COMP 6651
Design and Analysis of Algorithms

Lecturer: Thomas Fevens

Department of Computer Science and Software Engineering, Concordia U
thomas.fevens@concordia.ca

Elementary Graph Algorithms Data Structures for Disjoint Sets Minimum Spanning Trees

Table of Contents

LECTURE NOTES FOR CHAPTER 20

Elementary Graph Algorithms

GRAPH REPRESENTATION

Given graph $G = (V, E)$. In pseudocode, represent vertex set by $G.V$ and edge set by $G.E$.

- G may be either directed or undirected.
- Two common ways to represent graphs for algorithms:
  1. Adjacency lists.
  2. Adjacency matrix.

When expressing the running time of an algorithm, it's often in terms of both $|V|$ and $|E|$. In asymptotic notation—and *only* in asymptotic notation—we'll drop the cardinality. Example: $O(V + E)$ really means $O(|V| + |E|)$.

ADJACENCY LISTS

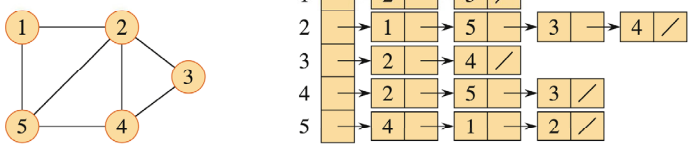
Array Adj of $|V|$ lists, one per vertex.

Vertex u's list has all vertices v such that $(u, v) \in E$. (Works for both directed and undirected graphs.)

In pseudocode, denote the array as attribute $G.Adj$, so will see notation such as $G.Adj[u]$.

EXAMPLE

For an undirected graph:



If edges have *weights*, can put the weights in the lists.

Weight: $w : E \to \mathbb{R}$

We'll use weights later on for spanning trees and shortest paths.

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

Time: to list all vertices adjacent to u: $\Theta(\text{degree}(u))$.

Time: to determine whether $(u, v) \in E$: $O(\text{degree}(u))$.

ADJACENCY MATRIX

$|V| \times |V|$ matrix $A = (a_{ij})$

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E\ , \\ 0 & \text{otherwise}\ . \end{cases}$$

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

	1	2	3	4
1	0	1	0	0
2	0	0	0	1
3	1	1	0	0
4	0	0	1	1

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

Time: to list all vertices adjacent to u: $\Theta(V)$.

Time: to determine whether $(u,v) \in E$: $\Theta(1)$.

Can store weights instead of bits for weighted graph.

BREADTH-FIRST SEARCH

Input: Graph $G = (V, E)$, either directed or undirected, and ***source vertex*** $s \in V$.

Output:

- $v.d$ = distance (smallest # of edges) from s to v, for all $v \in V$.
- $v.\pi$ is v's ***predecessor*** on a shortest path (smallest # of edges) from s.
  (u,v) is last edge on shortest path $s \rightsquigarrow v$.
  Predecessor subgraph contains edges (u,v) such that $v.\pi = u$.
  The predecessor subgraph forms a tree, called the ***breadth-first tree***.

Later, we'll see a generalization of breadth-first search, with edge weights. For now, we'll keep it simple.

BREADTH-FIRST SEARCH (continued)

Intuition

Breadth-first search expands the frontier between discovered and undiscovered vertices uniformly across the breath of the frontier.

Discovers vertices in waves, starting from s.

- First visits all vertices 1 edge from s.
- From there, visits all vertices 2 edges from s.
- Etc.

Use FIFO queue Q to maintain wavefront.

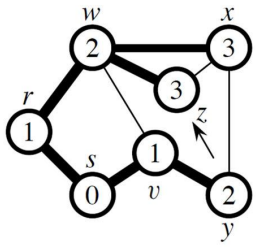
- $v \in Q$ if and only if wave has visited v but has not come out of v yet.
- Q contains vertices at a distance k, and possibly some vertices at a distance $k+1$. Therefore, at any time Q contains portions of two consecutive waves.

PSEUDOCODE

BFS(G, s)

```
1  for each vertex u ∈ G.V − {s}
2      u.color = WHITE
3      u.d = ∞
4      u.π = NIL
5  s.color = GRAY
6  s.d = 0
7  s.π = NIL
8  Q = ∅
9  ENQUEUE(Q, s)
10 while Q ≠ ∅
11     u = DEQUEUE(Q)
12     for each vertex v in G.Adj[u]   // search the neighbors of u
13         if v.color == WHITE         // is v being discovered now?
14             v.color = GRAY
15             v.d = u.d + 1
16             v.π = u
17             ENQUEUE(Q, v)           // v is now on the frontier
18     u.color = BLACK                 // u is now behind the frontier
```

EXAMPLE



$Q \quad \varnothing$

- Edges drawn with heavy lines are in the predecessor subgraph.
- Dashed lines go to newly discovered vertices. They are drawn with heavy lines because they are also now in the predecessor subgraph.
- Double-outline vertices have been discovered and are in Q, waiting to be visited.
- Heavy-outline vertices have been discovered, dequeued from Q, and visited.

BFS may not reach all vertices.

Time $= O(V + E)$.

- $O(V)$ because every vertex enqueued at most once.
- $O(E)$ because every vertex dequeued at most once and edge (u, v) is examined only when u is dequeued. Therefore, every edge examined at most once if directed, at most twice if undirected.

PSEUDOCODE

To print the vertices on a shortest path from s to v:

PRINT-PATH(G, s, v)

```
1  if v == s
2        print s
3  elseif v.π == NIL
4        print "no path from" s "to" v "exists"
5  else PRINT-PATH(G, s, v.π)
6        print v
```

DEPTH-FIRST SEARCH

Input: $G = (V, E)$, directed or undirected. No source vertex given.

Output:

- 2 ***timestamps*** on each vertex:
  - $v.d =$ ***discovery time***
  - $v.f =$ ***finish time***

  These will be useful for other algorithms later on.
- $v.\pi$ is v's predecessor in the ***depth-first forest*** of ≥ 1 ***depth-first trees***. If $u = v.\pi$, then (u, v) is a ***tree edge***.

DEPTH-FIRST SEARCH (continued)

Methodically explores *every* edge.

- Start over from different vertices as necessary.

As soon as a vertex is discovered, explore from it.

- Unlike BFS, which puts a vertex on a queue so that it's explored from later.

As DFS progresses, every vertex has a ***color***:

- WHITE = undiscovered
- GRAY = discovered, but not finished (not done exploring from it)
- BLACK = finished (have found everything reachable from it)

Discovery and finish times:

- Unique integers from 1 to $2|V|$.
- For all v, $v.d < v.f$.

In other words, $1 \leq v.d < v.f \leq 2|V|$.

PSEUDOCODE

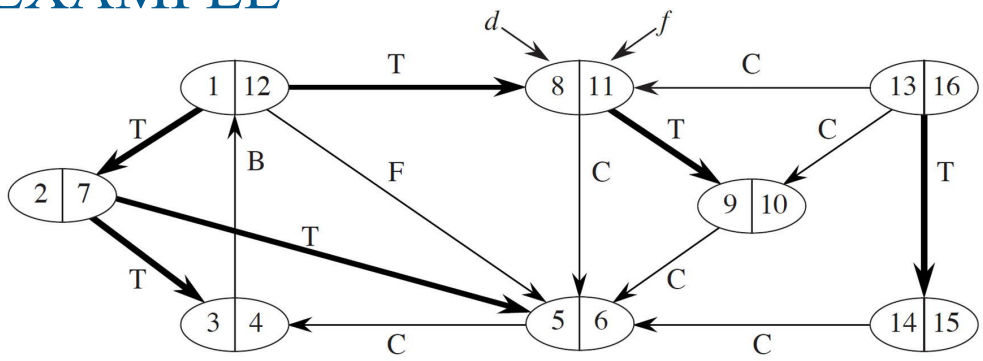
```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

Global search starts a local search on each vertex to explore entire graph.

PSEUDOCODE (continued)

```
DFS-VISIT(G, u)
1  time = time + 1                  // white vertex u has just been discovered
2  u.d = time
3  u.color = GRAY
4  for each vertex v in G.Adj[u]    // explore each edge (u, v)
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  time = time + 1
9  u.f = time
10 u.color = BLACK                  // blacken u; it is finished
```

EXAMPLE



Time $= \Theta(V + E)$.

- Similar to BFS analysis.
- Θ, not just O, since guaranteed to examine every vertex and edge.

Each depth-first tree is made of edges (u, v) such that u is gray and v is white when (u, v) is explored.

THEOREM (PARENTHESIS THEOREM)

For all u, v, exactly one of the following holds:

1. $u.d < u.f < v.d < v.f$ or $v.d < v.f < u.d < u.f$ (i.e., the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are disjoint) and neither of u and v is a descendant of the other.
2. $u.d < v.d < v.f < u.f$ and v is a descendant of u. (v is discovered after and finished before u.)
3. $v.d < u.d < u.f < v.f$ and u is a descendant of v. (u is discovered after and finished before v.)

So $u.d < v.d < u.f < v.f$ (v is both discovered and finished after u) *cannot* happen.

CLASSIFICATION OF EDGES

- ***Tree edge:*** in the depth-first forest. Found by exploring (u, v).
- ***Back edge:*** (u, v), where u is a descendant of v.
- ***Forward edge:*** (u, v), where v is a descendant of u, but not a tree edge.
- ***Cross edge:*** any other edge. Can go between vertices in same depth-first tree or in different depth-first trees.

In an undirected graph, there may be some ambiguity since (u, v) and (v, u) are the same edge. Classify by the first type above that matches.

TOPOLOGICAL SORT

Directed acyclic graph (dag)

A directed graph with no cycles.

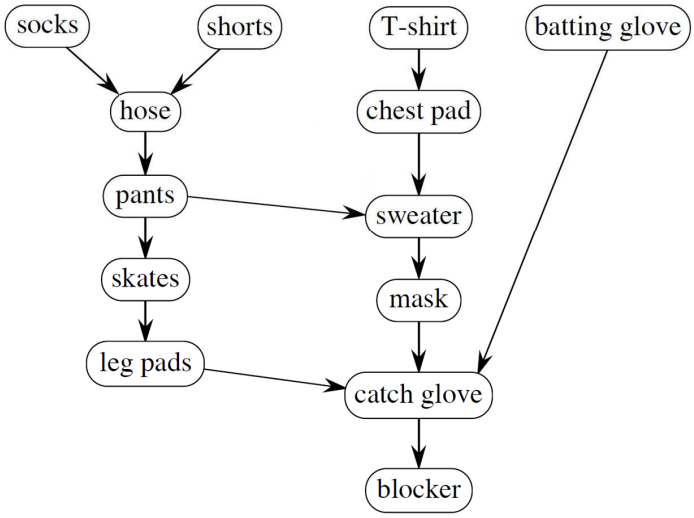
Good for modeling processes and structures that have a ***partial order:***

- $a > b$ and $b > c \Rightarrow a > c$.
- But may have a and b such that neither $a > b$ nor $b > c$.

Can always make a ***total order*** (either $a > b$ or $b > a$ for all $a \neq b$) from a partial order. In fact, that's what a topological sort will do.

EXAMPLE

Dag of dependencies for putting on goalie equipment for ice hockey:

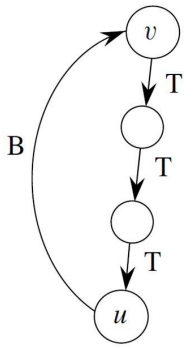


EXAMPLE (continued)

A directed graph G is acyclic if and only if a DFS of G yields no back edges.

Proof $\Rightarrow$: Show that back edge $\Rightarrow$ cycle.

Suppose there is a back edge (u, v). Then v is ancestor of u in depth-first forest.



Therefore, there is a path $v \rightsquigarrow u$, so $v \rightsquigarrow u \rightarrow v$ is a cycle.

EXAMPLE (continued)

Topological sort of a dag: a linear ordering of vertices such that if $(u, v) \in E$, then u appears somewhere before v. (Not like sorting numbers.)

TOPOLOGICAL-SORT(G)

1 call DFS(G) to compute finish times $v.f$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices

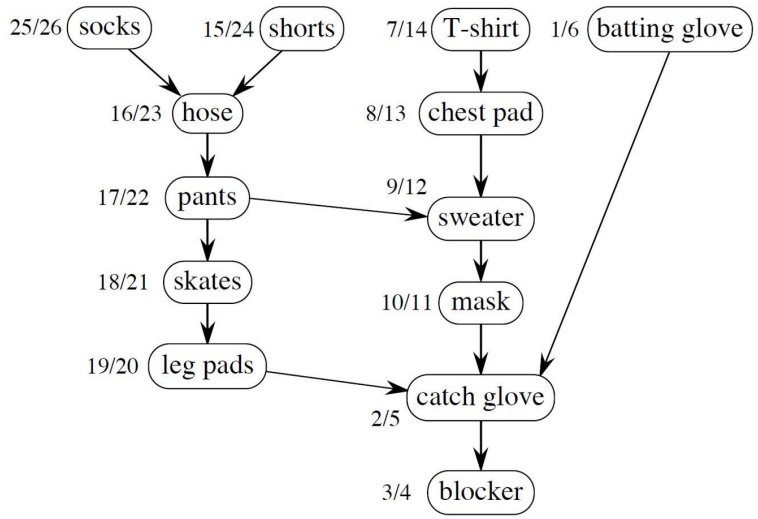
Don't need to sort by finish times.

- Can just output vertices as they're finished and understand that we want the *reverse* of this list.
- Or put them onto the *front* of a linked list as they're finished. When done, the list contains vertices in topologically sorted order.

Time $\Theta(V + E)$.

EXAMPLE (continued)

Let's run DFS on this example to find finish times.



EXAMPLE (continued)

Correctness

Just need to show if $(u, v) \in E$, then $v.f < u.f$.
When edge (u, v) is explored, what are the colors of u and v?

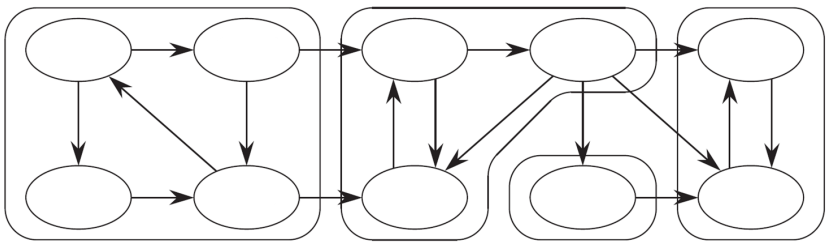
- u is gray.
- Is v gray, too?
  - *No*, because then v would be ancestor of u.
    $\Rightarrow (u, v)$ is a back edge.
    $\Rightarrow$ contradiction of previous lemma (dag has no back edges).
- Is v white?
  - Then becomes descendant of u.
    By parenthesis theorem, $u.d < v.d < v.f < u.f$.
- Is v black?
  - Then v is already finished.
    Since exploring (u, v), u is not yet finished.
    Therefore, $v.f < u.f$.

STRONGLY CONNECTED COMPONENTS

Given directed graph $G = (V, E)$.

A ***strongly connected component*** (***SCC***) of G is a maximal set of vertices $C \subseteq V$ such that for all $u, v \in C$, both $u \rightsquigarrow v$ and $v \rightsquigarrow u$.

Example



EXAMPLE (continued)

Algorithm uses $G^{\mathrm{T}} =$ ***transpose*** of G.

- $G^{\mathrm{T}} = (V, E^{\mathrm{T}})$, $E^{\mathrm{T}} = \{(u, v) : (v, u) \in E\}$.
- G^{T} is G with all edges reversed.

Can create G^{T} in $\Theta(V + E)$ time if using adjacency lists.

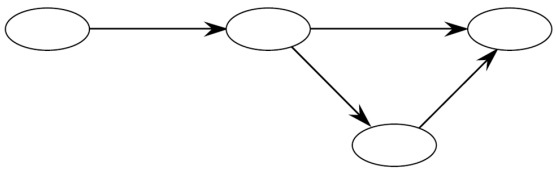
Observation

G and G^{T} have the *same* SCC's. (u and v are reachable from each other in G if and only if reachable from each other in G^{T}.)

EXAMPLE (continued)

- $G^{\mathrm{SCC}} = (V^{\mathrm{SCC}}, E^{\mathrm{SCC}})$.
- V^{SCC} has one vertex for each SCC in G.
- E^{SCC} has an edge if there's an edge between the corresponding SCC's in G.

For our example:



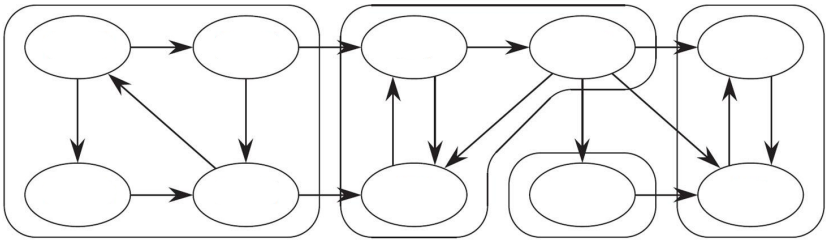
Lemma

G^{SCC} is a dag. More formally, let C and C' be distinct SCC's in G, let $u, v \in C$, $u', v' \in C'$, and suppose there is a path $u \rightsquigarrow u'$ in G. Then there cannot also be a path $v' \rightsquigarrow v$ in G.

EXAMPLE (continued)

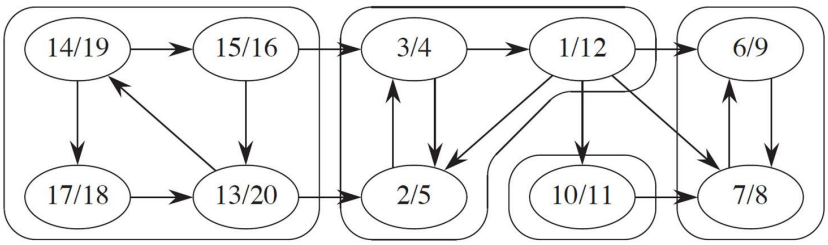
SCC(G)
  call DFS(G) to compute finish times $u.f$ for each vertex u
  create G^T
  call DFS(G^T), but in the main loop, consider vertices in order of decreasing $u.f$
      (as computed in first DFS)
  output the vertices in each tree of the depth-first forest formed in second DFS
      as a separate SCC

Do DFS in G.



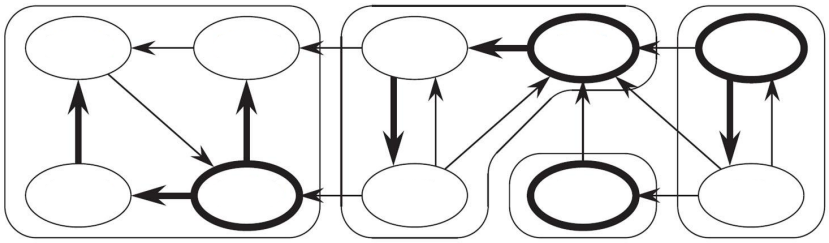
EXAMPLE (continued)

Run DFS.



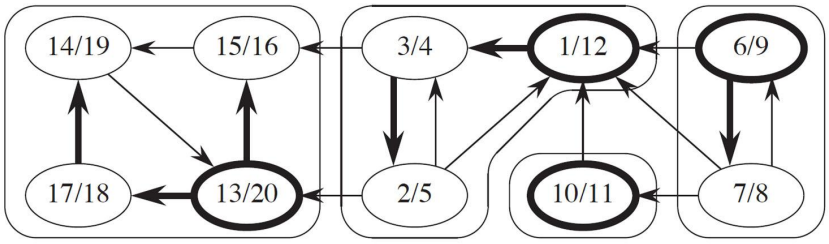
EXAMPLE (continued)

Take G^T.



EXAMPLE (continued)

DFS in G^T.



Time: $\Theta(V + E)$.

Each top level DFS-Visit will discover its component.

EXAMPLE (continued)

How can this possibly work?

Idea

By considering vertices in second DFS in decreasing order of finish times from first DFS, visiting vertices of the component graph in topological sort order.

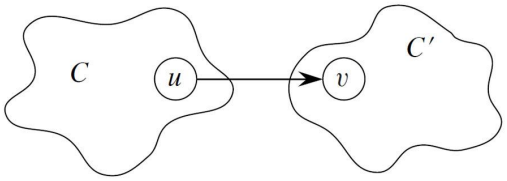
To prove that it works, first deal with 2 notational issues:

- Will be discussing $u.d$ and $u.f$. These always refer to *first* DFS.
- Extend notation for d and f to sets of vertices $U \subseteq V$:
  - $d(U) = \min\{u.d : u \in U\}$ (earliest discovery time in U)
  - $f(U) = \max\{u.f : u \in U\}$ (latest finish time in U)

EXAMPLE (continued)

Lemma

Let C and C' be distinct SCC's in $G = (V, E)$. Suppose that there is an edge $(u, v) \in E$ such that $u \in C$ and $v \in C'$.



Then $f(C) > f(C')$.

STRONGLY CONNECTED COMPONENTS

Each root chosen for the second DFS can reach only

- vertices in its SCC—get tree edges to these,
- vertices in SCC's *already visited* in second DFS—get *no* tree edges to these.

Visiting vertices of $(G^T)^{SCC}$ in reverse of topologically sorted order.

LECTURE NOTES FOR CHAPTER 19

Data Structures for Disjoint Sets

CHAPTER 19 OVERVIEW

Disjoint-set data structures

- Also known as "union find."
- Maintain collection $\mathcal{S} = \{S_1, \ldots, S_k\}$ of disjoint dynamic (changing over time) sets.
- Each set is identified by a *representative*, which is some member of the set.

  Doesn't matter which member is the representative, as long as if you ask for the representative twice without modifying the set, you get the same answer both times.

OPERATIONS

- MAKE-SET(x): make a new set $S_i = \{x\}$, and add S_i to $\mathcal{S}$.
- UNION(x, y): if $x \in S_x$, $y \in S_y$, then $\mathcal{S} = \mathcal{S} - S_x - S_y \cup \{S_x \cup S_y\}$.
  - Representative of new set is any member of $S_x \cup S_y$, often the representative of one of S_x and S_y.
  - Destroys S_x and S_y (since sets must be disjoint).
- FIND-SET(x): return representative of set containing x.

Analysis in terms of:

- n = # of elements = # of MAKE-SET operations,
- m = total # of operations.

OPERATIONS (continued)

Analysis

- Since MAKE-SET counts toward total # of operations, $m \geq n$.
- Can have at most $n - 1$ UNION operations, since after $n - 1$ UNIONs, only 1 set remains.
- Assume that the first n operations are MAKE-SET (helpful for analysis, usually not really necessary).

Application

Dynamic connected components.

For a graph $G = (V, E)$, vertices u, v are in same connected component if and only if there's a path between them.

- Connected components partition vertices into equivalence classes.

OPERATIONS (continued)

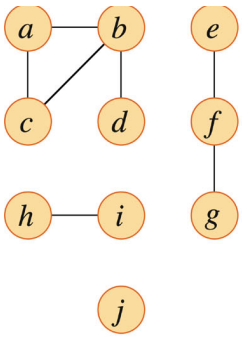
```
CONNECTED-COMPONENTS(G)
1  for each vertex v ∈ G.V
2      MAKE-SET(v)
3  for each edge (u, v) ∈ G.E
4      if FIND-SET(u) ≠ FIND-SET(v)
5          UNION(u, v)
```

```
SAME-COMPONENT(u, v)
1  if FIND-SET(u) == FIND-SET(v)
2      return TRUE
3  else return FALSE
```

Note: If actually implementing connected components,

- each vertex needs a handle to its object in the disjoint-set data structure,
- each object in the disjoint-set data structure needs a handle to its vertex.

CONNECTED COMPONENTS EXAMPLE



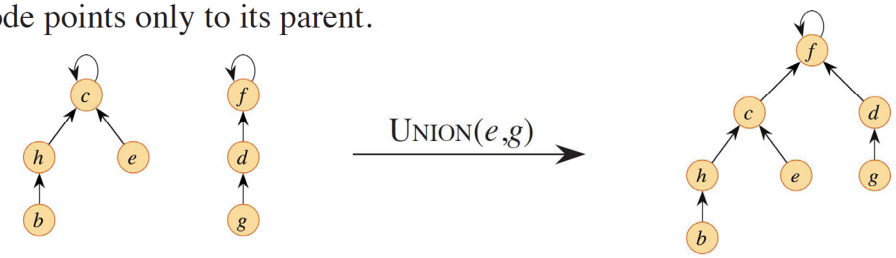
Edge processed	Collection of disjoint sets									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b, d)	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e, f)	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e, f\}$		$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(a, c)	$\{a, c\}$	$\{b, d\}$			$\{e, f\}$		$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(h, i)	$\{a, c\}$	$\{b, d\}$			$\{e, f\}$		$\{g\}$	$\{h, i\}$		$\{j\}$
(a, b)	$\{a, b, c, d\}$				$\{e, f\}$		$\{g\}$	$\{h, i\}$		$\{j\}$
(f, g)	$\{a, b, c, d\}$				$\{e, f, g\}$			$\{h, i\}$		$\{j\}$
(b, c)	$\{a, b, c, d\}$				$\{e, f, g\}$			$\{h, i\}$		$\{j\}$

(a) (b)

DISJOINT-SET FOREST REPRESENTATION

Forest of trees.

- 1 tree per set. Root is representative.
- Each node points only to its parent.

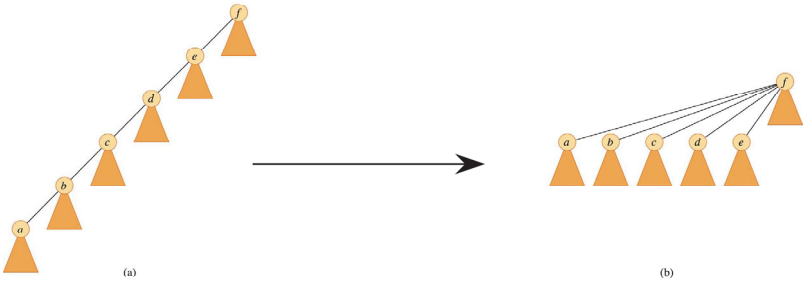


- MAKE-SET: make a single-node tree.
- UNION: make one root a child of the other.
- FIND-SET: follow pointers to the root.

Not so good—could get a linear chain of nodes.

GREAT HEURISTICS

- ***Path compression: Find path*** = nodes visited during FIND-SET on the trip to the root. Make all nodes on the find path direct children of root.



- ***Union by rank:*** make the root of the smaller tree (fewer nodes) a child of the root of the larger tree.
  - Don't actually use *size*.
  - Use *rank*, which is an upper bound on height of node.
  - Make the root with the smaller rank into a child of the root with the larger rank.

GREAT HEURISTICS (continued)

```
MAKE-SET(x)
1  x.p = x
2  x.rank = 0

UNION(x, y)
1  LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
1  if x.rank > y.rank
2       y.p = x
3  else x.p = y
4       if x.rank == y.rank
5            y.rank = y.rank + 1

FIND-SET(x)
1  if x ≠ x.p                  // not the root?
2       x.p = FIND-SET(x.p)    // the root becomes the parent
3  return x.p                  // return the root
```

GREAT HEURISTICS (continued)

Running time

If use both union by rank and path compression, $O(m\,\alpha(n))$.

$\alpha(n)$ is a *very* slowly growing function:

n	$\alpha(n)$
0–2	0
3	1
4–7	2
8–2047	3
2048–$A_4(1)$	4

What's $A_4(1)$? See Section 19.4, if you dare. It's $\gg 10^{80} \approx$ # of atoms in observable universe.

This bound is tight—there exists a sequence of operations that takes $\Omega(m\,\alpha(n))$ time.

LECTURE NOTES FOR CHAPTER 21

Minimum Spanning Trees

CHAPTER 21 OVERVIEW

Problem

- A town has a set of houses and a set of roads.
- A road connects 2 and only 2 houses.
- A road connecting houses u and v has a repair cost $w(u, v)$.
- ***Goal:*** Repair enough (and no more) roads such that

  1. everyone stays connected: can reach every house from all other houses, and
  2. total repair cost is minimum.

CHAPTER 21 OVERVIEW (continued)

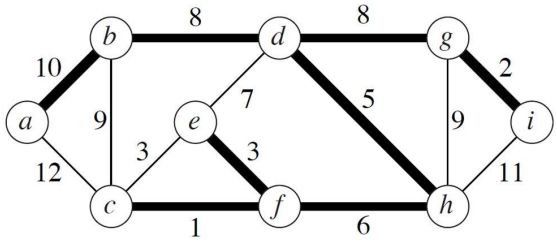
Model as a graph:

- Undirected graph $G = (V, E)$.
- ***Weight*** $w(u, v)$ on each edge $(u, v) \in E$.
- Find $T \subseteq E$ such that

  1. T connects all vertices (T is a ***spanning tree***), and
  2. $w(T) = \sum_{(u,v)\in T} w(u, v)$ is minimized.

CHAPTER 21 OVERVIEW (continued)

A spanning tree whose weight is minimum over all spanning trees is called a ***minimum spanning tree***, or ***MST***.

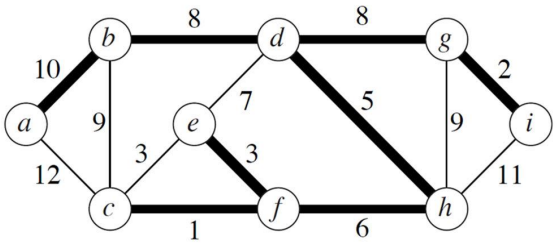


In this example, there is more than one MST. Replace edge (e, f) in the MST by (c, e). Get a different spanning tree with the same weight.

GROWING A MINIMUM SPANNING TREE

Some properties of an MST:

- It has $|V| - 1$ edges.
- It has no cycles.
- It might not be unique.



PROVING CORRECTNESS

Loop invariants help us understand why an algorithm is correct.

When you're using a loop invariant, you need to show three things:

Initialization: It is true prior to the first iteration of the loop.
Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.
Termination: The loop terminates, and when it terminates, the invariant—usually along with the condition that caused the loop to terminate—gives us a useful property that helps show that the algorithm is correct.

BUILDING UP THE SOLUTION

- Build a set A of edges.
- Initially, A has no edges.
- As edges are added to A, maintain a loop invariant:

  Loop invariant: A is a subset of some MST.
- Add only edges that maintain the invariant. If A is a subset of some MST, an edge (u, v) is ***safe*** for A if and only if $A \cup \{(u, v)\}$ is also a subset of some MST. So add only safe edges.

GENERIC MST ALGORITHM

GENERIC-MST(G, w)

1 $A = \emptyset$
2 **while** A does not form a spanning tree
3 find an edge (u, v) that is safe for A
4 $A = A \cup \{(u, v)\}$
5 **return** A

Use the loop invariant to show that this generic algorithm works.

Initialization: The empty set trivially satisfies the loop invariant.

Maintenance: Since only safe edges are added, A remains a subset of some MST.

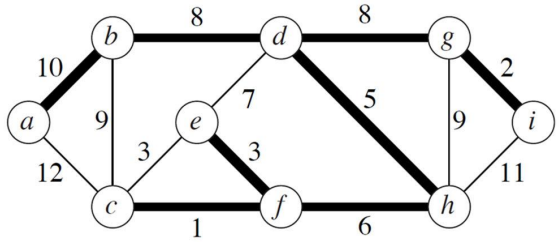
Termination: The loop must terminate by the time it considers all edges. All edges added to A are in an MST, so upon termination. A is a spanning tree that is also an MST.

FINDING A SAFE EDGE

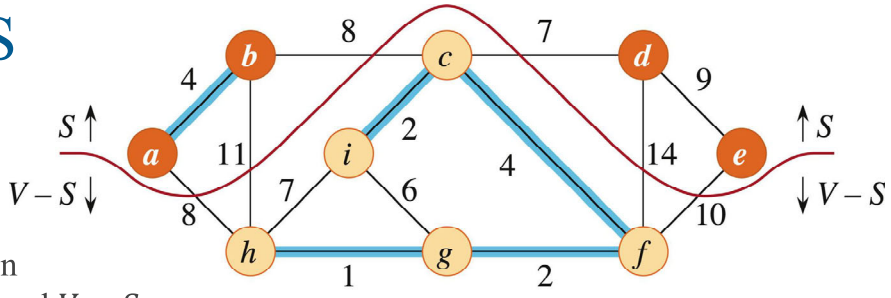
How to find safe edges?

Let's look at the example. Edge (c, f) has the lowest weight of any edge in the graph. Is it safe for $A = \emptyset$?

Intuitively: Let $S \subset V$ be any proper subset of vertices that includes c but not f (so that f is in $V - S$). In any MST, there has to be one edge (at least) that connects S with $V - S$. Why not choose the edge with minimum weight? (Which would be (c, f) in this case.)



DEFINITIONS

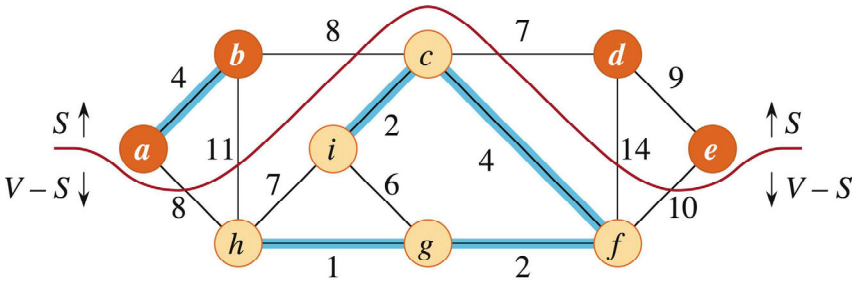


- A **cut** $(S, V - S)$ of an undirected graph $G = (V, E)$ is a partition of V into disjoint sets S and $V - S$.
- Edge $(u, v) \in E$ ***crosses*** the cut $(S, V - S)$ if one of its endpoints belongs to S and the other belongs to $V - S$.
- A cut ***respects*** a set A of edges (e.g., the turquoise-colored edges in the figure) if no edge in A crosses the cut.
- An edge is a ***light edge*** crossing a cut if its weight is the minimum of any edge crossing the cut. There can be > 1 light edges crossing a cut in the case of ties.

PROVING CORRECTNESS

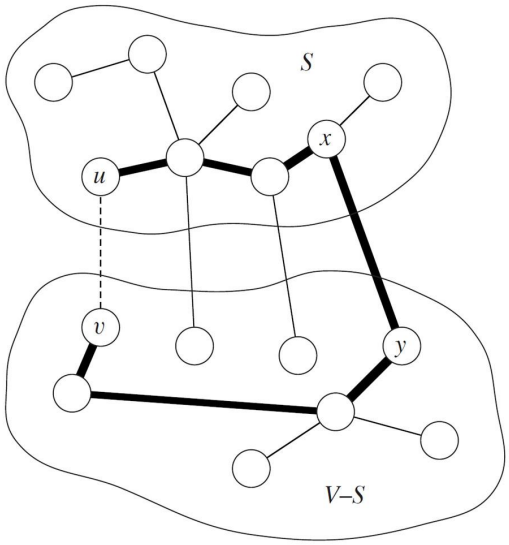
Theorem

Let $G = (V, E)$ be a connected, undirected graph with a 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, and let (u, v) be a light edge crossing $(S, V - S)$. Then, edge (u, v) is safe for A.



PROOF

Let (S, V-S) be a partition respecting A, a subset of some *MST* T. All drawn edges here are part of T except for (u,v), the light edge of the cut. If (u,v) is not in the tree, replacing (x,y) with (u,v) gives a new spanning tree with lower cost, a contradiction.



KRUSKAL'S ALGORITHM

$G = (V, E)$ is a connected, undirected, weighted graph. $w : E \to \mathbb{R}$.

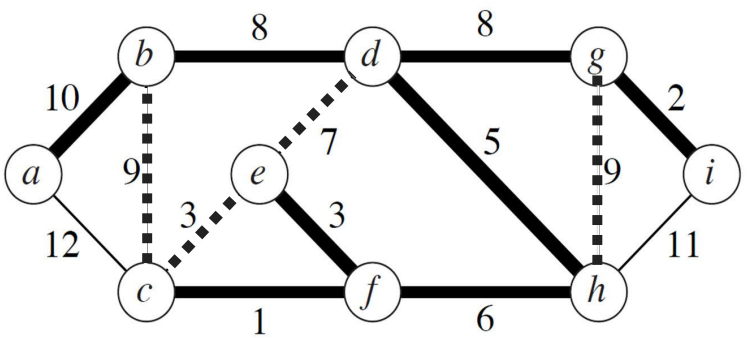
- Starts with each vertex being its own component.
- Repeatedly merges two components into one by choosing the light edge that connects them (i.e., the light edge crossing the cut between them).
- Scans the set of edges in monotonically increasing order by weight.
- Uses a disjoint-set data structure to determine whether an edge connects vertices in different components.

PSEUDOCODE

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3       MAKE-SET(v)
4  create a single list of the edges in G.E
5  sort the list of edges into monotonically increasing order by weight w
6  for each edge (u, v) taken from the sorted list in order
7       if FIND-SET(u) ≠ FIND-SET(v)
8            A = A ∪ {(u, v)}
9            UNION(u, v)
10 return A
```


EXAMPLE

Let's see Kruskal's algorithm on this graph.



(c, f) : safe
(g, i) : safe
(e, f) : safe
(c, e) : reject
(d, h) : safe
(f, h) : safe
(e, d) : reject
(b, d) : safe
(d, g) : safe
(b, c) : reject
(g, h) : reject
(a, b) : safe

ANALYSIS

Initialize A:	$O(1)$
First **for** loop:	$\lvert V\rvert$ MAKE-SETs
Sort E:	$O(E \lg E)$
Second **for** loop:	$O(E)$ FIND-SETs and UNIONs

- Assuming the implementation of disjoint-set data structure, already seen in Chapter 19, that uses union by rank and path compression:

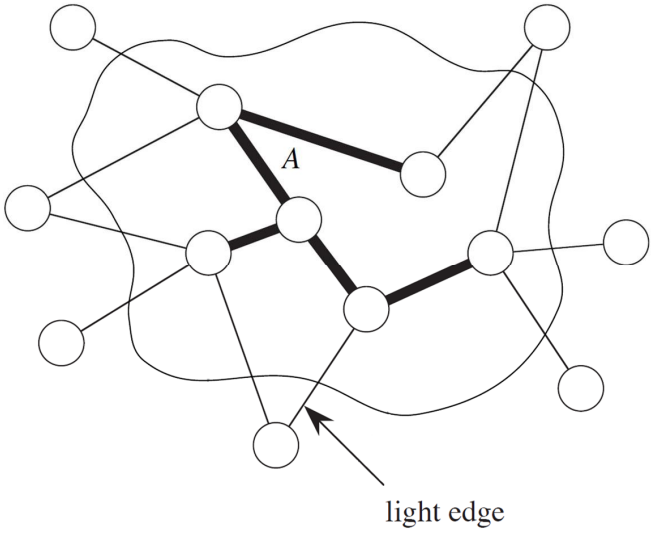
  $$O((V + E)\,\alpha(V)) + O(E \lg E)\ .$$

- Since G is connected, $|E| \geq |V| - 1 \Rightarrow O(E\,\alpha(V)) + O(E \lg E)$.
- $\alpha(|V|) = O(\lg V) = O(\lg E)$.
- Therefore, total time is $O(E \lg E)$.
- $|E| \leq |V|^2 \Rightarrow \lg |E| = O(2 \lg V) = O(\lg V)$.
- Therefore, $O(E \lg V)$ time. (If edges are already sorted, $O(E\,\alpha(V))$, which is almost linear.)

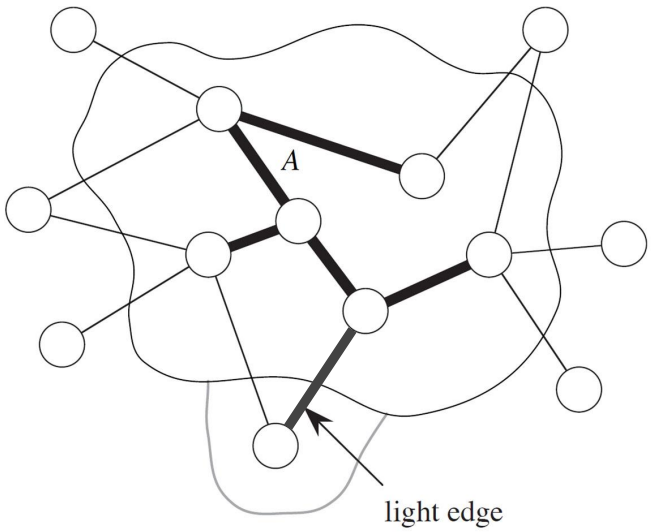
PRIM'S ALGORITHM

- Builds one tree, so A is always a tree.
- Starts from an arbitrary "root" r.
- At each step, find a light edge connecting A to an isolated vertex. Such an edge must be safe for A. Add this edge to A.

PRIM'S ALGORITHM (continued)



PRIM'S ALGORITHM (continued)



FINDING A LIGHT EDGE

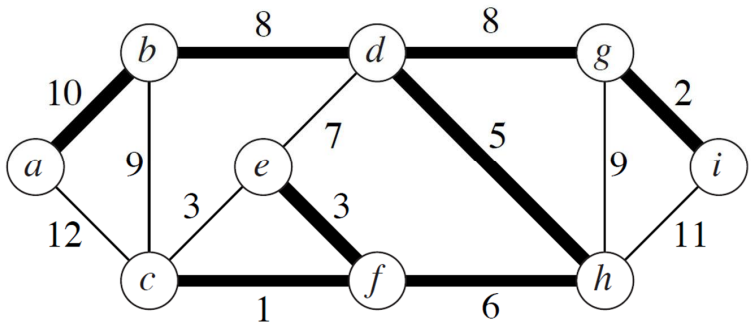
How to find the light edge quickly?

Use a priority queue Q:

- Each object is a vertex *not* in A.
- $v.key$ is the minimum weight of any edge connecting v to a vertex in A. $v.key = \infty$ if no such edge.
- $v.\pi$ is v's parent in A.
- Maintain A implicitly as $A = \{(v, v.\pi) : v \in V - \{r\} - Q\}$.
- At completion, Q is empty and the minimum spanning tree is $A = \{(v, v.\pi) : v \in V - \{r\}\}$.

EXAMPLE

Let's see Prim's algorithm on this graph. Pick a root.



ANALYSIS

Depends on how the priority queue is implemented:

- Suppose Q is a binary heap.

Initialize Q and first **for** loop:	$O(V \lg V)$
Decrease key of r:	$O(\lg V)$
while loop:	$\lvert V\rvert$ EXTRACT-MIN calls $\Rightarrow O(V \lg V)$
	$\leq \lvert E\rvert$ DECREASE-KEY calls $\Rightarrow O(E \lg V)$
Total:	$O(E \lg V)$

- Suppose DECREASE-KEY could take $O(1)$ *amortized* time.

  Then $\leq \lvert E\rvert$ DECREASE-KEY calls take $O(E)$ time altogether $\Rightarrow$ total time becomes $O(V \lg V + E)$.

  In fact, there is a way to perform DECREASE-KEY in $O(1)$ amortized time: Fibonacci heaps, mentioned in the introduction to Part V.