Topological Sort

Topological Sort

- A topological sort of a directed acyclic graph is an ordering of all its vertices such that if G contains an edge (u,v), then u appears before v in the ordering.
- (If the graph contains a cycle, then no linear ordering is possible.)

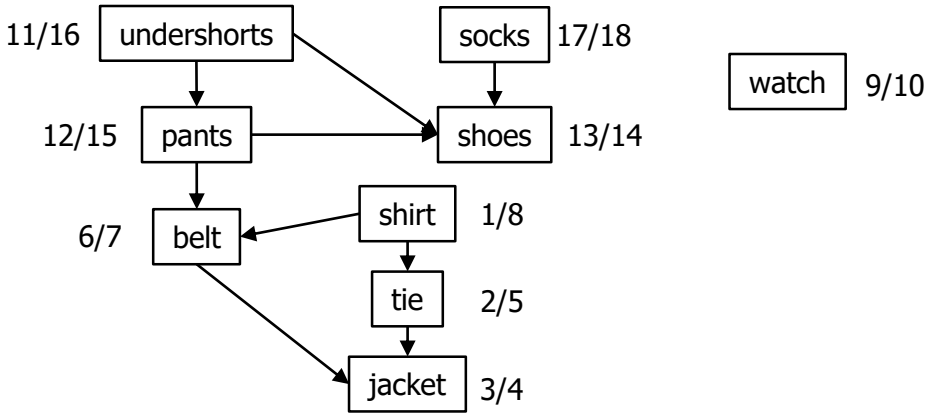
TOPOLOGICAL-SORT(G)

1. call DFS(G) to compute finishing 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

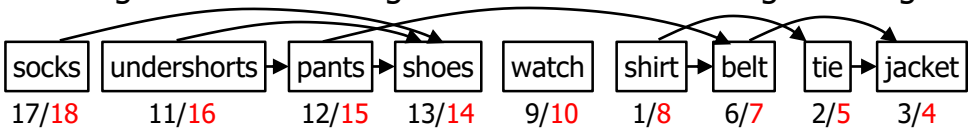
- Running time of DFS : $\Theta(|V|+|E|)$

Topological Sort

- Each directed edge (u,v) means that garment u must be put on before garment v



- The same graph shown topologically sorted, with its vertices arranged from left to right in order of decreasing finishing time.



Lemma 22.11

- A directed graph G is acyclic if and only if a depth-first search of G yields no back edges.

Lemma 22.11

- A directed graph G is acyclic if and only if a depth-first search of G yields no back edges
- Proof (⇒):
  - Assume that (u,v) is a back edge.
  - Then, vertex v is an ancestor of vertex u in the depth-first forest.
  - Thus, G contains a path p from v to u.
  - With the back edge (u, v) and p, G has a cycle.

Lemma 22.11

- A directed graph G is acyclic if and only if a depth-first search of G yields no back edges
- Proof ($\Leftarrow$):
  - Assume that G has a cycle c.
  - We show a depth-first search of G yields a back edge.
  - Let v be the first vertex to be discovered in c, (u,v) be the preceding edge in c.
  - At time v.d, the cycle c forms a white path from v to u.
  - By the white-path theorem, vertex u becomes a descendant of v in the depth-first forest.
  - Therefore (u,v) is a back edge.

Theorem 22.9 (White-path Theorem)
In a depth-first forest of a (directed or undirected) graph G=(V,E), vertex v is a descendant of vertex u if and only if at the time u.d that the search discovers u, vertex v can be reached from u along a path consisting entirely of white vertices.

Theorem 22.12

- TOPOLOGICAL-SORT(G) produces a topological sort of the directed acyclic graph provided as its input.

Theorem 22.12

- TOPOLOGICAL-SORT(G) produces a topological sort of the directed acyclic graph provided as its input.
- Proof:
  - Assume that DFS is run on a given dag G=(V,E) to determine finishing times for its vertices.
  - It suffices to show that for any pair of distinct vertices u, v $\in$ V, if G contains an edge from u to v, then v.f <u.f
  - When (u,v) is explored, v can not be gray because it would make a cycle.
  - If v is white, v.f<u.f because v is descendant of u.
  - If v is black, v.f<u.f because v.f has already been set and we have yet to assign a timestamp to u.f.
  - Thus, for any edge (u,v) in the dag, we have v.f<u.f.

Strongly Connected Components

Strongly Connected Components

- Many algorithms that work with directed graphs begin with such a decomposition.
- After decomposing the graph into strongly connected components, such algorithms run separately on each one and then combine the solutions according to the structure of connections among components.

Strongly Connected Components

- A strongly connected component (SCC) of a directed graph G is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices u and v in C, vertices u and v are reachable from each other
- To find SCCs, call DFS twice, one on G and one on G^T
  - Transpose of G: $G^T=(V, E^T)$, where $E^T=\{(u, v): (v, u) \in E\}$
  - Time Complexity: $\Theta(|V|+|E|)$
- G and G^T have exactly the same strongly connected components
  - u and v are reachable from each other in G if and only if they are reachable from each other in G^T.

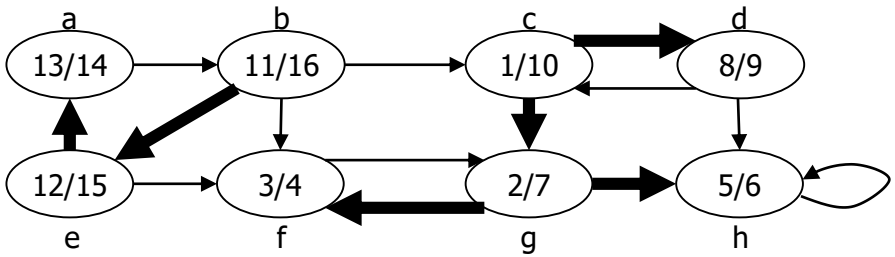
Strongly Connected Components

STRONGLY-CONNECTED-COMPONENTS(G)

1. call DFS(G) to compute finishing times u.f for each vertex u
2. compute G^T
3. call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing u.f (as computed in line 1)
4. output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

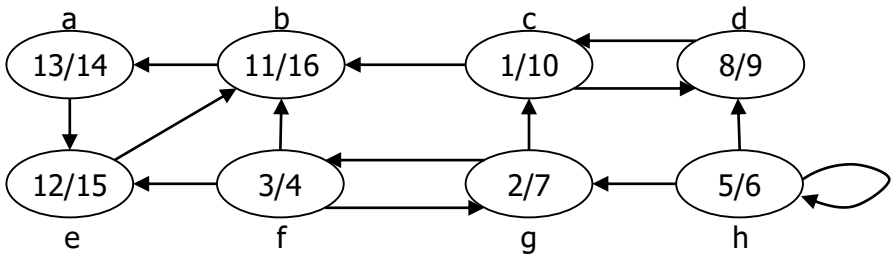
Strongly Connected Components

- Call DFS(G) to compute finishing times



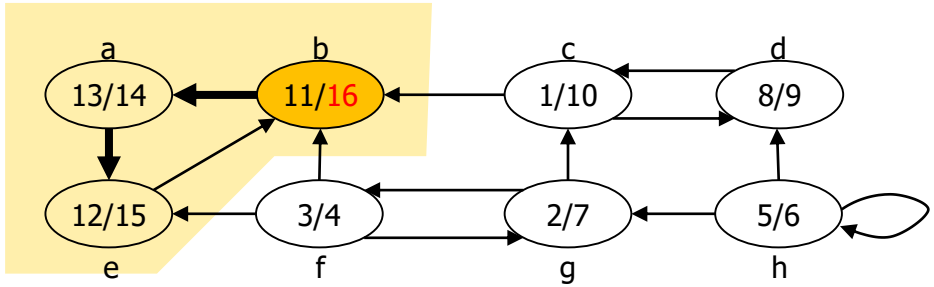
Strongly Connected Components

- Compute G^T, transpose of G



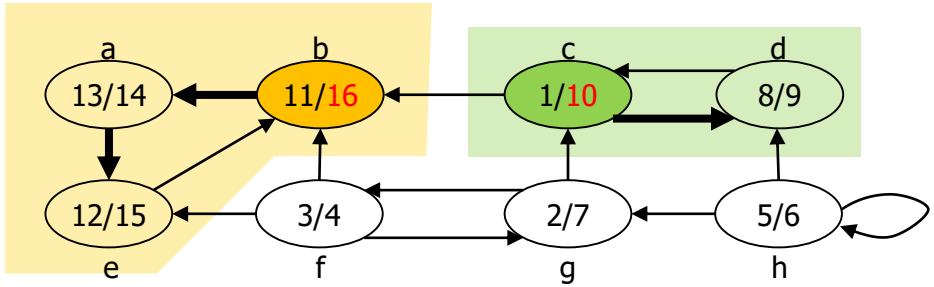
Strongly Connected Components

- Call DFS(G^T) in order of decreasing $u.f$



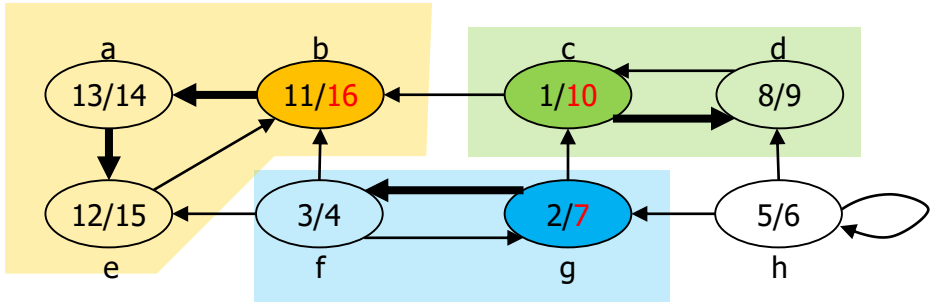
Strongly Connected Components

- Call DFS(G^T) in order of decreasing $u.f$



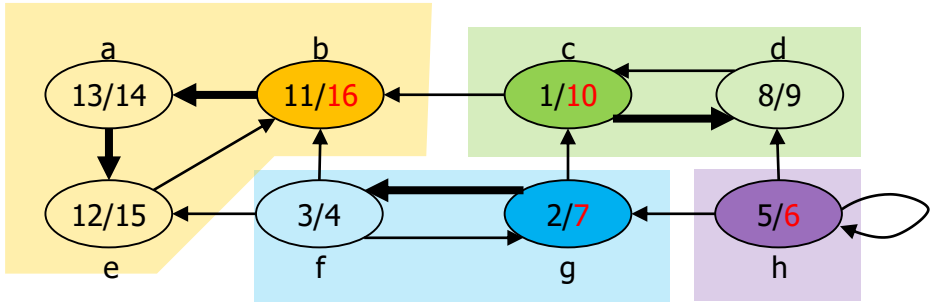
Strongly Connected Components

- Call DFS(G^T) in order of decreasing $u.f$



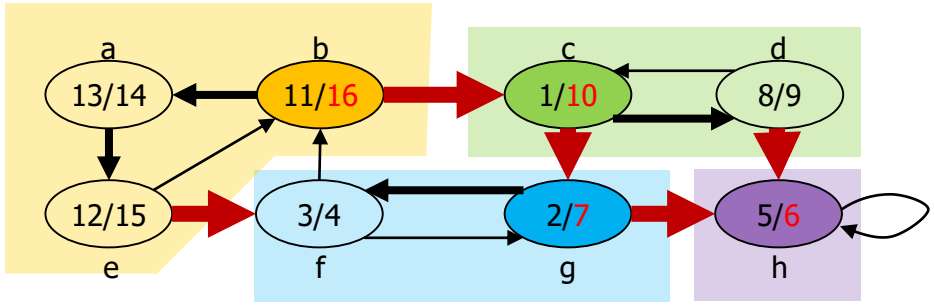
Strongly Connected Components

- Call DFS(G^T) in order of decreasing $u.f$



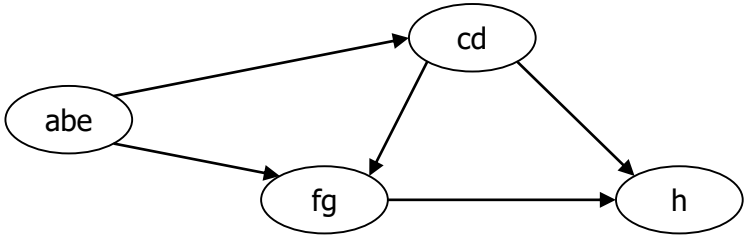
Strongly Connected Components

- Output the strongly connected component



Strongly Connected Components

- Output the strongly connected component

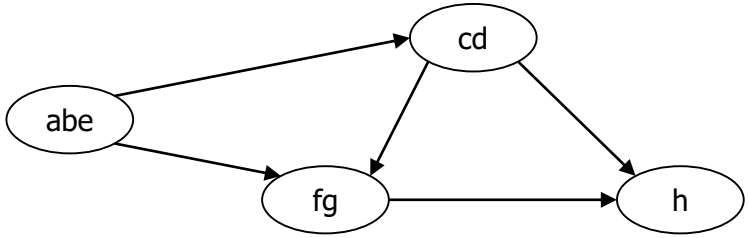


Strongly Connected Components

- A strongly connected component (SCC) of a directed graph G is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices u and v in C, vertices u and v are reachable from each other
- Component graph $G^{SCC}=(V^{SCC}, E^{SCC})$ (DAG)
  - Suppose that G has strongly connected components $C_1, C_2, \ldots, C_k$.
  - The vertex set V^{SCC} is $\{v_1, v_2, \ldots, v_k\}$, and it contains a vertex v_i for each strongly connected component C_i of G.
  - There is an edge $(v_i, v_j) \in E^{SCC}$ if G contains a directed edge (x, y) for some $x \in C_i$ and some $y \in C_j$.

Strongly Connected Components

- The acyclic component graph G^{SCC} obtained by contracting all edges within each strongly connected component of G so that only a single vertex remains in each component.



Strongly Connected Components

- Lemma 22.13
  - Let C and C′ be distinct strongly connected components in directed graph G=(V,E), let u,v∈C, let u′,v′∈C′, and suppose that G contains a path u~>u′.
  - Then G cannot also contain a path v′~>v

Strongly Connected Components

- Lemma 22.13
  - Let C and C′ be distinct strongly connected components in directed graph G=(V,E), let u,v∈C, let u′,v′∈C′, and suppose that G contains a path u~>u′.
  - Then G cannot also contain a path v′~>v
- Proof
  - If G contains a path v′~>v, then it contains paths u~>u′~>v′ and v′~>v~>u.
  - Thus, u and v′ are reachable from each other, thereby contradicting the assumption that C and C′ are distinct strongly connected components.

Strongly Connected Components

- Let $d(C)=\min_{u \in C}\{u.d\}$ and $f(C)=\max_{u \in C}\{u.f\}$
- Lemma 22.14
  - Let C and C' be distinct strongly connected components in directed graph G=(V,E).
  - Suppose that there is an edge $(u,v) \in E$, where $u \in C$ and $v \in C'$.
  - Then, $f(C)>f(C')$.

Strongly Connected Components

- Let $d(C)=\min_{u\in C}\{u.d\}$ and $f(C)=\max_{u\in C}\{u.f\}$
- Lemma 22.14
  - Let C and C′ be distinct strongly connected components in directed graph G=(V,E).
  - Suppose that there is an edge $(u,v)\in E$, where $u\in C$ and $v\in C'$.
  - Then, $f(C)>f(C')$.
- Proof of Lemma 22.14 - when $d(C)<d(C')$,
  - Let x be the first vertex discovered in C.
  - At time x.d, all vertices in C and C′ are white.
  - At that time, G contains a path from x to each vertex in C consisting only of white vertices.
  - Because $(u,v)\in E$, for any vertex $w\in C'$, there is also a path in G at time x.d from x to w consisting only of white vertices : $x \rightsquigarrow u \rightarrow v \rightsquigarrow w$.
  - By the white-path theorem, all vertices in C and C′ become descendants of x in the depth-first tree.
  - By Corollary 22.8, x has the latest finishing time of any of its descendants, and so $x.f=f(C)>f(C')$.

Theorem 22.9 (White-path Theorem)
In a depth-first forest of a (directed or undirected) graph G=(V,E), vertex v is a descendant of vertex u if and only if at the time u.d that the search discovers u, vertex v can be reached from u along a path consisting entirely of white vertices.

Strongly Connected Components

- Let $d(C)=\min_{u\in C}\{u.d\}$ and $f(C)=\max_{u\in C}\{u.f\}$
- Lemma 22.14
  - Let C and C′ be distinct strongly connected components in directed graph G=(V,E).
  - Suppose that there is an edge $(u,v)\in E$, where $u\in C$ and $v\in C'$.
  - Then, $f(C)>f(C')$.
- Proof of Lemma 22.14 - when $d(C)>d(C')$,
  - Let y be the first vertex discovered in C′.
  - At time y.d, all vertices in C′ are white and G contains a path from y to each vertex in C′ consisting only of white vertices.
  - By the white-path theorem, all vertices in C′ become descendants of y in the depth-first tree.
  - By corollary 22.8, y.f=f(C′).
  - At time y.d, all vertices in C are white.
  - Since there is an edge (u,v) from C to C′, Lemma 22.13 implies that there cannot be a path from C′ to C.
  - Hence, no vertex in C is reachable from y.
  - At time y.f all vertices in C are still white.
  - Thus, any vertex w in C, w.f>y.f, meaning $f(C)>f(C')$.

Strongly Connected Components

- Corollary 22.15
  - Let C and C′ be distinct strongly connected components in directed graph G=(V,E).
  - Suppose that there is an edge $(u,v) \in E^T$, where $u \in C$ and $v \in C'$.
  - Then $f(C) < f(C')$

Strongly Connected Components

- Proof of Corollary 22.15
  - Since $(u,v) \in E^T$, we have $(v,u) \in E$.
  - Because the strongly connected components of G and G^T are the same, Lemma 22.14 implies that $f(C) < f(C')$.

Strongly Connected Components

- Theorem 22.16
  - The STRONGLY-CONNECTED-COMPONENTS procedure correctly computes the strongly connected components of the directed graph G provided as its input.

Strongly Connected Components

- Proof of Theorem 22.16
- We argue by induction on the number of depth-first trees found in the depth-first search of G^T that the vertices of each tree form a strongly connected component.
- Inductive hypothesis
  - The first k trees produced are strongly connected components.
- Basis case
  - When k = 0, it is trivial.

Strongly Connected Components

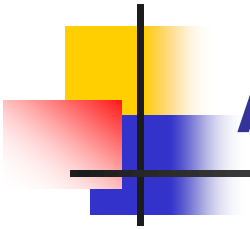
- Proof of Theorem 22.16 (continued)
- Inductive step
  - Assume that each of the first k depth-first trees is a strongly connected component, and we consider the (k+1)-st tree.
  - Let the root of this tree be vertex u, and let u be in a strongly connected component (SCC) C.
  - Because we choose roots in decreasing order of finishing time, u.f = f(C) > f(C′) for any SCC C′ other than C that has yet to be visited.
  - By the inductive hypothesis, at the time that the search visits u, all other vertices of C are white.
  - By the white-path theorem, therefore, all other vertices of C are descendants of u in its depth-first tree.

Strongly Connected Components

- Proof of Theorem 22.16 (continued)
- Inductive step (continued)
  - Moreover, by the inductive hypothesis and by Corollary 22.15, any edges in G^T that leave C must be to SCCs that have already been visited.
  - Thus, no vertex in any strongly connected component other than C will be a descendant of u during the depth-first search of G^T.
  - Thus, the vertices of the depth-first tree in G^T that is rooted at u form exactly one strongly connected component, which completes the inductive step and the proof.

Corollary 22.15

Let C and C′ be distinct strongly connected components in directed graph G=(V,E). Suppose that there is an edge $(u,v) \in E^T$, where $u \in C$ and $v \in C'$. Then $f(C) < f(C')$

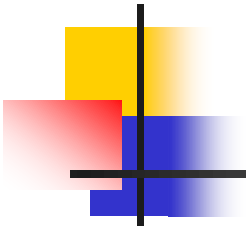


Any Question?

Introduction to Algorithms (Chapter 23: Minimum Spanning Trees)

Kyuseok Shim

Electrical and Computer Engineering Seoul National University



Outline

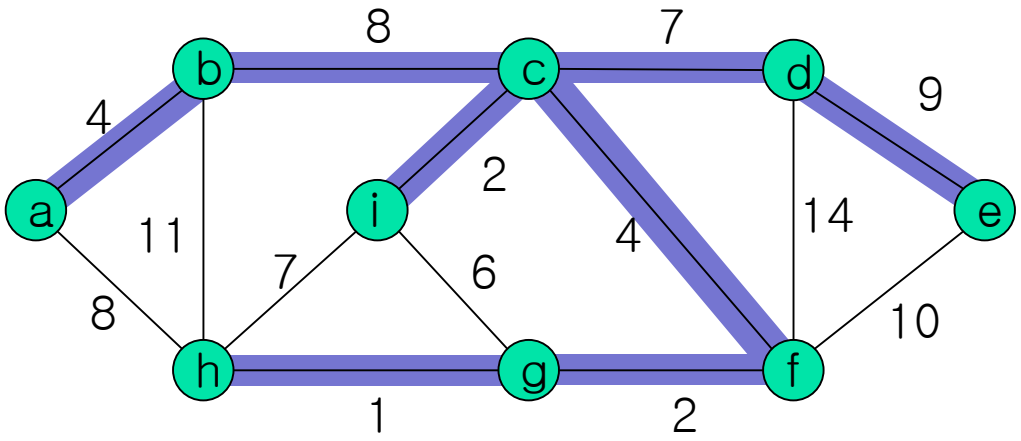
- In this chapter, we shall examine two algorithms for solving the minimum spanning-tree problem: Kruskal's algorithm and Prim's algorithm.

Minimum Spanning Tree

- In the design of electronic circuitry, it is often necessary to make the pins of several components electrically equivalent by wiring together.
- To interconnect a set of n pins, we can use an arrangement of n-1 pins.
- Of all such arrangements, the one using the least amount of wire is the most desirable.

Minimum Spanning Tree

- Given a connected, undirected, weighted graph
- Find a spanning tree using edges that minimizes the total weight



Minimum Spanning Tree

- We can model this wiring problem with a connected, undirected graph G=(V,E), where
  - V is the set of pins
  - E is the set of possible interconnections between pair of pins
  - A weight w(u,v) for each edge $(u,v) \in E$ that specifying the cost to connect u and v
- Find an acyclic subset $T \subseteq E$ that connects all of the vertices and whose total weight $w(T) = \sum_{(u,v) \in T} w(u,v)$ is minimized.
- Since T is acyclic and connects all the vertices, we call a minimum spanning tree.

Minimum Spanning Tree

- We first introduce a "generic" minimum-spanning-tree method that grows a spanning tree by adding one edge at a time.
- We shall examine two greedy algorithms for solving the minimum spanning-tree problem.
- The greedy strategy advocates making the choice that is the best at the moment.
- We prove that certain greedy strategies do yield a spanning tree with minimum weight.
- We make both Kruskal's algorithm and Prim's algorithm run in time O(E lg V) using ordinary binary heaps.
- By using Fibonacci heaps, Prim's algorithm runs in time O(E + V lg V), which improves over the binary-heap implementation if |V| is much smaller than |E|.

Minimum Spanning Tree

- Let A be a subset of a minimum spanning tree .
- An edge (u,v) is safe if we can add it to A while maintaining that $A \cup \{(u, v)\}$ is also a subset of a minimum spanning tree.

GENERIC-MST(G, w)

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

Minimum Spanning Tree

- Loop Invariant
  - Prior to each iteration, A is a subset of some minimum spanning tree.
- Initialization
  - After line 1, the set A = Ø trivially satisfies the loop invariant.

```
GENERIC-MST(G, w)
1.  A = Ø
2.  while A does not form a spanning tree
3.      find an edge (u,v) that is safe for A
4.      A = A U {(u,v)}
5.  return A
```

Minimum Spanning Tree

- Loop Invariant
  - Prior to each iteration, A is a subset of some minimum spanning tree.
- Maintenance
  - The loop in lines 2-4 maintains the invariant by adding only safe edges.

GENERIC-MST(G, w)

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

Minimum Spanning Tree

- Loop Invariant
  - Prior to each iteration, A is a subset of some minimum spanning tree.
- Termination
  - All edges added to A are in a minimum spanning tree, and so the set A returned in line 5 must be a minimum spanning tree.

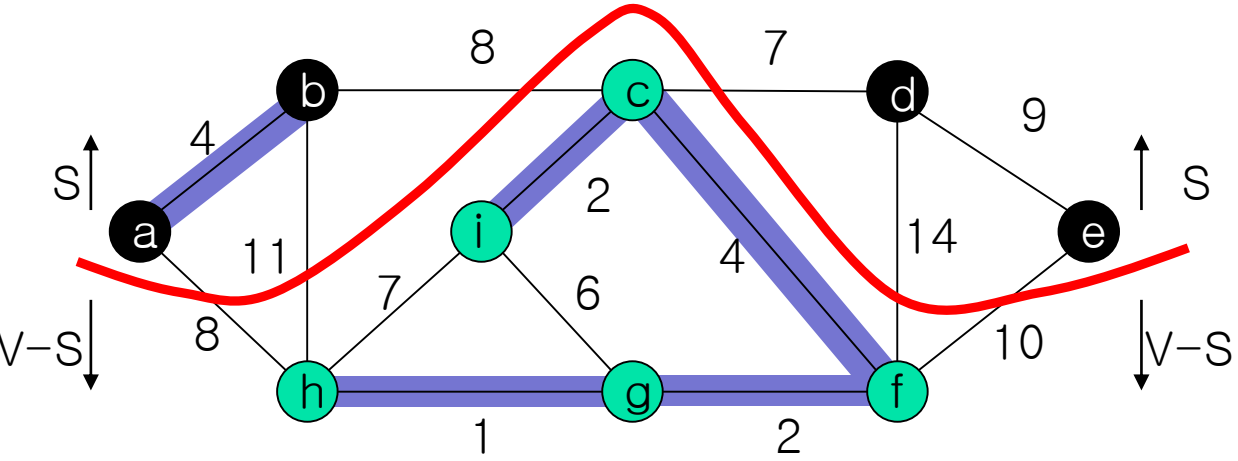
GENERIC-MST(G, w)

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

Minimum Spanning Tree

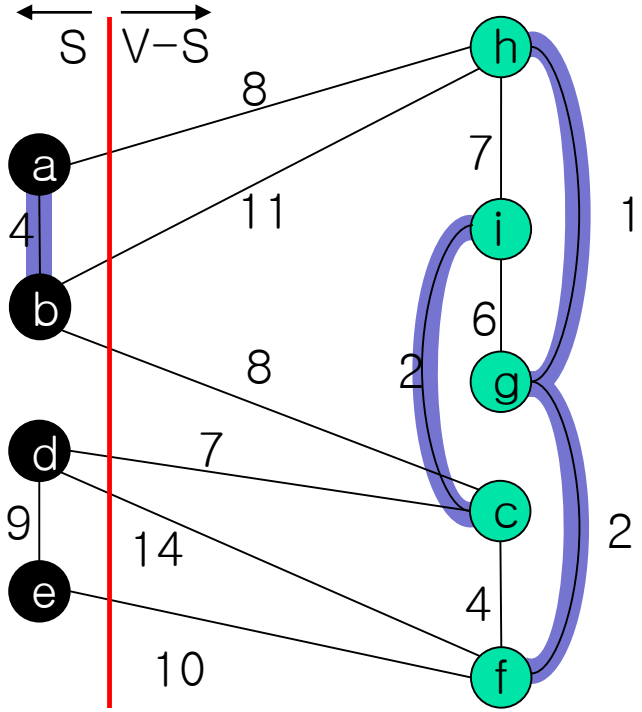
- If A U {(u,v)} is also a subset of a minimum spanning tree, we call the edge (u,v) a safe edge.
- A cut (S, V-S) of an undirected graph G=(V,E) is a partition of V.
- An edge (u,v)$\in$E crosses the cut (S, V-S) if one of its endpoints is in S and the other is in V-S.
- A cut respects a set A of edges 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.

One Way of Viewing a Cut (S, V-S)



- Black vertices are in the set S, and green vertices are in V-S.
- The edge (d,c) is the unique light edge crossing the cut.

Another Way of Viewing a Cut (S, V-S)



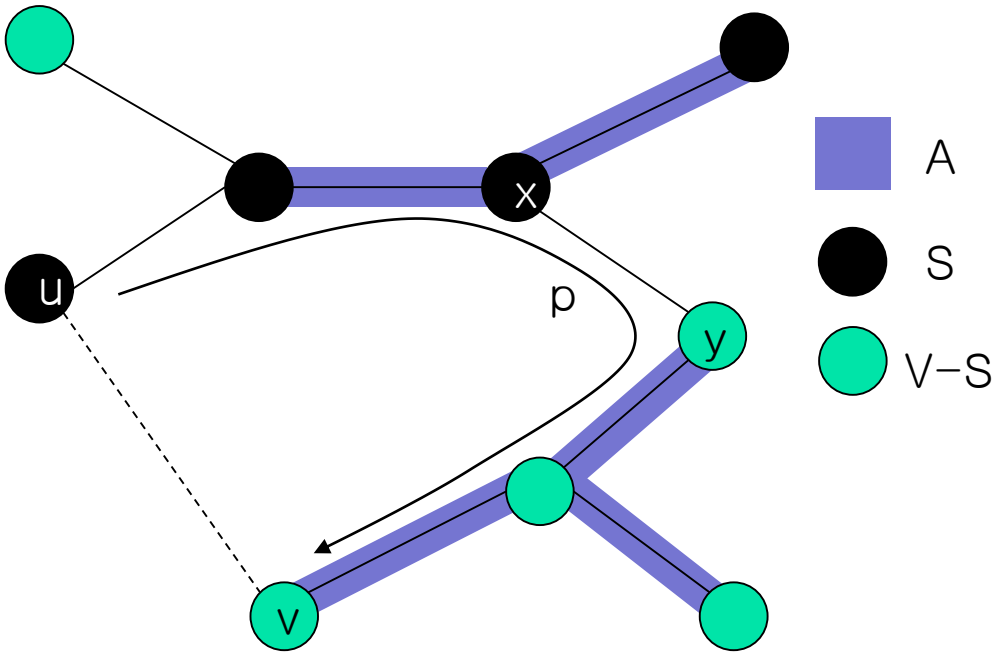
Theorem 23.1

- 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, the edge (u,v) is safe for A

Theorem 23.1 (Proof)

- Let T be a minimum spanning tree that includes A
- Assume that T does not contain the light edge (u,v)
- Construct another minimum spanning tree T' that include A U {(u,v)} by using a cut-and-paste technique, thereby showing that (u,v) is a safe edge for A

Theorem 23.1 (Proof)



- The edge (x, y) is an edge on the unique simple path p from u to v in T and the edges in A are shaded.

Theorem 23.1 (Proof)

- The light edge (u,v) forms a cycle with the edge on the simple path p from u to v in T.
- Since u and v are on opposite sides of the cut (S, V-S), there is at least one edge in T on the simple path p that also crosses the cut. Let (x,y) be any such edge.
- The edge (x, y) is not in A, since the cut respects A.
- Since (x,y) is on the unique path from u to v in T, removing (x,y) breaks T into two components.
- Adding (u,v) re-connects them to form a new spanning tree T' = T – {(x,y)} U{(u,v)}.

Theorem 23.1 (Proof)

- We next show that $T' = T - \{(x,y)\} \cup \{(u,v)\}$ is a minimum spanning tree.
- Since (u, v) is a light edge crossing (S, V-S) and (x, y) also crosses this cut, we have $w(u,v) \leq w(x, y)$ resulting that $w(T') = w(T) - w(x,y) + w(u,v) \leq w(T)$.
- But $w(T) \leq w(T')$, since T is a minimum spanning tree.
- Thus, T′ must be a minimum spanning tree.
- Because $A \subseteq T'$ and $A \cup \{(u,v)\} \subseteq T'$ where T′ is a minimum spanning tree, (u,v) is safe for A.