Strongly Connected Component, Minimum Spanning Tree

Congduan Li

Chinese University of Hong Kong, Shenzhen

congduan.li@gmail.com

Nov 28 & 30, 2017

Elementary Graph Algorithms (Chap 22)

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

- May be either directed or undirected.
- Two common ways to represent 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: $O(V + E)$.

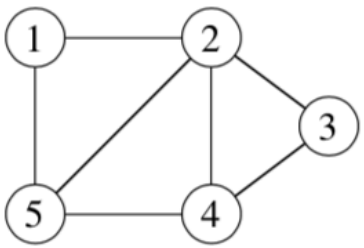
Adjacency lists

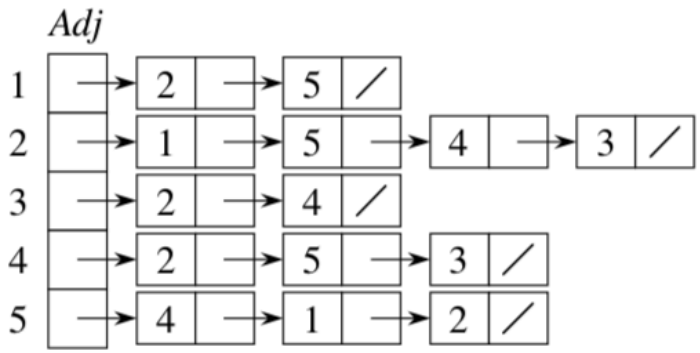
Adjacency lists

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.)

Example: For an undirected graph:





Directed case

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

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

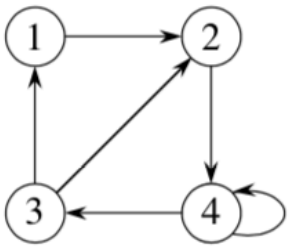
We'll use weights later on for spanning trees and shortest paths.

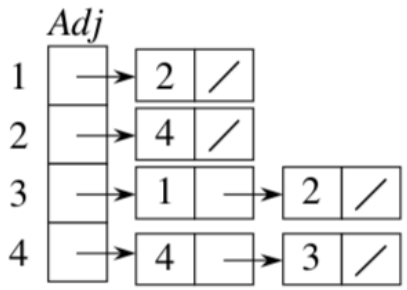
Space: $\Theta(V + E)$.

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

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

Example: For a directed graph:





Same asymptotic space and time.

Adjacency matrix

Adjacency matrix

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

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

	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 if $(u, v) \in E$: $\Theta(1)$.

Can store weights instead of bits for weighted graph.

We'll use both representations in these lecture notes.

Breadth First Search

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

Output: $d[v]$ = distance (smallest # of edges) from s to v, for all $v \in V$.
In book, also $\pi[v] = u$ such that (u, v) is last edge on shortest path $s \rightsquigarrow v$.

- u is v's ***predecessor***.
- set of edges $\{(\pi[v], v) : v \neq s\}$ forms a tree.

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

- Compute only $d[v]$, not $\pi[v]$. *[See book for $\pi[v]$.]*
- Omitting colors of vertices. *[Used in book to reason about the algorithm. We'll skip them here.]*

BFS

Idea: Send a wave out from s.

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

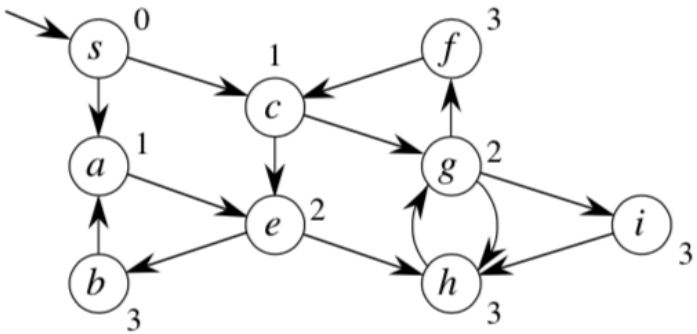
Use FIFO queue Q to maintain wavefront.

- $v \in Q$ if and only if wave has hit v but has not come out of v yet.

```
BFS(V, E, s)
for each u ∈ V − {s}
    do d[u] ← ∞
d[s] ← 0
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    do u ← DEQUEUE(Q)
       for each v ∈ Adj[u]
           do if d[v] = ∞
                 then d[v] ← d[u] + 1
                      ENQUEUE(Q, v)
```

Example

Example: directed graph *[undirected example in book]* .



Can show that Q consists of vertices with d values.

$i \quad i \quad i \quad \ldots \quad i \quad i+1 \quad i+1 \quad \ldots \quad i+1$

- Only 1 or 2 values.
- If 2, differ by 1 and all smallest are first.

BFS Analysis

Since each vertex gets a finite d value at most once, values assigned to vertices are monotonically increasing over time.

Actual proof of correctness is a bit trickier. See book.

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 we examine (u, v) only when u is dequeued. Therefore, every edge examined at most once if directed, at most twice if undirected.

Depth First Search

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

Output: 2 ***timestamps*** on each vertex:

- $d[v] =$ ***discovery time***
- $f[v] =$ ***finishing time***

These will be useful for other algorithms later on.

Can also compute $\pi[v]$. *[See book.]*

Will methodically explore *every* edge.

- Start over from different vertices as necessary.

As soon as we discover a vertex, explore from it.

- Unlike BFS, which puts a vertex on a queue so that we explore from it 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, $d[v] < f[v]$.

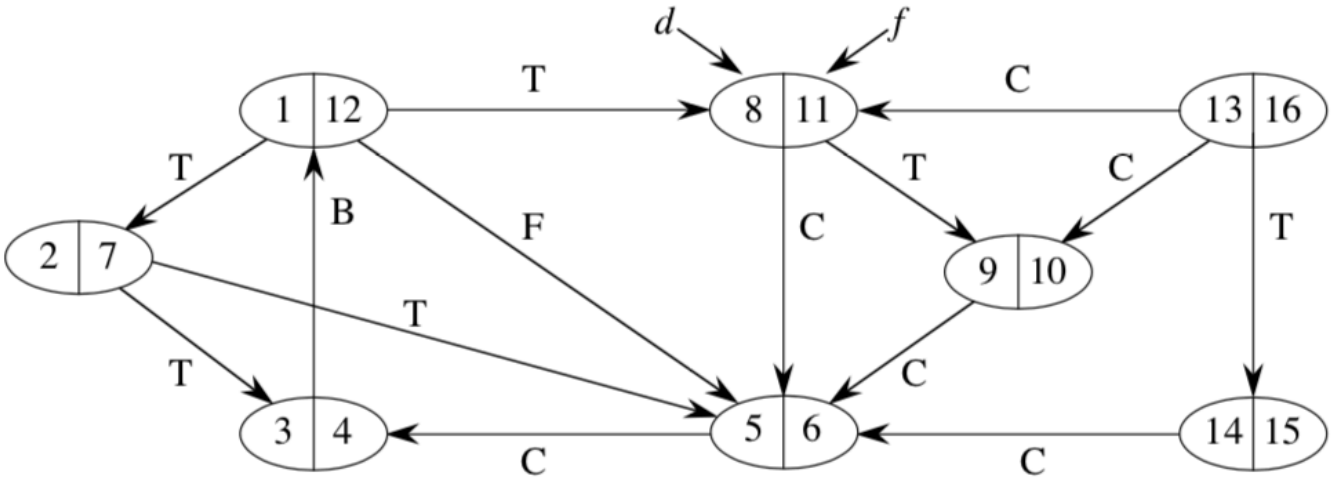
In other words, $1 \leq d[v] < f[v] \leq 2|V|$.

Pseudocode: Uses a global timestamp *time*.

```
DFS(V, E)
for each u ∈ V
    do color[u] ← WHITE
time ← 0
for each u ∈ V
    do if color[u] = WHITE
          then DFS-VISIT(u)

DFS-VISIT(u)
color[u] ← GRAY          ▷ discover u
time ← time +1
d[u] ← time
for each v ∈ Adj[u]      ▷ explore (u, v)
    do if color[v] = WHITE
          then DFS-VISIT(v)
color[u] ← BLACK
time ← time +1
f[u] ← time              ▷ finish u
```

Example



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

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

DFS forms a ***depth-first forest*** comprised of > 1 ***depth-first trees***. Each tree is made of edges (u, v) such that u is gray and v is white when (u, v) is explored.

Parenthesis Theorem

Theorem (Parenthesis theorem)

[Proof omitted.]

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

1. $d[u] < f[u] < d[v] < f[v]$ or $d[v] < f[v] < d[u] < f[u]$ and neither of u and v is a descendant of the other.
2. $d[u] < d[v] < f[v] < f[u]$ and v is a descendant of u.
3. $d[v] < d[u] < f[u] < f[v]$ and u is a descendant of v.

So $d[u] < d[v] < f[u] < f[v]$ *cannot* happen.

Like parentheses:

- OK: ( ) [ ] ([ ]) [( )]
- Not OK: ([ )] [( ])

White-path Theorem

Corollary

v is a proper descendant of u if and only if $d[u] < d[v] < f[v] < f[u]$.

Theorem (White-path theorem)

[Proof omitted.]

v is a descendant of u if and only if at time $d[u]$, there is a path $u \rightsquigarrow v$ consisting of only white vertices. (Except for u, which was *just* colored gray.)

Edge classes

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.

[Now label the example from above with edge types.]

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.

Theorem

[Proof omitted.]

In DFS of an *un*directed graph, we get only tree and back edges. No forward or cross edges.

Topological Order

Directed acyclic graph (dag)

A directed graph with no cycles.

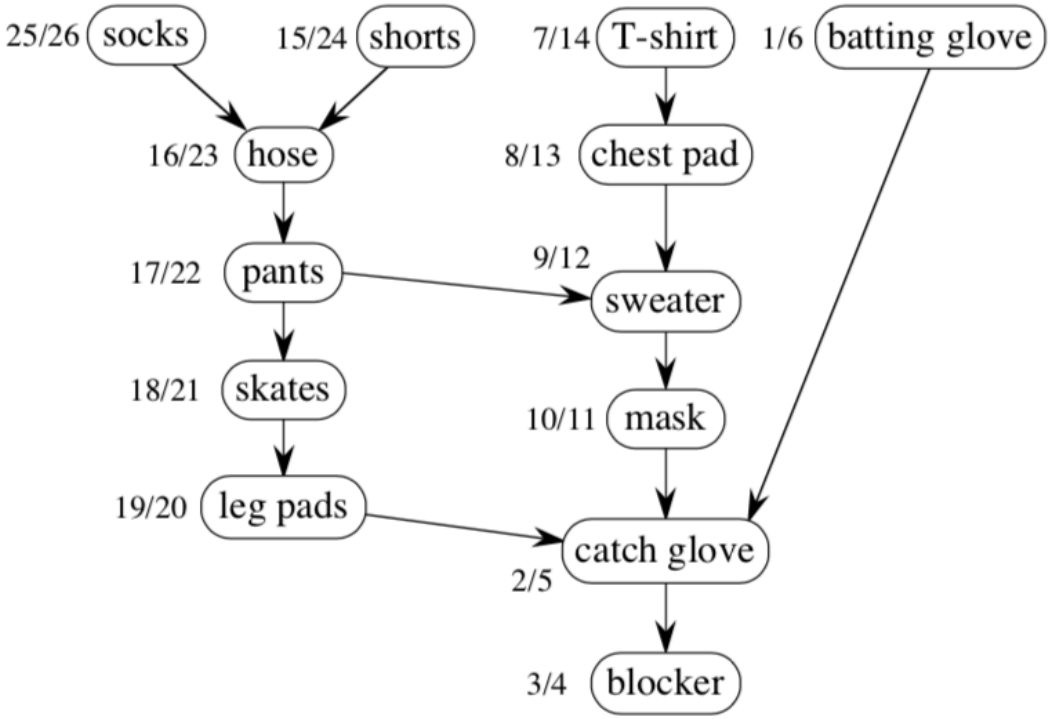
Good for modeling processes and structures that have a ***partial order:***

- $a > b$ 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

Example: dag of dependencies for putting on goalie equipment:



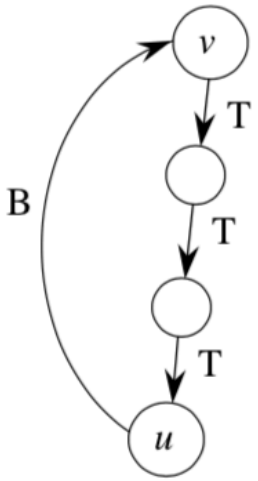
Acyclic

Lemma

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.

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

Suppose G contains cycle c. Let v be the first vertex discovered in c, and let (u, v) be the preceding edge in c. At time $d[v]$, vertices of c form a white path $v \rightsquigarrow u$ (since v is the first vertex discovered in c). By white-path theorem, u is descendant of v in depth-first forest. Therefore, (u, v) is a back edge. ■ (lemma)

Topological Sort

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(V, E)
call DFS(V, E) to compute finishing times $f[v]$ for all $v \in V$
output vertices in order of *decreasing* finish times

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

Do example.

Order:

26	socks
24	shorts
23	hose
22	pants
21	skates
20	leg pads
14	t-shirt
13	chest pad
12	sweater
11	mask
6	batting glove
5	catch glove
4	blocker

Correctness

Correctness: Just need to show if $(u, v) \in E$, then $f[v] < f[u]$.
When we explore (u, v), what are the colors of u and v?

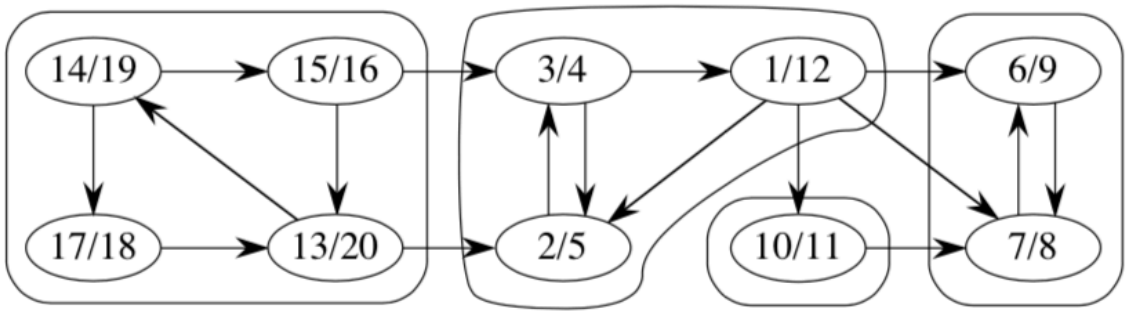
- u is gray.
- Is v gray, too?
  - *No*, because then v would be ancestor of u.
    $\Rightarrow (u, v)$ is a back edge.
    $\Rightarrow$ contradiction of previous lemma (dag has no back edges).
- Is v white?
  - Then becomes descendant of u.
    By parenthesis theorem, $d[u] < d[v] < \underline{f[v]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

Strongly Connected Component

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:



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.

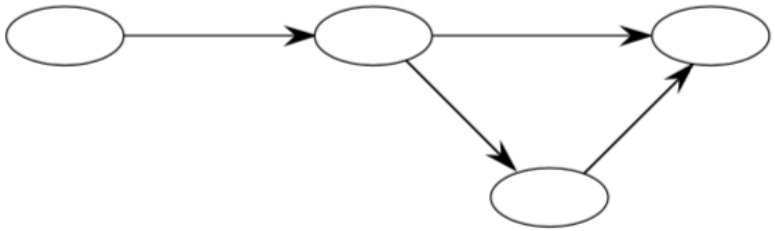
Observation: G and G^{T} have the *same* SCC's. (u and v are reachable from each other in G if and only if reachable from each other in G^{T}.)

Component Graph

Component graph

- $G^{SCC} = (V^{SCC}, E^{SCC})$.
- 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.

Proof Suppose there is a path $v' \rightsquigarrow v$ in G. Then there are paths $u \rightsquigarrow u' \rightsquigarrow v'$ and $v' \rightsquigarrow v \rightsquigarrow u$ in G. Therefore, u and v' are reachable from each other, so they are not in separate SCC's. ■ (lemma)

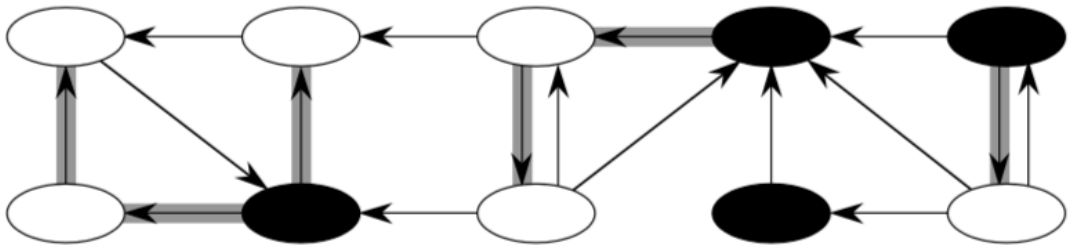
SCC(G)

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

Example

Example:

1. Do DFS
2. G^{T}
3. DFS (roots blackened)



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

How does it work

How can this possibly work?

Idea: By considering vertices in second DFS in decreasing order of finishing times from first DFS, we are visiting vertices of the component graph in topological sort order.

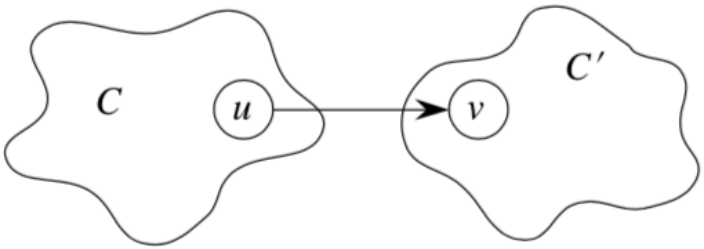
To prove that it works, first deal with 2 notational issues:

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

Property

Lemma

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



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

Proof Two cases, depending on which SCC had the first discovered vertex during the first DFS.

Proof

- If $d(C) < d(C')$, let x be the first vertex discovered in C. At time $d[x]$, all vertices in C and C' are white. Thus, there exist paths of white vertices from x to all vertices in C and C'.

  By the white-path theorem, all vertices in C and C' are descendants of x in depth-first tree.

  By the parenthesis theorem, $f[x] = f(C) > f(C')$.
- If $d(C) > d(C')$, let y be the first vertex discovered in C'. At time $d[y]$, all vertices in C' are white and there is a white path from y to each vertex in $C' \Rightarrow$ all vertices in C' become descendants of y. Again, $f[y] = f(C')$.

  At time $d[y]$, all vertices in C are white.

  By earlier lemma, since there is an edge (u, v), we cannot have a path from C' to C.

  So no vertex in C is reachable from y.

  Therefore, at time $f[y]$, all vertices in C are still white.

  Therefore, for all $w \in C$, $f[w] > f[y]$, which implies that $f(C) > f(C')$.

  ■ (lemma)

Properties

Corollary

Let C and C' be distinct SCC's in $G = (V, E)$. Suppose there is an edge $(u, v) \in E^{\mathrm{T}}$, where $u \in C$ and $v \in C'$. Then $f(C) < f(C')$.

Proof $(u, v) \in E^{\mathrm{T}} \Rightarrow (v, u) \in E$. Since SCC's of G and G^{T} are the same, $f(C') > f(C)$. ■ (corollary)

Corollary

Let C and C' be distinct SCC's in $G = (V, E)$, and suppose that $f(C) > f(C')$. Then there cannot be an edge from C to C' in G^{T}.

Proof It's the contrapositive of the previous corollary. ■

Now we have the intuition to understand why the SCC procedure works.

When we do the second DFS, on G^{T}, start with SCC C such that $f(C)$ is maximum. The second DFS starts from some $x \in C$, and it visits all vertices in C. Corollary says that since $f(C) > f(C')$ for all $C' \neq C$, there are no edges from C to C' in G^{T}.

Therefore, DFS will visit *only* vertices in C.

Which means that the depth-first tree rooted at x contains *exactly* the vertices of C.

The next root chosen in the second DFS is in SCC C' such that $f(C')$ is maximum over all SCC's other than C. DFS visits all vertices in C', but the only edges out of C' go to C, *which we've already visited.*

Therefore, the only tree edges will be to vertices in C'.

We can continue the process.

Each time we choose a root for the second DFS, it 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.

We are visiting vertices of $(G^{\mathrm{T}})^{\mathrm{SCC}}$ in reverse of topologically sorted order.

[The book has a formal proof.]

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.

Model as a graph:

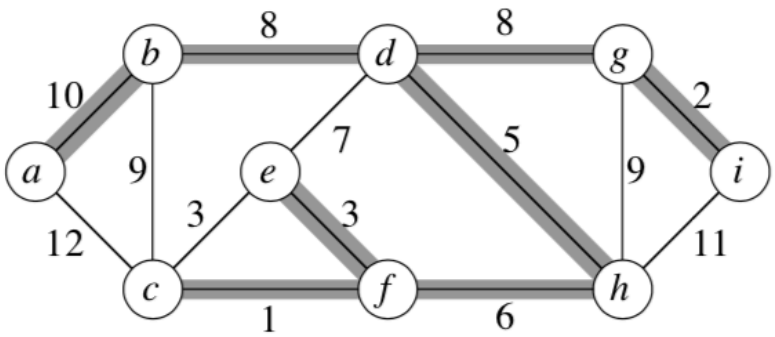
- Undirected graph $G = (V, E)$.
- ***Weight*** $w(u, v)$ on each edge $(u, v) \in E$.
- Find $T \subseteq E$ such that

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

Minimum Spanning Tree

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

Example of such a graph *[edges in MST are shaded]* :



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

Minimum Spanning Tree

Some properties of an MST:

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

Building up the solution

- We will build a set A of edges.
- Initially, A has no edges.
- As we add edges 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 we will add only safe edges.

Generic MST algorithm

GENERIC-MST(G, w)
$A \leftarrow \emptyset$
while A is not a spanning tree
  do find an edge (u, v) that is safe for A
    $A \leftarrow A \cup \{(u, v)\}$
return A

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

Initialization: The empty set trivially satisfies the loop invariant.

Maintenance: Since we add only safe edges, A remains a subset of some MST.

Termination: All edges added to A are in an MST, so when we stop, A is a spanning tree that is also an MST.

Finding a safe edge

How do we 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 set 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.)

Some definitions: Let $S \subset V$ and $A \subseteq E$.

Minimum Spanning Tree

- A ***cut*** $(S, V - S)$ is a partition of vertices into disjoint sets V and $S - V$.
- Edge $(u, v) \in E$ ***crosses*** cut $(S, V - S)$ if one endpoint is in S and the other is in $V - S$.
- A cut ***respects*** A if and only if no edge in A crosses the cut.
- An edge is a ***light edge*** crossing a cut if and only if its weight is minimum over all edges crossing the cut. For a given cut, there can be > 1 light edge crossing it.

Theorem

Let A be a subset of some MST, $(S, V - S)$ be a cut that respects A, and (u, v) be a light edge crossing $(S, V - S)$. Then (u, v) is safe for A.

Minimum Spanning Tree

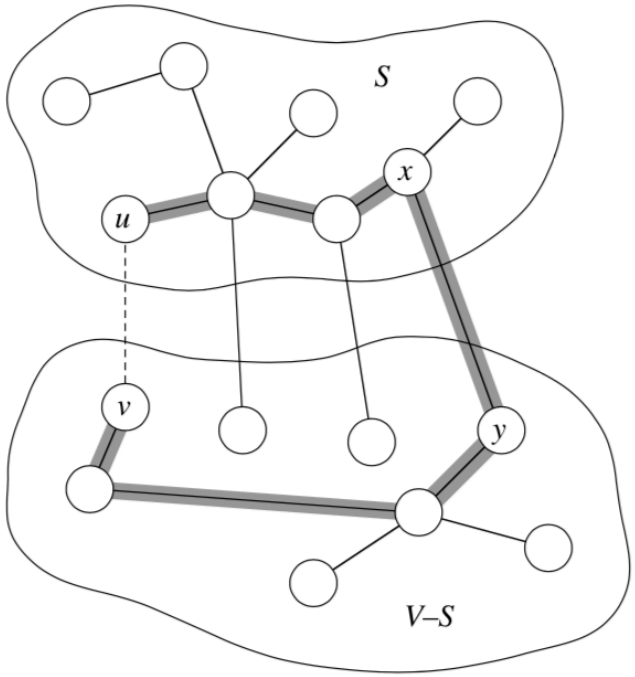
Proof Let T be an MST that includes A.

If T contains (u, v), done.

So now assume that T does not contain (u, v). We'll construct a different MST T' that includes $A \cup \{(u, v)\}$.

Recall: a tree has unique path between each pair of vertices. Since T is an MST, it contains a unique path p between u and v. Path p must cross the cut $(S, V - S)$ at least once. Let (x, y) be an edge of p that crosses the cut. From how we chose (u, v), must have $w(u, v) \leq w(x, y)$.

Minimum Spanning Tree



[Except for the dashed edge (u, v), all edges shown are in T. A is some subset of the edges of T, but A cannot contain any edges that cross the cut $(S, V - S)$, since this cut respects A. Shaded edges are the path p.]

Since the cut respects A, edge (x, y) is not in A.

To form T' from T:

- Remove (x, y). Breaks T into two components.
- Add (u, v). Reconnects.

Minimum Spanning Tree

So $T' = T - \{(x, y)\} \cup \{(u, v)\}$.

T' is a spanning tree.

$$\begin{aligned} w(T') &= w(T) - w(x, y) + w(u, v) \\ &\leq w(T) , \end{aligned}$$

since $w(u, v) \leq w(x, y)$. Since T' is a spanning tree, $w(T') \leq w(T)$, and T is an MST, then T' must be an MST.

Need to show that (u, v) is safe for A:

- $A \subseteq T$ and $(x, y) \notin A \Rightarrow A \subseteq T'$.
- $A \cup \{(u, v)\} \subseteq T'$.
- Since T' is an MST, (u, v) is safe for A. ■ (theorem)

Minimum Spanning Tree

So, in GENERIC-MST:

- A is a forest containing connected components. Initially, each component is a single vertex.
- Any safe edge merges two of these components into one. Each component is a tree.
- Since an MST has exactly $|V| - 1$ edges, the **for** loop iterates $|V| - 1$ times. Equivalently, after adding $|V|-1$ safe edges, we're down to just one component.

Corollary

If $C = (V_C, E_C)$ is a connected component in the forest $G_A = (V, A)$ and (u, v) is a light edge connecting C to some other component in G_A (i.e., (u, v) is a light edge crossing the cut $(V_C, V - V_C)$), then (u, v) is safe for A.

Proof Set $S = V_C$ in the theorem. ■ (corollary)

This naturally leads to the algorithm called Kruskal's algorithm to solve the minimum-spanning-tree problem.

Kruskal's algorithm

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

- 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.

Kruskal's algorithm

KRUSKAL(V, E, w)
$A \leftarrow \emptyset$
for each vertex $v \in V$
    do MAKE-SET(v)
sort E into nondecreasing order by weight w
for each (u, v) taken from the sorted list
    do if FIND-SET$(u) \neq$ FIND-SET(v)
        then $A \leftarrow A \cup \{(u, v)\}$
            UNION(u, v)
return A

Kruskal's algorithm

Run through the above example to see how Kruskal's algorithm works on it:

(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

At this point, we have only one component, so all other edges will be rejected. *[We could add a test to the main loop of* KRUSKAL *to stop once* $|V| - 1$ *edges have been added to A.]*

Get the shaded edges shown in the figure.

Suppose we had examined (c, e) *before* (e, f). Then would have found (c, e) safe and would have rejected (e, f).

Kruskal's algorithm

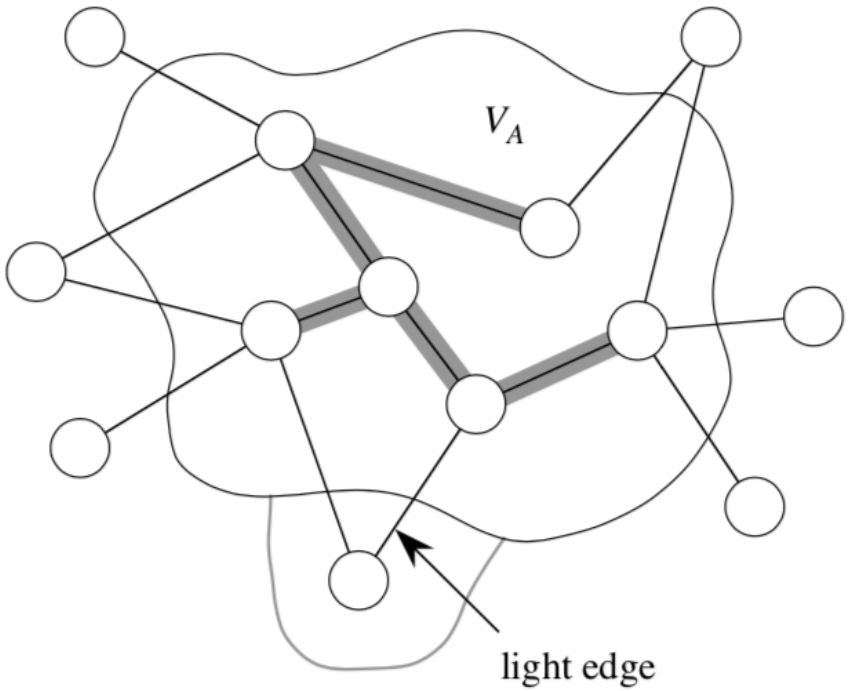
Analysis

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

Total: $O(V + E \lg E)$
If edges sorted, 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 crossing cut $(V_A, V - V_A)$, where $V_A =$ vertices that A is incident on. Add this edge to A.



[Edges of A are shaded.]

Prim's algorithm

How to find the light edge quickly?

Use a priority queue Q:

- Each object is a vertex in $V - V_A$.
- Key of v is minimum weight of any edge (u, v), where $u \in V_A$.
- Then the vertex returned by EXTRACT-MIN is v such that there exists $u \in V_A$ and (u, v) is light edge crossing $(V_A, V - V_A)$.
- Key of v is ∞ if v is not adjacent to any vertices in V_A.

The edges of A will form a rooted tree with root r:

- r is given as an input to the algorithm, but it can be any vertex.
- Each vertex knows its parent in the tree by the attribute $\pi[v]$ = parent of v. $\pi[v] = \text{NIL}$ if $v = r$ or v has no parent.
- As algorithm progresses, $A = \{(v, \pi[v]) : v \in V - \{r\} - Q\}$.
- At termination, $V_A = V \Rightarrow Q = \emptyset$, so MST is $A = \{(v, \pi[v]) : v \in V - \{r\}\}$.

Prim's algorithm

```
PRIM(V, E, w, r)
Q ← ∅
for each u ∈ V
    do key[u] ← ∞
       π[u] ← NIL
       INSERT(Q, u)
DECREASE-KEY(Q, r, 0)        ▷ key[r] ← 0
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       for each v ∈ Adj[u]
           do if v ∈ Q and w(u, v) < key[v]
                 then π[v] ← u
                      DECREASE-KEY(Q, v, w(u, v))
```

Do example from previous graph. *[Let a student pick the root.]*

Prim's algorithm

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 we could do DECREASE-KEY in $O(1)$ *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 do DECREASE-KEY in $O(1)$ amortized time: Fibonacci heaps, in Chapter 20.

Homework 6

1. Exercise 22.2-1
2. Exercise 22.3-2
3. Exercise 22.4-3
4. Exercise 22.5-2
5. Problem 23-1, question a. and b.
6. Exercise 24.3-1

Due date: Dec 14, Thu