	7
H	9->8
F	20



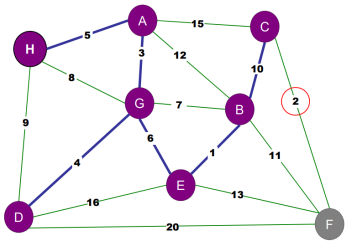
Prim's Example

Vertex	Weight
B	7->1
C	15
F	20->13



Prim's Example

Vertex	Weight
F	11->2



⌨️ Pseudo-code for Prim's Algorithm

```
// Initialize priority queue
PriorityQueue pq = new PriorityQueue();
for (Node v : G.V()) {
  pq.insert(v, INFTY); // node, weight
}
pq.decreaseKey(start, 0);

// Initialize set S - to store visited nodes
HashSet<Node> S = new HashSet<Node>();
S.put(start);
```

```
// Initialize parent hash table - to construct MST
HashMap<Node,Node> parent = new HashMap<Node,Node>();
parent.put(start, null);

while (!pq.isEmpty()){
    Node v = pq.deleteMin();
    S.put(v);
    for each (Edge e : v.edgeList()){
      Node w = e.otherNode(v);
      if (!S.get(w) && e.weight < pq.lookup(w)) {
      // only decrease key when the new weight is smaller
        pq.decreaseKey(w, e.getWeight());
        parent.put(w, v);
    }
  }
}
```

⌛ Time Complexity

$T(n) = O(E \cdot log(V))$

- using a binary heap (priority queue)

📁 Space Complexity

$S(n) = O(V)$

- Binary Heap
- challenging to achieve

Kruskal's Algorithm

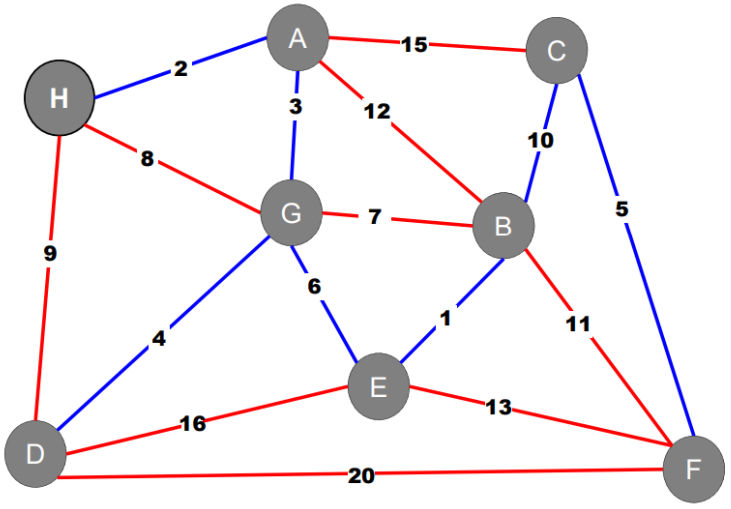
Basic idea:

- Sort the edges by weight from smallest to biggest
- Consider edges in ascending order
  - if both endpoints are in the same blue tree, then colour the edge red (must be the heaviest edge in the cycle)
  - otherwise, colour the edge blue

Data Structure

- Union-Find
- Connect 2 nodes if they are the same blue tree

Kruskal's Example



Weight	Edge
1	(E,B)
2	(C,F)
3	(A,G)
4	(D,G)
5	(C,F)
6	(E,G)
7	(B,G)
8	(G,H)
9	(D,G)
10	(B,C)
11	(B,F)
12	(A,B)
13	(E,F)
15	(A,C)
16	(D,E)
20	(D,F)

⌨️Pseudo-code for Kruskal's Algorithm

```
// Sort edges and initialize
Edge[] sortedEdges = sort(G.E());
ArrayList<Edge> mstEdges = new ArrayList<Edge>();
UnionFind uf = new UnionFind(G.V());

// Iterate through all the edges, in order
for (int i=0; i<sortedEdges.length; i++) {
  Edge e = sortedEdges[i]; // get edge
  Node v = e.one(); // get node endpoints
  Node w = e.two();

  if (!uf.find(v,w)) { // in the same tree?
    mstEdges.add(e); // save edge
    uf.union(v,w); // combine trees
  }
}
```

⌛ Time Complexity

$T(n) = O(E \cdot log(V))$

- Sorting: O(E log E) = O(E log V)
- For E edges:
  - Find: O(α(n)) or O(log V)
  - Union: O(α(n)) or O(log V)

📁 Space Complexity

$S(n) = O(V)$

- Union Find with path compression

Boruvka's Algorithm

- Parallelizable
- flexible

Basic ideas

Initially:

- create n connected components, one for each node in the graph
- for each node: store a component identifier → **O(V)**

One "Boruvka" Step: **O(V+E)**

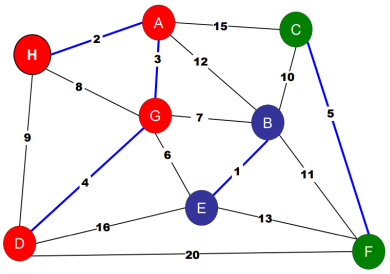
- for each connected component, search for the minimum weight outgoing edges.
  - DFS/BFS: check if edge connects two components → **O(V+E)**
  - remember min cost edge connected to each component
- add selected edges
- merge connected components.
  - Scan every node → **O(V)**
  - compute new component ids
  - update components ids
  - mark added edges

Component	1	2	3
Min cost edge	(G,E), 6	(G,E), 6	(B,C), 10
To be merged	1 and 2	1 and 2	2 and 3
New ID:	1	1	1

In each "Boruvka" Step O(V+E)

- Assume k components initially
- at least k/2 edges added
  - each component adds 1 edge
  - some chose the same edge
  - each edge is chosen by at most 2 different components → k/2
- at leas k/2 components merge → each edge merges 2 components
- End: at most k/2 components remain

Boruvka's Example

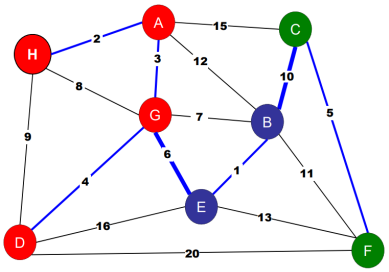


Weight	Edge
1	(E,B)
2	(C,F)
3	(A,G)
4	(D,G)
5	(C,F)
6	(E,G)
7	(B,G)
8	(G,H)
9	(D,G)
10	(B,C)
11	(B,F)
12	(A,B)
13	(E,F)
15	(A,C)
16	(D,E)
20	(D,F)

Look at connected components…

At most n/2 connected components.

Boruvka's Example



Weight	Edge
1	(E,B)
2	(C,F)
3	(A,G)
4	(D,G)
5	(C,F)
6	(E,G)
7	(B,G)
8	(G,H)
9	(D,G)
10	(B,C)
11	(B,F)
12	(A,B)
13	(E,F)
15	(A,C)
16	(D,E)
20	(D,F)

Repeat: for every connected components, add minimum outgoing edge.

Summary:

- Initially: n components
- At each step: k components → k/2 components
- Termination: 1 component
- Conclusion: at most O(log V) Boruvka steps

Parallelism:

- each component can search for its own minimum weight outgoing edges → many edges found in parallel
- merging requires coordination between components → key bottleneck
- lots of parallelism can be used in searching for edges and updating the connected components

⌛ Time Complexity

$T(n) = ((E + V) \cdot log(V)) = O(E \cdot log(V))$

Prim's Variants

All edges have weights from {1, ..., 10}

Idea:

- use an array of size 10 as a Priority Queue: `A[j]` holds a linked list of **nodes** of weight `j`

- `decreaseKey`: move node to new linked list

Implement PQ (note: does not work for Dijkstra)

- use an array of size 10 to implement
- `insert`: put node in correct list O(V)
- `remove`: lookup node (e.g. in hash table) and remove from linked list O(V)
- `extractMin`: remove from the minimum bucket
- `decreaseKey`: lookup node(in hash table) and move to correct linked list O(E) - linear search through LL
- $T(n) = O(V + E) = O(E)$

Kruskal's Variants

All edges have weights from {1, ..., 10}

Idea:

- use an array of size 10 to sort: `A[j]` holds a linked list of **edges** of weight `j`
- Putting edges in an array of linked lists O(E)
- Iterate over all edges in ascending order O(E)
- For each edge:
  - check whether to add an edge O(α)
  - union two components O(α)
- Total $T(n) = \alpha E$

Directed MST

⇒ Rooted spanning tree

- each node is reachable on a path from the root
- no cycles

Problems:

- cut & cycle properties do not hold
- generic MST algo does not work, i.e. Prim, Kruskal, Boruvska algo do not work

Special Case: DAG with one "root" (one node with no incoming edge)

- for every node except the root: add min weight incoming edge
- Observations:
  - no cycles (acyclic)
  - each edge is chosen only once → Tree: V nodes, (V-1) edges, no cycles
  - each node has to have ≥ 1 incoming edge in the MST
- $T(n) = O(E)$

Maximum Spanning Tree

- adding/subtracting weights do not affect MST (unlike SSSP)
- for MST with -ve weight
  - no need to reweight: only relative weights matter (comparison between the known edges)
  - can add a big enough value to each edge to become positive but unnecessary
- hence, to find MaxST:

1. negative the edges (or run reverse Kruskal)
2. run MST algo
3. "most negative" = max

MST Algo Summary

Name	DS	T(n)	Column
Prim's	Priority Queue	O(E log V)	
Kruskal's	Union-Find	O(E log V)	
Boruvka's		O(E log V)	
BFS/DFS	(same weight)		If weight = k, cost of MST = k(V-1) - (V-1) edges in MST
Chazelle (2000) (not in syllabus)		O(m α(m;n))	
Untitled			

Steiner Tree

Find MST of a subset of the vertices

- use the sub-graph (with required nodes)
- use other nodes necessary to give a smaller weight (Steiner nodes)

Steiner MST approximation algorithms

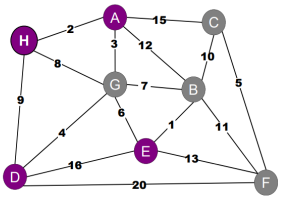
- OPT(G) = min cost Steiner Tree
- T = output of Steiner MST
- T < 2*OPT(G)

Step 1: For every pair of required vertices `(v,w)`, calculate the shortest path from (v to w)

- use Dijkstra V times
- or All-Pairs-Shortest-Paths (not covered yet)

Example: Step 1

Shortest Paths:
(A,H) = 2
(A,D) = 7
(A,E) = 9
(H,D) = 9
(H,E) = 11
(D,E) = 10



Step 2: Construct new graph on required nodes

- V = required nodes
- E = shortest path distances

Step 3: Run MST on new graph

- use Prim's or Kruskal's or Boruvka's
- MST gives edges on a new graph

Step 4: Map new edges back to original graph

- use the shortest path discovered in Step 1
- add these edges to Steiner MST
- remove duplicates

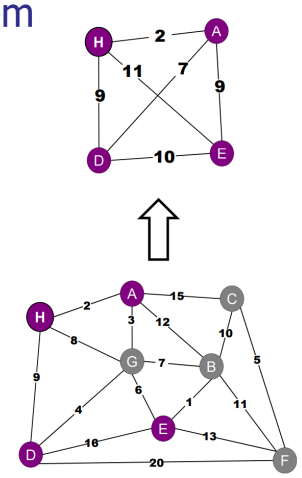
NOTE: this does not guarantee optimal Steiner Tree

Steiner Tree Problem

Example: Step 2

Shortest Paths:

(A,H) = 2
(A,D) = 7
(A,E) = 9
(H,D) = 9
(H,E) = 11
(D,E) = 10

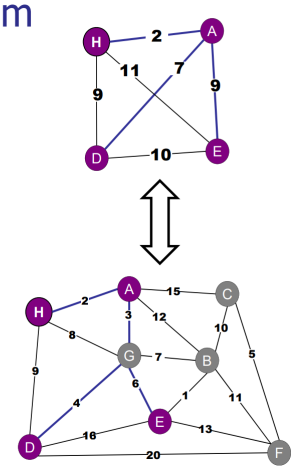


Steiner Tree Problem

Example: Step 4

Shortest Paths:

(A,H) = 2
(A,D) = 7
(A,E) = 9
(H,D) = 9
(H,E) = 11
(D,E) = 10



Proof

1. Let O be OPT tree.
   Let T be SteinerMST tree.
2. Let D = DFS on O.
   cost(D) = 2*OPT.
3. D = {H, A, D, G, E, G, A, H}
4. cost(D) = w(H,A) + w(A,G) + … + w(A,H)
5. Skip Steiner Nodes: D' = {H, A, D, E, A, H}
   - D' = set of edges on shortest path graph = spanning subgraph of shortest path graph
   - cost(D') = cost of traversing shortest path ≤ cost (D) ≤ 2*OPT
6. D' spans shortest path graph
7. cost(T) = cost(MST) < cost(D') ≤ 2*OPT

W11L1 Heap (a.k.a. Binary Heap/Max Heap)

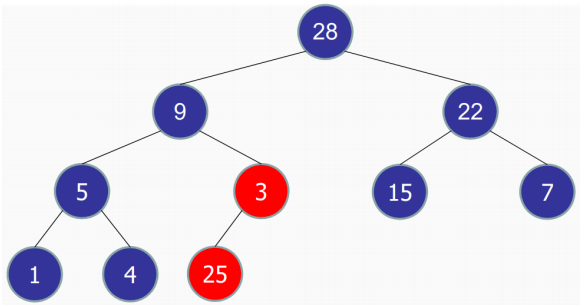
- implements a Max Priority Queue
- Maintain a set of prioritised objects

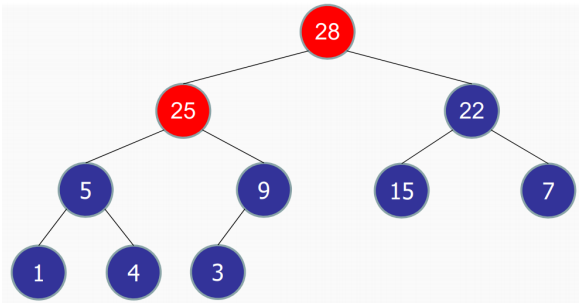
- store items in a tree (but itself is not a BST)
  - biggest items at root
  - smallest items at leaves
  - (duplicates are allowed)
- Properties:
  1. Heap Ordering `priority[parent] ≥ priority[child]`
  2. Complete binary tree
     - every level is full, except possibly the last (leaves)
     - all nodes are as far left as possible
- Max height of heap: $\lfloor log_2 n \rfloor = O(log_2 n)$

Insert

1. insert at the leaf level
2. bubble up by comparing its priority with its parent node

$T(n) = O(log(n))$





⌨️ Pseudo-Code for Insert

```
bubbleUp(Node v) {
  while (v != null) {
    if (priority(v) > priority(parent(v)))
      swap(v, parent(v));
    else return;

    v = parent(v);
  }
}

insert(Priority p, Key k) {
  Node v = completeTree.insert(p,k);
  bubbleUp(v);
}
```

increaseKey

- bubble up

$T(n) = O(log(n))$

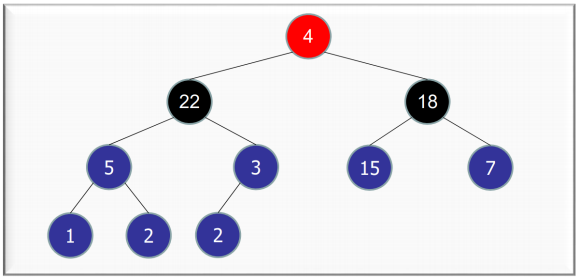
decreaseKey

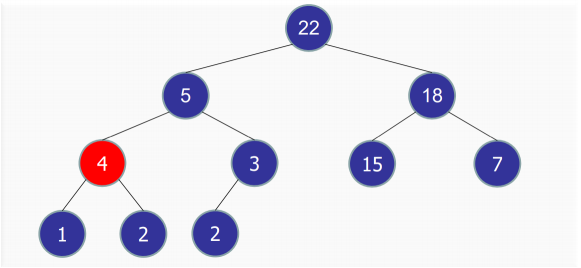
1. Update the priority
2. bubble down

$T(n) = O(log(n))$

`decreaseKey(28 → 4)`:

- Step 1: Update the priority
- Step 2: bubbleDown(4)





⌨️ Pseudo-Code for Decrease Key

```
bubbleDown(Node v)
  while (!leaf(v)) {
    leftP = priority(left(v));
    rightP = priority(right(v));
    maxP = max(leftP, rightP, priority(v));

    if (leftP == max) {
      swap(v, left(v));
      v = left(v); }
    else if (rightP == max) {
      swap(v, right(v));
      v = right(v); }
    else return;
}
```

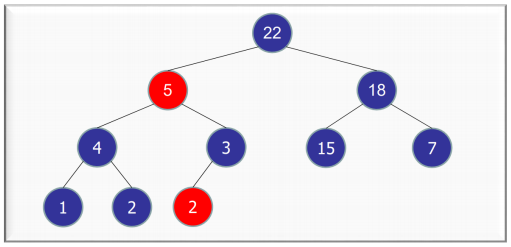
delete

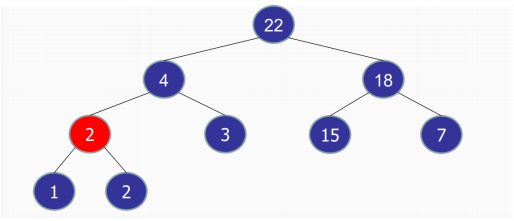
1. swap node with the last one in the right subtree
2. remove the last node (node to be deleted) in the right subtree
3. bubble down the new subtree root

$T(n) = O(log(n))$

`delete(5)`:

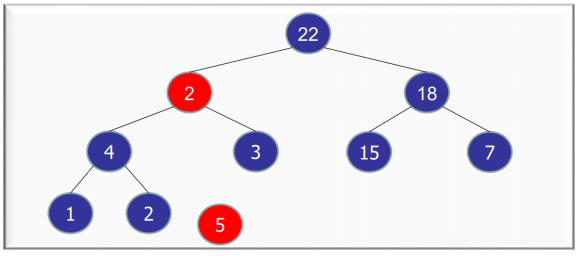
- swap(5, last())





`delete(5)`:

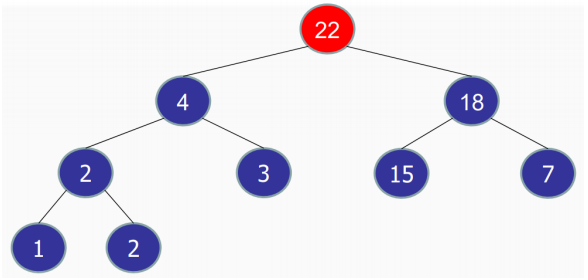
- swap(5, last())
- remove(last())
- bubbleDown(2)

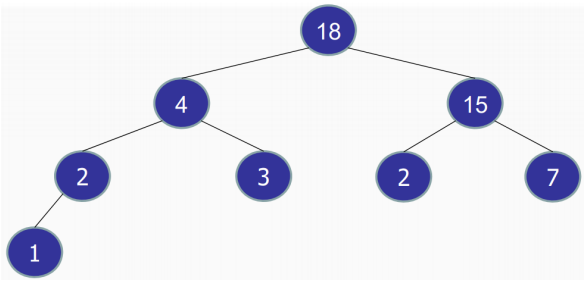


extractMax

1. node v = root;
2. delete(root)

$T(n) = O(log(n))$





Heap v.s. AVL

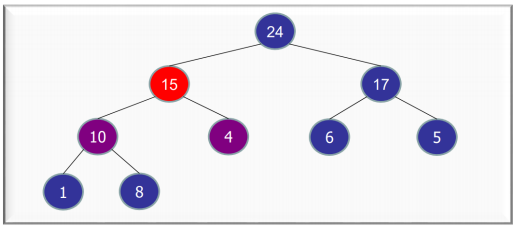
- same asymptotic cost
- faster real cost (no constant factors)
- simpler: no rotations
- slightly better concurrency

Heap Sort

Store Tree in an Array

- map each node in a complete binary tree into a slot in an array
- cannot store AVL in an array ⇒ expensive when rotation occurs
- index of root of left subtree ⇒ left(x) = 2x+1
- index of root of right subtree ⇒ right(x) = 2x + 2
- index of parent ⇒ parent(x) = $\lfloor \frac{x-1}{2} \rfloor$

array slot	0	1	2	3	4	5	6	7	8
priority	24	15	17	10	4	6	5	1	8



HeapSort

Unsorted list:

array slot	0	1	2	3	4	5	6	7	8
key	6	4	5	3	10	17	24	1	8

Step 1. Unsorted list ➔ Heap

array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	5	1	3

Step 2. Heap ➔ Sorted list:

array slot	0	1	2	3	4	5	6	7	8
key	1	3	4	5	6	8	10	17	24

⌨️ Pseudo-Code for Heap Sort

```
// Heapify: unsorted list -> heap
// int[] A = array of unsorted integers
/* version 1 O(n log n) */
for (int i=0; i<n; i++) {
  int value = A[i];
  A[i] = EMPTY:
  heapInsert(value, A, 0, i); // O(log n)
}

/* version 2 - recursion O(n)*/
for (int i=(n-1); i>=0; i--) {
  bubbleDown(i, A); // O(height) <= O(log n)
}
/*
cost(bubbleDown) = height
> n/2 nodes are leaves (height = 0) -> most nodes have small heights
cost of building a heap = O(n)
*/

// heap array -> sorted list O(n log n)
// int[] A = array stored as a heap
for (int i=(n-1); i>=0; i--) {
  int value = extractMax(A); // O(log n)
  A[i] = value;
}
```

⌛ Time Complexity for Heap Sort

1. Unsorted list → heap (heapify):
   - Version 1: $T(n) = O(n \cdot log(n))$
   - Version 2 (recursion): $T(n) = O(n)$

Height	0	1	2	3	...	⌊ log(n) ⌋
Number	⌈n/2⌉	⌈n/4⌉	⌈n/8⌉	⌈n/16⌉	...	1

$$\sum_{h=0}^{h=\log(n)} \frac{n}{2^h} O(h) \leq cn\left(\frac{1}{2} + \frac{2}{2^2} + \frac{3}{2^3} + \frac{4}{2^4} + \dots\right)$$

$$\leq cn\left(\frac{1/2}{(1-1/2)^2}\right) \leq 2 \cdot O(n)$$

2. Heap arr → sorted list: $T(n) = O(n \cdot log(n))$
   - speed: merge ≤ heap ≤ quick
   - deterministic: always completes in O(n log n)

📁 Space Complexity for Heap Sort

$S(n) = O(n)$

- in-place

💣 Stability

- unstable

Disjoint Set (Union-Find)

Application: Dynamic Connectivity

Mazes

Pre-process:

- identify connected components (maximal set of mutually connected objects).
- Label each location with its component no.
- Transitivity: If p is connected to q and if q is connected to r, then p is connected to r.

`isConnected(y, z)` ⇒ find

- returns `true` if A&B are in the same connected component

`destroyWall(x, y)` ⇒ union

- remove walls from the maze

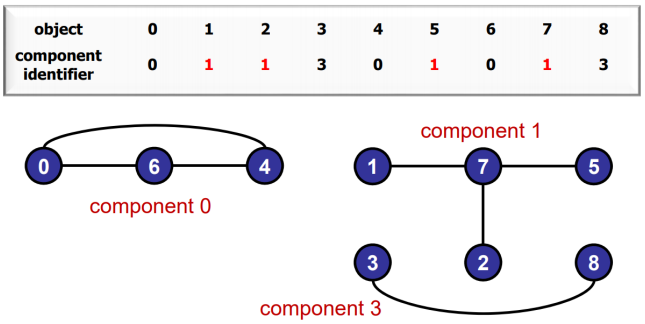
Disjoint Set

Aa public interface	DisjointSet<Key>	Function
Untitled	`DisjointSet(int N)`	Constructor: N objects
boolean	`findKey(Key p, Key q)`	check if p & q are in the same set
void	`union(Key p, Key q)`	replace sets containing p & q with their union
Untitled		

Quick Find

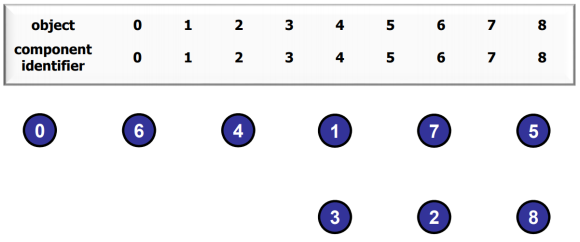
Data Structure

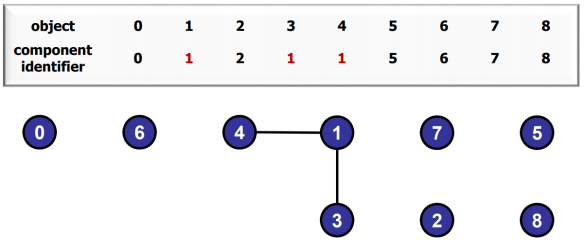
- `int[] componentId`
- 2 objects are connected if they have the same component identifier
- if objects are not integers, use **hash table + open addressing** to convert them into integers

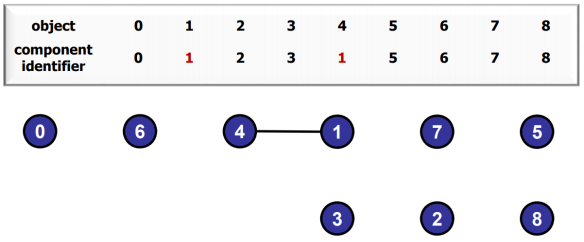


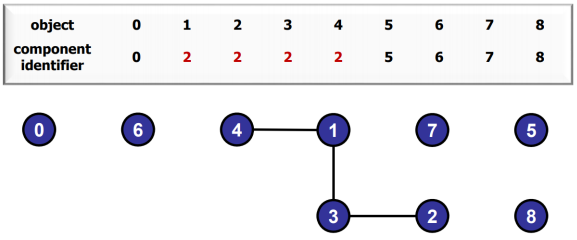
⌨️ Pseudo-Code for find & union

```
find(int p, int q) // O(1)
  return (componentId[p] == componentId[q]);

union(int p, int q) // connect p & q together O(n)
  updateComponent = componentId[q]
  for (int i=0; i<componentId.length; i++)
    if (componentId[i] == updateComponent)
    componentId[i] = componentId[p];
```

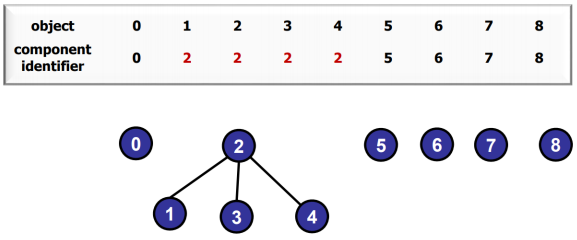








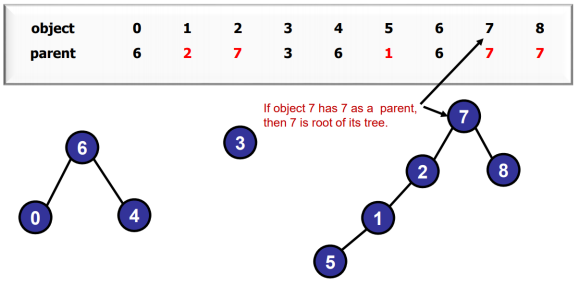
Flat trees:



Quick Union

Data Structure

- `int[] parent`
- two objects are connected if they are part of the same tree



⌨️ Pseudo-Code for find & union

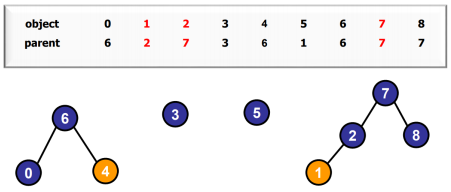
```
find(int p, int q) // O(n)
  while (parent[p] != p) p = parent[p];
  while (parent[q] != q) q = parent[q];
  return (p == q); // check if they have the same parent

unweightedUnion(int p, int q) // O(n)
  while (parent[p] != p) p = parent[p];
  while (parent[q] != q) q= parent[q];
  parent[p] = q;
```

Example: find(4, 1)

4 → 6 → 6

1 → 2 → 7 → 7

return (6 == 7) → **false**

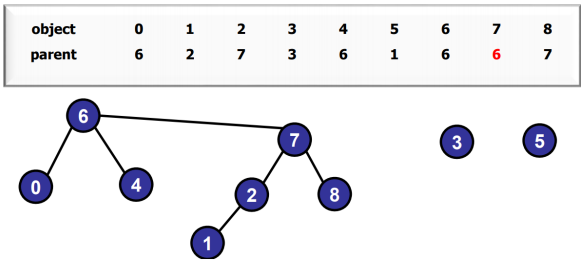


Example: union(1, 4)

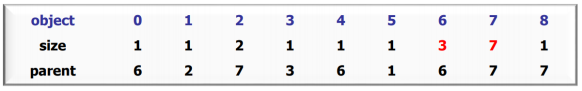
4 → 6 → 6

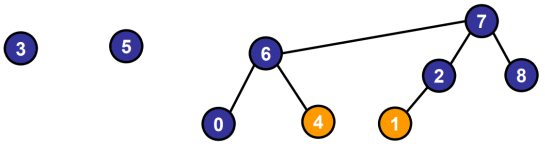
1 → 2 → 7 → 7

parent[7] = 6;



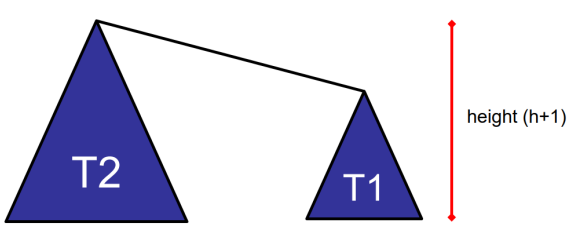
union(1, 4)

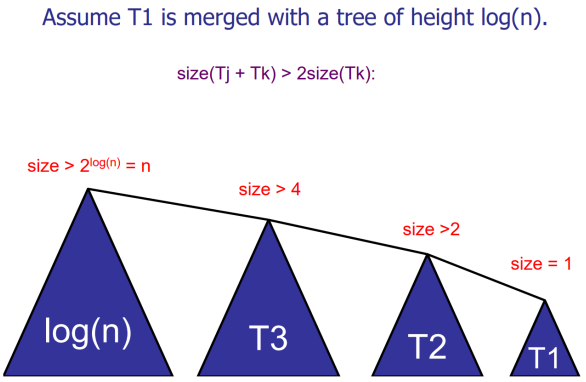




Weighted Union

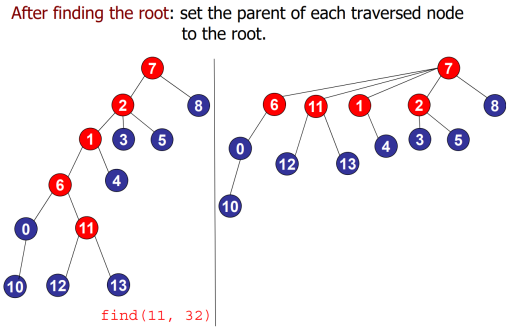
- Tree T1 is merged with Tree T2: Only if: size(T2) ≥ size(T1) ⇒ T1 is one level deeper
- size(T1 + T2) > 2 * size(T1)






```
weightedUnion(int p, int q) // optimize tree height = O(log n)
  while (parent[p] !=p) p = parent[p];
  while (parent[q] !=q) q = parent[q];
  if (size[p] > size[q]) {
    parent[q] = p; // Link q to p
    size[p] = size[p] + size[q];
  } else {
    parent[p] = q; // Link p to q
    size[q] = size[p] + size[q];
  }
```

Path Compression



⌨️ Pseudo-Code

```
/* Version 1 */
findRoot(int p) {
  root = p;
  while (parent[root] != root) root = parent[root];
  while (parent[p] != p) {
    temp = parent[p];
    parent[p] = root;
    p = temp;
  }
  return root;
}

/* Version 2 */
findRoot(int p) {
  root = p;
  while (parent[root] != root) { // parent of root is itself
    parent[root] = parent[parent[root]];
    root = parent[root];
    // or make every other node in the path point to its grandparent
  }
```

```
  return root;
}
```

Weight Union with Path Compression

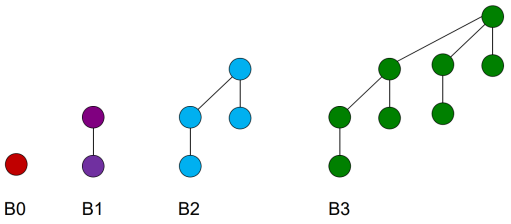
Theorem: [Tarjan 1975]

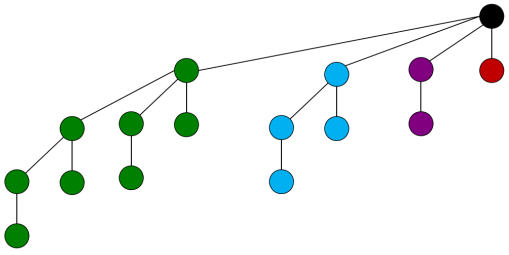
Starting from empty, any sequence of m union/find operations on n objects takes: $O(n + m\alpha(m, n))$ time.

Inverse Ackermann function: always ≤ 5 in this universe.

n	α(n, n)
4	0
8	1
32	2
8,192	3
2^{65533}	4

Binomial Tree





size(Bk) = $\Theta(2^k)$

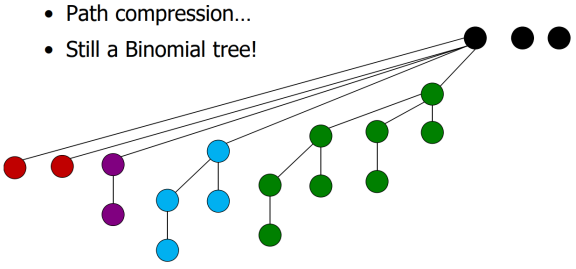
height(Bk) = k-1

Step 1: Build Binomial tree using union operations.

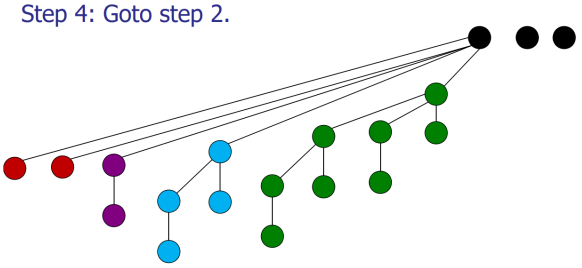
Step 2: Union: create new root [O(1)]

Step 3: Find deepest leaf [O(log n)]

- Path compression…
- Still a Binomial tree!



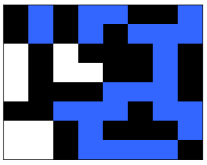
Step 4: Goto step 2.



Application: Percolation System

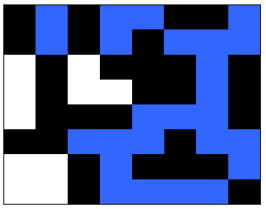
Physical system:

- Sharp threshold p*: (Via simulation: p* = 0.592746)
  - p>p* : percolates
  - p<p* : does not percolate



Simulation:

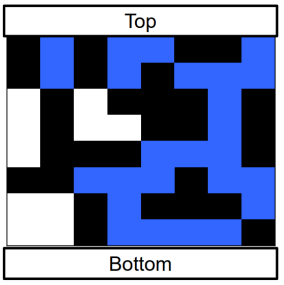
- Add each site to union-find object.
- Connect all open sites to neighboring open sites.
- For every pair on the top/bottom row, check if connected.



Percolation

Slightly better:

- Create virtual top and bottom.
- Only check if top and bottom are connected.



Union-Find Summary

Name (per operation)	find	union	Tree
Quick Find	O(1)	O(n)	flat
Quick Union	O(n)	O(n)	unbalanced
Weighted Union	O(log n)	O(log n)	
Weighted Union w Path Compression	α(m;n)	α(m;n)	flat
Path Compression (avg)	O(log n)	O(log n)	
Path Compression (worst)	O(n)	O(n)	one-liner case
Untitled			

Priority Queue

Interface	IPriorityQueue<Key, Priority>	Function	T(n)
void	`insert(Key k, Priority p)`	insert k with priority p	- sorted arr O(n): find insertion location → move everything over - unsorted arr O(1): add to the end - AVL (indexed by priority) O(log n)
Data	`extractMin()` `extractMax()`	remove key with min priority	- sorted arr O(1): order has been maintained - unsorted arr O(n): search for min elem, remove and move everything over - AVL O(log n): find max and delete

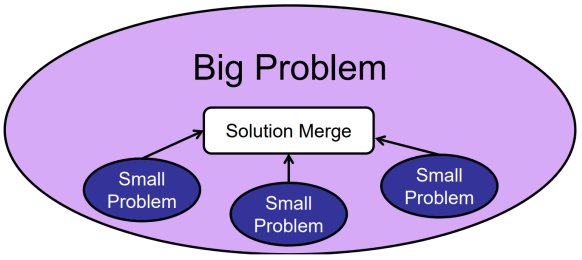
Aa Interface	IPriorityQueue<Key, Priority>	Function	T(n)
void	`decreasesKey(Key k, Priority p)`	reduce the priority of key k to priority p	search → remove → re-insert
boolean	`contains(Key k)`	check if priority queue contains key k	
boolean	`isEmpty()`	check if priority queue is empty	
Untitled		Note: assume data items are unique	
Hash table		maps priorities to arr alots or nodes in tree	
Untitled			

W12L2 Dynamic Programming

Basics

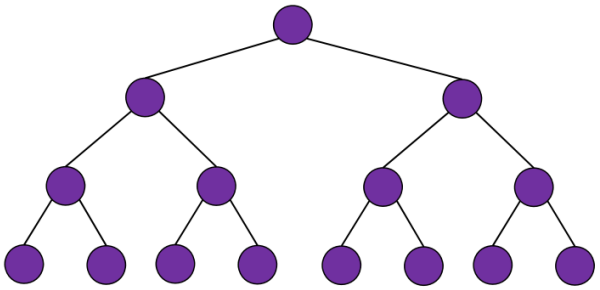
Optimal sub-structure

- A given problems has Optimal Substructure Property if optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.
- e.g. some Greedy Algo (always select the largest / smallest)
  - SSSP
  - MST
- e.g. Divide-and-Conquer
  - merge (sort)
  - reverse string



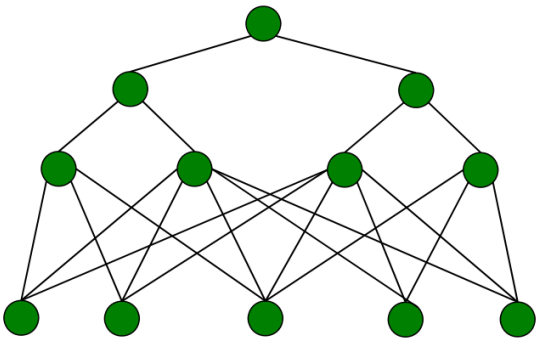
Contrast: Both have optimal substructure

No overlapping subproblems



Divide-and-Conquer

Overlapping subproblems



Dynamic Programming

Overlapping sub-problems: the same smaller problem is used to solve multiple diff bigger problems

Bottom-up dynamic programming

Step 1: Solve the smallest problems

Step 2: Combine smaller problems

Step 3: Solve the root problem

Top-down dynamic programming

Step 1: Start at root and recurse

Step 2: Recurse

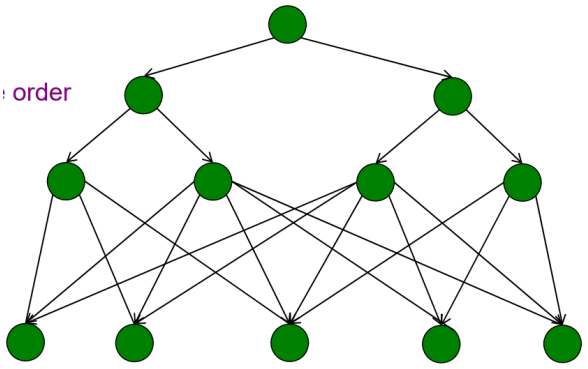
Step 3: Solve and memoize (use table/arr?).

Only compute each solution once

DAG + Topological sort

Step 1: Topologically sort DAG

Step 2: Solve problems in reverse order



Longest Increasing Subsequence (LIS)

Input: sequence of integers

- e.g. {8, 3, 6, 4, 5, 7, 7}

Output: length of increasing sequence

- e.g. {4, 5, 7}

Goal: output max length

- e.g. {3, 4, 5, 7} - no need be consecutive

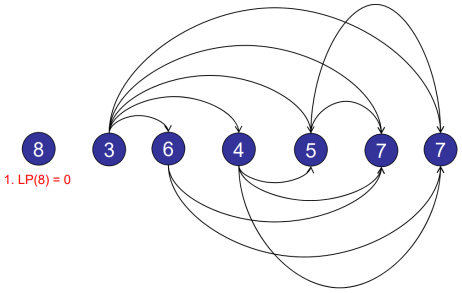
DAG Solution

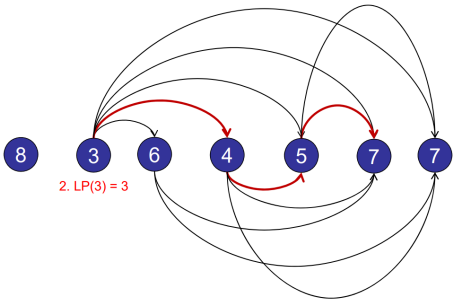
Step 1: topo sort - do nth, alr sorted in topo order

Step 2: calculate the longest paths. `LIS = max(LP)+1`

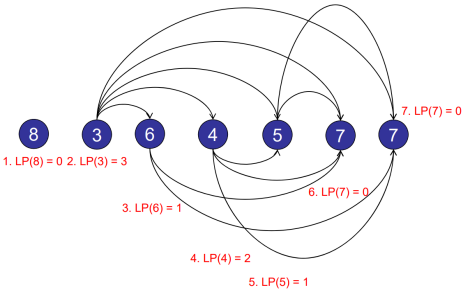
$T(n) = O(n^3)$

- longest path BFS/DFS: $O(V + E) = O(n^2)$
  - edges can be at most $O(n^2)$ e.g. complete graphs
- run longest path n times: $O(n^3)$





relax edges in topo sort order



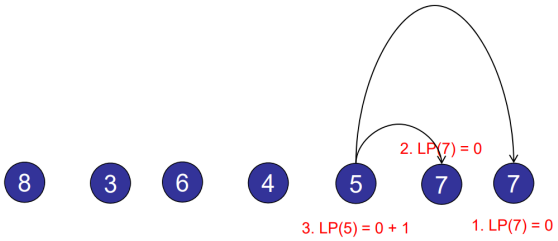
Bottom-up DP

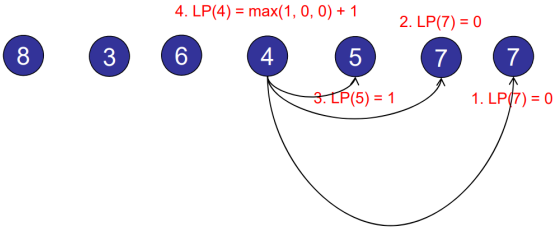
Step 1: start with the smallest sub-problem:

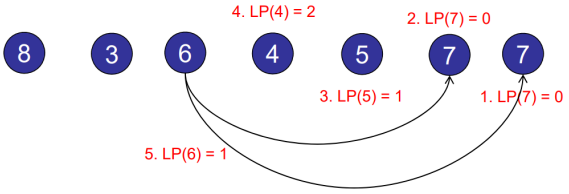
LP(7) = 0

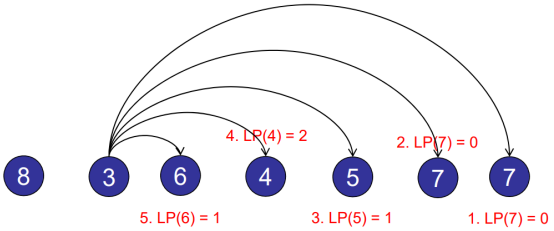
Step 2: combine sub-problems

- e.g. LP(5)
- examine each outgoing edge
- find the maximum
- add 1









6. LP(3) = max(1, 2, 1, 0, 0) + 1 = 3

Optimised Algo with Memoisation

Input: array `A[1...n]`

Greedy sub-problems: `S[i] = LIS(A[i...n])` starting at `A[i]`

- $S[n] = 0$
- $S[i] = (max_{(i,j) \in E} S[j]) + 1$

Greedy sub-problems: `S[i] = LIS(A[i...n])` ending at `A[i]`

- $S[1] = 0$
- $S[i] = (max_{(j<i, A[j]<A[i])} S[j]) + 1$

$T(n) = O(n^2)$

e.g. {8, 3, 6, 4, 5, 7, 7}

- starting at `A[i]`: S[5] = 2 → {5, 7}; S[2] = 4 → {3, 4, 5, 7}
- ending at `A[i]`: S[4] = 2 → {8, 3, 6, 4}; S[5] = 3 → {8, 3, 6, 4, 5}

```
/* Version 1: starting at A[i] */
LIS(V): // Assume graph is already topo-sorted
  int[] S = new int[V.length]; // Create memo array

  for (i=0; i<V.length; i++)
    S[i] = 0; // Initialize array to zero

  S[n-1] = 1; // Base case: node V[n-1]

  for (int v = A.length-2; v>=0; v--) {
    int max = 0; // Find maximum S for any outgoing edge
    for (Node w : v.nbrList()) { // Examine each outgoing edge
        if (S[w] > max)
          max = S[w]; // Check S[w], which we already
                      // calculated earlier.
  }
  S[v] = max + 1; // Calculate S[v] from max of outgoing edges.
}

/* Version 2: ending at A[i] */
LIS(A):
  int[] S = new int[A.length]; // Create memo array

  for (i=0; i<A.length; i++)
    S[i] = 0; // Initialize array to zero

  S[0] = 1; // Base case: length 1

  for (int i = 0; i<A.length; i++) { // O(ij)=O(n^2)
    int max = 0; // Find maximum S for any preceding node
    for (int j=0; j<i; j++) { // Examine each preceding element in the sequence
      if (A[j] < A[i]) // If A[i] is bigger than A[j]
        if (S[j] > max)
          max = S[j]; // If S[j] is longer sequence
  }
  S[i] = max + 1; // Calculate S[i] from max of preceding elements.
}
```

LIS in O(n log n)

```
import java.io.*;
import java.util.*;
import java.lang.Math;

class LIS {
  // Binary search (note boundaries in the caller)
  // A[] is ceilIndex in the caller
  static int CeilIndex(int A[], int l, int r, int key) {
    while (r - l > 1) {
      int m = l + (r - l) / 2;
      if (A[m] >= key)
        r = m;
      else
        l = m;
    }

    return r;
  }
```

```
  static int LongestIncreasingSubsequenceLength(int A[], int size) {
    // Add boundary case, when array size is one

    int[] tailTable = new int[size];
    int len; // always points empty slot

    tailTable[0] = A[0];
    len = 1;
    for (int i = 1; i < size; i++) {
      if (A[i] < tailTable[0])
        // new smallest value
        tailTable[0] = A[i];

      else if (A[i] > tailTable[len - 1])
        // A[i] wants to extend largest subsequence
        tailTable[len++] = A[i]; // eqv tailTable[len] = A[i]; len++A given problems has Optimal Substructure Property if optimal solu

      else
        // A[i] wants to be current end candidate of an existing
        // subsequence. It will replace ceil value in tailTable
        tailTable[CeilIndex(tailTable, -1, len - 1, A[i])] = A[i];
    }

    return len;
  }
}
```

Longest path & Shortest path

- the longest path problem commonly means finding the longest simple path (no repeating vertices): points 3&4
- the shortest path problem focuses on finding the shortest (simple or non-simple)

1. longest path in a graph without cycles has optimal substructure and so does the shortest path
2. longest path without positive cycles has optimal substructure and so does a shortest path without negative cycles
3. longest path with positive cycles and shortest path with negative cycles do not exist due to infinite loops --> look at simple paths (no repeating vertices)
4. longest path with positive cycles and shortest path with negative cycles are both NP-hard (cannot be solved in polynomial time)

DP solves (1) in O(V+E).

Bellman-Ford solves (2) in O(VE)

Dijkstra/Topo sort help in certain subcases of (2), e.g. DAGs (just negate the edges and run Dijkstra)

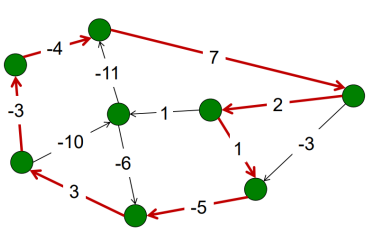
Prize Collecting

Input:

- directed, connected graph G = (V, E)
- edge weights w = prizes on each edge
- limit k: only cross at most k edges (at most k moves)

Example:

- k = 1 → 7
- k = 2 → 9
- k = 3 → 10
- k = 4 → 10
- k = 5 → 10
- ...
- k = 70 → 10
- k = 71 → 17



Cannot use Bellman-Ford

1. negate the edges
2. run Bellman-Ford
3. if estimates are still changing after n iterations, then report a cycle → terminate → cannot find a simple path

If positive weight cycle is present, these are the possible ideas to search for the longest path:

Idea 1:

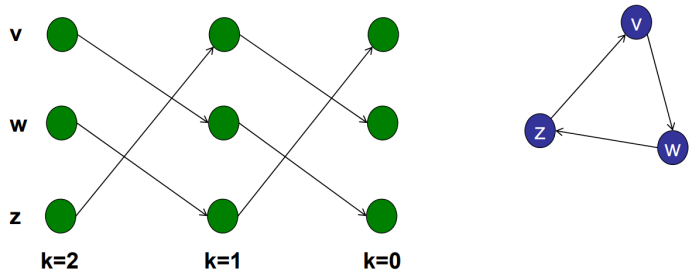
Step 1: Transform graph G into a DAG: kV nodes, kE edges

Step 2: Make k copies of every node: (v, 1), (v, 2) etc (1, 2... represents the limit k) → stacking of graphs

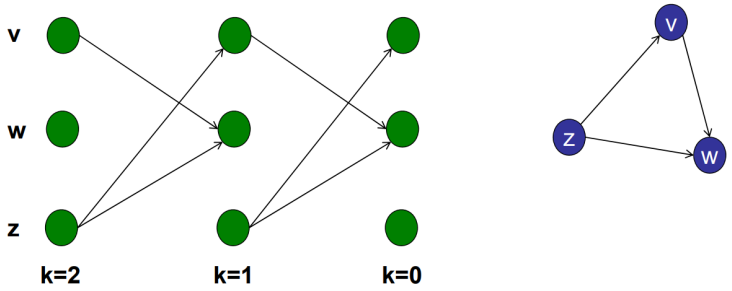
Step 3: Topo-Sort: O(kV+kE) = O(kE)

- Cannot use Dijkstra: possible

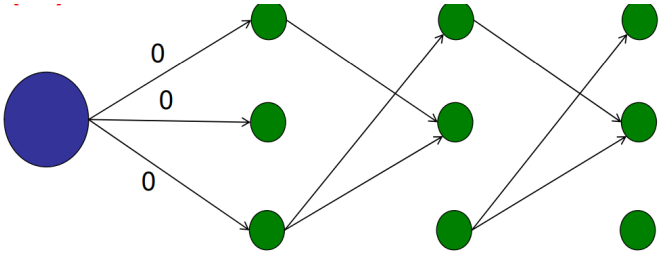
Step 4: Solve the longest path problem from each source using Topo-Sort: once per source and repeat V times → O(kVE) total



new DAG for a graph with a cycle initially



Optimised Step 4: Create super-source → only one shortest path → O(kE) total



Dynamic Programming Approach

Step 1: Define sub-problems

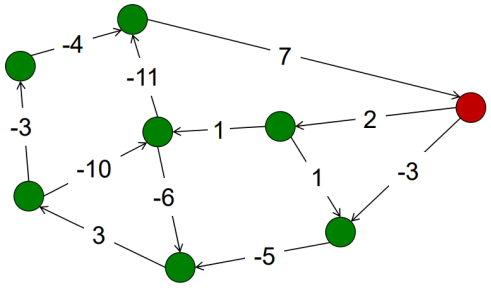
P[v, k] = max prize that can be collected starting at v and taking exactly k steps

Base case: P[v, 0] = 0

Step 2: Combine sub-problems to solve

$P[v,k] = max(P[w_1, k-1] + w(v,w_1), P[w_2, k-1] + w(v,w_2), P[w_3, k-1] + w(v,w_3), ...)$

- where $v.nbrList() = (w_1, w_2, w3, ...)$ are the neighbours of v



$P[v,1] = max(0+2, 0-3) = 2$

$P[v,2] = max(1+2, -5-3) = 3$

$P[v,3] = max(-4+2, -2-3) = -2$

⌛ Time Complexity

1. $T(n) = O(kV^2)$
   - no. of subproblems = kV
   - cost to solve each subproblem = $|v.nbrList|$ - worst case: complete graph
2. alternative perspective using info table: $T(n) = O(kE) = O(kV^2)$
   - P[v, k] = max prize that can be collected starting at v and taking exactly k steps
   - no. of rows = k
   - cost to solve each row = E

⌨️ Pseudo-Code for Lazy Prize Collecting

```
int LazyPrizeCollecting(V, E, kMax) {
  int[][] P = new int[V.length][kMax+1]; // create memo table P
  for (int i=0; i<V.length; i++) // initialize P to zero
    for (int j=0; j<kMax+1; j++)
      P[i][j] = 0;

  for (int k=1; k<kMax+1; k++) { // Solve for every value of k
    for (int v = 0; v<V.length; v++) { // For every node…
      int max = -INFTY;
      // …find max prize in next step
      for (int w : V[v].nbrList()) {
        if (P[w,k-1] + E[v,w] > max)
          max = P[w,k-1] + E[v,w];
        }
      P[v, k] = max;
    }
  }
  return maxEntry(P); // returns largest entry in P
}
```

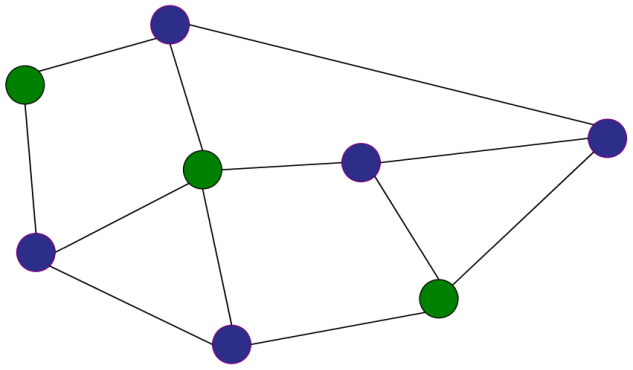
Vertex Cover

Input: Output:

Undirected, unweighted graph G = (V, E)

Set of nodes C where every edge is adjacent to at least one node in C

i.e. All the edges are covered by some vertex.



NP-Complete

- No polynomial time algorithm (unless P=NP).
- Easy 2-approximation (via matchings).
- Nothing better known.

Vertex Cover on a Tree

Input

- undirected, unweighted tree G = (V, e)
- root of tree

Output

- size of the minimum vertex cover

Subproblems:

`S[v, 0]` = size of vertex cover in subtree rooted at node v, if v is NOT covered

- If we do not cover v, then we must cover all of v's children.
- Remember: we have already solved the subproblems
- $S[v, 0] = S[w_1, 1] + S[w_2, 1] + S[w_3, 1] + \dots$
- $v.nbrList() = w1, w2, w3, \dots$

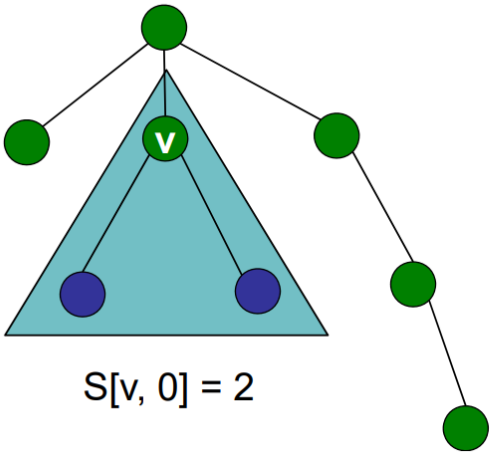
`S[v, 1]` = size of vertex cover in subtree rooted at node v, if v IS covered.

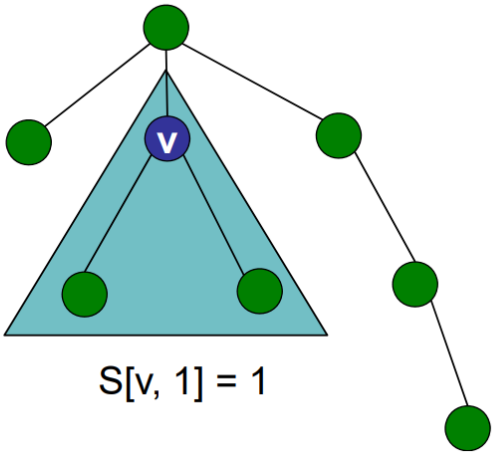
- $W_1 = min(S[w_1, 0], S[w_1, 1])$
- $W_2 = min(S[w_2, 0], S[w_2, 1])$
- $W_3 = min(S[w_3, 0], S[w_3, 1])$
- $S[v, 1] = 1 + W_1 + W_2 + W_3 + \dots$
- $v.nbrList() = w_1, w_2, w_3, \dots$

Base cases → leaf nodes

`S[leaf, 0]` = 0

`S[leaf, 1]` = 1





⌨️ Pseudo-Code for Tree Vertex Cover

```
int treeVertexCover(V){//Assume tree is ordered from root-to-leaf

int[][] S = new int[V.length][2]; // create memo table S

  for (int v=V.length-1; v>=0; v--){ //From the leaf to the root
    if (v.childList().size()==0) { // If v is a leaf…
      S[v][0] = 0;
      S[v][1] = 1;
    } else{ // Calculate S from v's children.
      int S[v][0] = 0; // not included
      int S[v][1] = 1; // included

      for (int w : V[v].childList()) {
        S[v][0] += S[w][1];
        S[v][1] += Math.min(S[w][0], S[w][1]);
      }
    }
  }
  return Math.min(S[0][0], S[0][1]); // returns min at root
}
```

⌛ Time Complexity

Perspective 1

- 2V subproblems
- O(V) time per sub-problem
- $T(n) = O(V^2)$

Perspective 2 - tighter bound

- 2V subproblems
- O(V) time to solve all sub-problems - similar to DFS
  - each edge is explored once
  - each sub-problem involves exploring children edges
- $T(n) = O(V)$

All Pairs Shortest Path (APSP)

Input

- directed, connected weighted graph G = (V, E)

Goal

- pre-process G
- answer queries: `minDist(v, w)`

Naïve Solution

- run Dijkstra on every query
  - preprocessing: 0
  - responding to q queries = O(qE log V)
- run Dijkstra from source `query(v,w)`
  - set distance array
  - responding to q queries = O(VE log V)

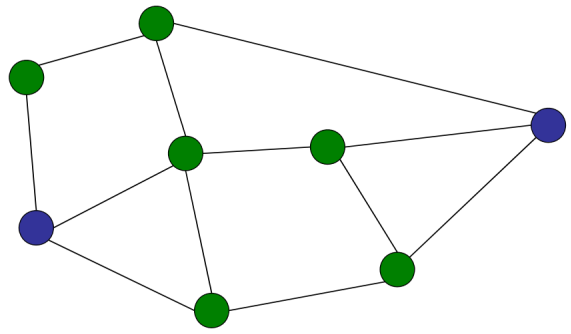
Diameter of a Graph

Input:

- undirected, weighted graph G = (V, E)

Output:

- a pair of nodes (v, w) such that the shortest path from v to w is maximal



diameter = 3

APSP

Input:

- weighted, directed graph G = (V, E)

Output:

- `dist[v,w]`: shortest distance from v to w, for all pairs of vertices (v,w)

Solution

- Run SSSP once for every vertex v in the graph
- assume all weights are positive

Sparse Graph → E = O(V):

$$T(n) = O(V \cdot V \cdot log(V)) = O(V^2 \cdot log(V))$$

Unweighted graph, use BFS: $O(V(E+V))$

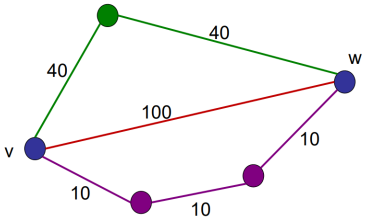
- In dense graph: $O(V^3)$
- In sparse graph: $O(V^2)$

Floyd-Warshall

Let `S[v,w,P]` be the shortest path from `v` to `w` that only uses intermediate nodes only in the set `P`.

P1 = no nodes (empty set)
P2 = green nodes
P3 = purple nodes

S(v,w,P1) = 100
S(v,w,P2) = 80
S(v,w,P3) = 30



Base case: $S[v, w, \emptyset] = E[v, w]$

- i.e. relax all outgoing edges from each node

Recursion step: $S[v, w, P_{k+1}] = min(S[v, w, P_k], S[v, k+1, P_k] + S[k+1, w, P_k]$

- for each vertex (with outgoing edges to nodes in P), relax the edge if the path passing through the set P yields a shorter path than the direct edge

Output:

Transitive Closure

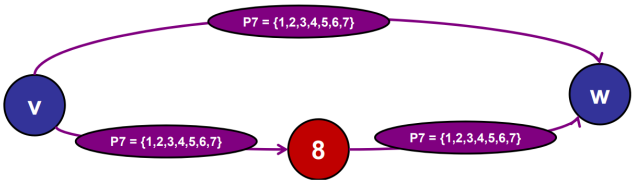
return a matrix M where

- `M[v,w] = 1` if there exists a path from `v` to `w`
- `M[v,w] = 0`, otherwise.

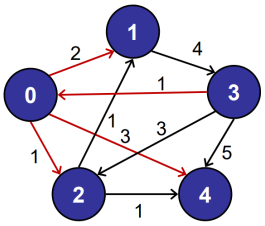
reduce space complexity by using only boolean variables

`P(v,w,k)` = true iff there is a path from `v` to `w` using only nodes `{0..k}`

`P(v,w,k)` = `P(v,w,k-1)` OR (`P(v,k,k-1)` AND `P(k,w,k-1)`)



Step: P = {0}

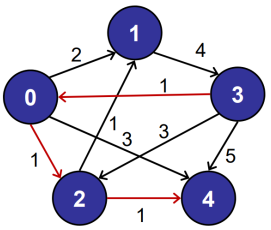


	0	1	2	3	4
0	0	**2**	**1**	∞	**3**
1	∞	0	∞	4	∞
2	∞	1	0	∞	1
3	**1**	∞	3	0	5
4	∞	∞	∞	∞	0

⇩

	0	1	2	3	4
0	0	2	1	∞	3
1	∞	0	∞	4	∞
2	∞	1	0	∞	1
3	1	**3**	**2**	0	**4**
4	∞	∞	∞	∞	0

Step: P = {0, 1, 2}

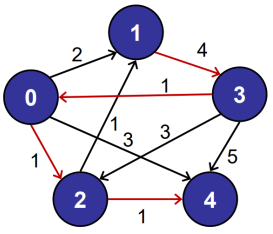


	0	1	2	3	4
0	0	2	**1**	6	3
1	∞	0	∞	4	∞
2	∞	1	0	5	**1**
3	1	3	**2**	0	4
4	∞	∞	∞	∞	0

⇩

	0	1	2	3	4
0	0	2	1	6	**2**
1	∞	0	∞	4	∞
2	∞	1	0	5	1
3	1	3	2	0	**3**
4	∞	∞	∞	∞	0

Step: P = {0, 1, 2, 3}



	0	1	2	3	4
0	0	2	1	6	2
1	∞	0	∞	**4**	∞
2	∞	1	0	**5**	1
3	**1**	3	**2**	0	**3**
4	∞	∞	∞	∞	0

⇩

	0	1	2	3	4
0	0	2	1	6	2
1	**5**	0	**6**	4	**7**
2	**6**	1	0	5	1
3	1	3	2	0	3
4	∞	∞	∞	∞	0

⌨️ Pseudo-Code for Floyd Warshall

```
Floyd-Warshall // Time: O(V^3), Space: O(V^2)
int[][] APSP(E){ // Adjacency matrix E
int[][] S = new int[V.length][V.length];//create memo table S
// Initialize every pair of nodes
  for (int v=0; v<V.length; v++)
     for (int w=0; w<V.length; w++)
         S[v][w] = E[v][w]
  // For sets P0, P1, P2, P3, …, for every pair (v,w)
  for (int k=0; k<V.length; k++)
     for (int v=0; v<V.length; v++)
       for (int w=0; w<V.length; w++)
          S[v][w] = min(S[v][w], (S[v][k] + S[k][w]));
// OR(S[v][w], (S[v][k] AND S[k][w]));
// Matrix Multiplication: OR - +; AND: *
return S;
}
```

⌛ Time Complexity

$T(n) = O(V^3)$

- no. of subproblems = $O(V^3)$
- cost per subproblem = O(1) → `min(...)`

📁 Space Complexity

$S(n) = O(V^2)$

Floyd-Warshall Variants

Minimum Bottleneck Edge:

- For `(v,w)`, the bottleneck is the heaviest edge on path between v and w.
- Return a matrix B where: `B[v,w]` = weight of the minimum bottleneck.