Algorithms: UNION FIND AND MINIMUM SPANNING TREES

Alessandro Chiesa, Ola Svensson

EPFL School of Computer and Communication Sciences

Lecture 19, 29.04.2025

Recall: The Ford-Fulkerson Method

Max-flow

Start with 0-flow
while there is an augmenting path from s to t in residual network **do**

- Find augmenting path
- Compute bottleneck= min capacity on path
- Increase flow on the path by the bottleneck

When finished, resulting flow is maximal

Recall: The Ford-Fulkerson Method

Max-flow

Start with 0-flow
while there is an augmenting path from s to t in residual network **do**

- Find augmenting path
- Compute bottleneck= min capacity on path
- Increase flow on the path by the bottleneck

When finished, resulting flow is maximal

Min-cut

If no augmenting path exists in residual network, then

- Find set of nodes S reachable from s in residual network
- Set $T = V \setminus S$

S and T define a minimum cut

Recall: The Ford-Fulkerson Method

Max-flow

Start with 0-flow
while there is an augmenting path from s to t in residual network **do**

- Find augmenting path
- Compute bottleneck= min capacity on path
- Increase flow on the path by the bottleneck

When finished, resulting flow is maximal

Min-cut

If no augmenting path exists in residual network, then

- Find set of nodes S reachable from s in residual network
- Set $T = V \setminus S$

S and T define a minimum cut

Max-flow = Min-cut (learn the proof)

Recall: The Ford-Fulkerson Method

Max-flow

Start with 0-flow
while there is an augmenting path from s to t in residual network **do**

- Find augmenting path
- Compute bottleneck= min capacity on path
- Increase flow on the path by the bottleneck

When finished, resulting flow is maximal

Min-cut

If no augmenting path exists in residual network, then

- Find set of nodes S reachable from s in residual network
- Set $T = V \setminus S$

S and T define a minimum cut

Max-flow = Min-cut (learn the proof)

Gives a way to verify that the step-by-step calculations of the flow are correct!

Recall: The Ford-Fulkerson Method

Max-flow

Start with 0-flow
while there is an augmenting path from s to t in residual network **do**

- Find augmenting path
- Compute bottleneck= min capacity on path
- Increase flow on the path by the bottleneck

When finished, resulting flow is maximal

Min-cut

If no augmenting path exists in residual network, then

- Find set of nodes S reachable from s in residual network
- Set $T = V \setminus S$

S and T define a minimum cut

Max-flow = Min-cut (learn the proof)

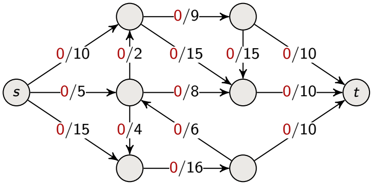
Gives a way to verify that the step-by-step calculations of the flow are correct!

The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

1. **Initialize flow f to 0**
2. **while** exists an augmenting path p in the residual network G_f
3. augment flow f along p
4. **return** f

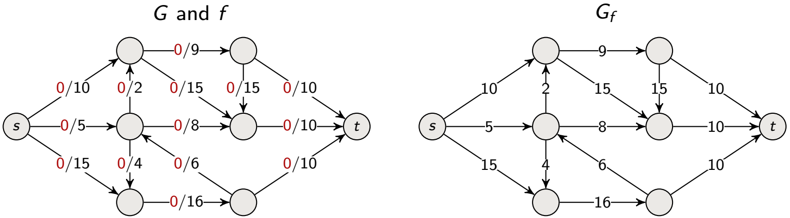
G and f



The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

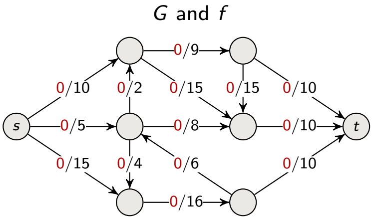
1. Initialize flow f to 0
2. **while exists an augmenting path p in the residual network G_f**
3. augment flow f along p
4. **return** f

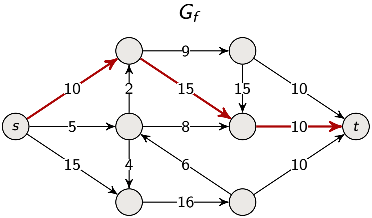


The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

1. Initialize flow f to 0
2. **while exists an augmenting path p in the residual network G_f**
3.   augment flow f along p
4. **return** f

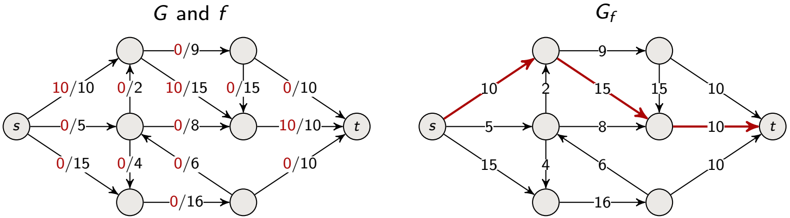




The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

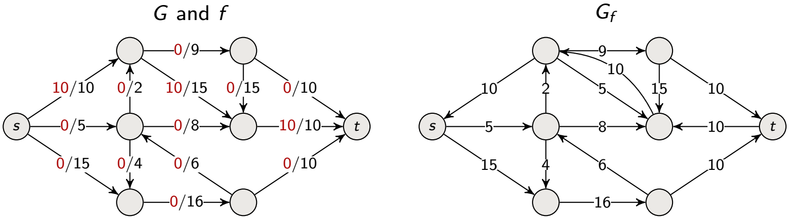
1. Initialize flow f to 0
2. **while** exists an augmenting path p in the residual network G_f
3. **augment flow f along p**
4. **return** f



The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

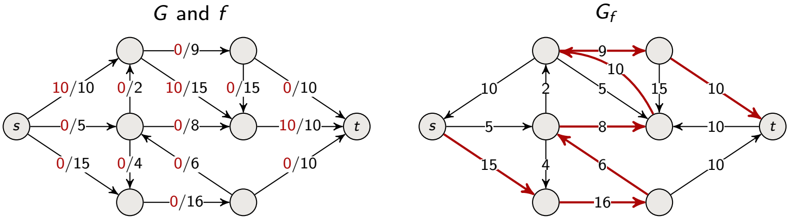
1. Initialize flow f to 0
2. **while exists an augmenting path p in the residual network G_f**
3. augment flow f along p
4. **return** f



The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

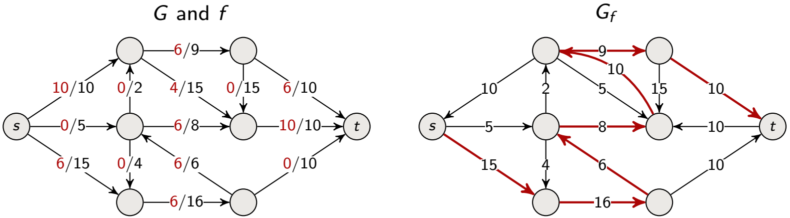
1. Initialize flow f to 0
2. **while exists an augmenting path p in the residual network G_f**
3. augment flow f along p
4. **return** f



The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

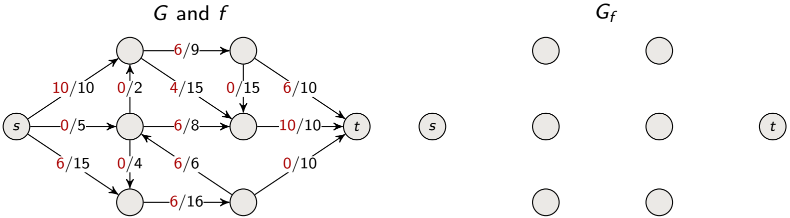
1. Initialize flow f to 0
2. **while** exists an augmenting path p in the residual network G_f
3. **augment flow f along p**
4. **return** f



The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

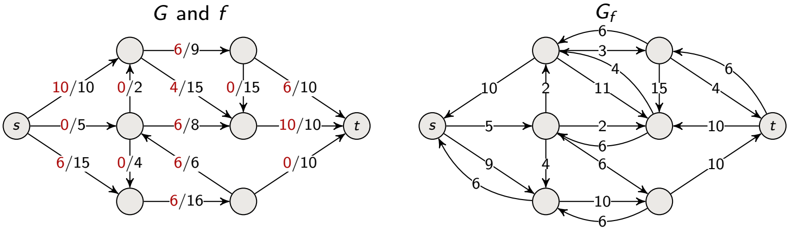
1. Initialize flow f to 0
2. **while exists an augmenting path p in the residual network G_f**
3. augment flow f along p
4. **return** f



The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

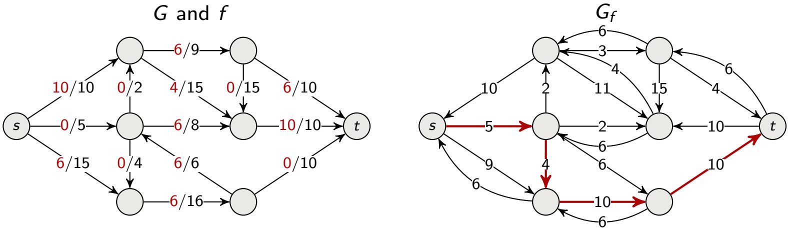
1. Initialize flow f to 0
2. **while exists an augmenting path p in the residual network G_f**
3. augment flow f along p
4. **return** f



The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

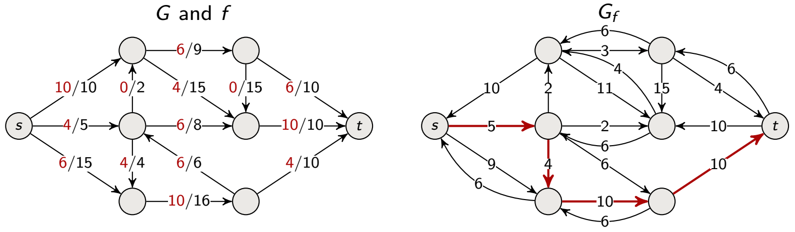
1. Initialize flow f to 0
2. **while exists an augmenting path p in the residual network G_f**
3. augment flow f along p
4. **return** f



The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

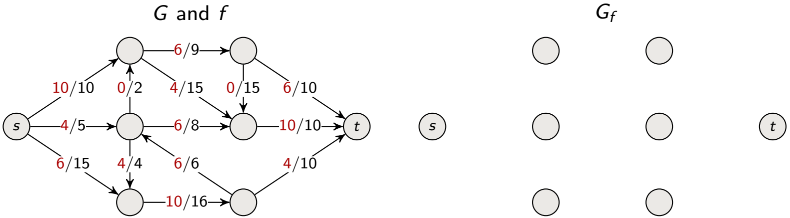
1. Initialize flow f to 0
2. **while** exists an augmenting path p in the residual network G_f
3. **augment flow f along p**
4. **return** f



The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

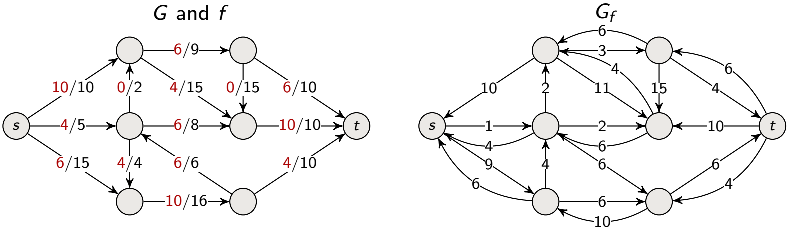
1. Initialize flow f to 0
2. **while exists an augmenting path p in the residual network G_f**
3. augment flow f along p
4. **return** f



The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

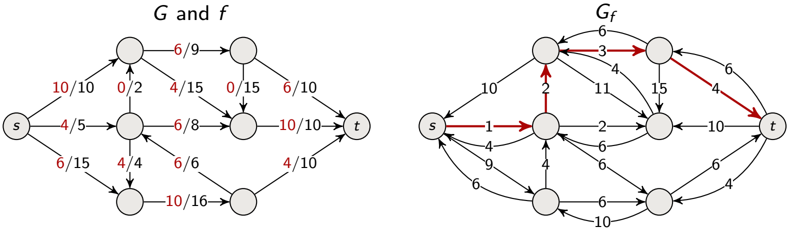
1. Initialize flow f to 0
2. **while exists an augmenting path p in the residual network G_f**
3. augment flow f along p
4. **return** f



The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

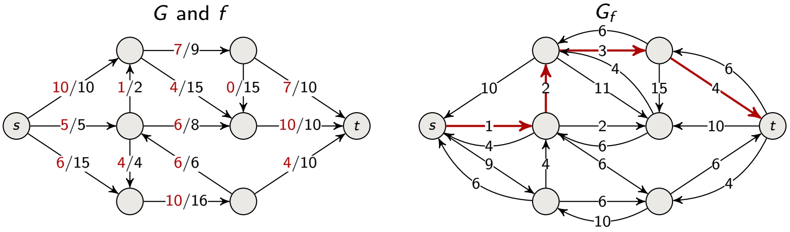
1. Initialize flow f to 0
2. **while exists an augmenting path p in the residual network G_f**
3. augment flow f along p
4. **return** f



The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

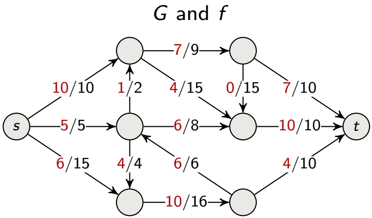
1. Initialize flow f to 0
2. **while** exists an augmenting path p in the residual network G_f
3. **augment flow f along p**
4. **return** f

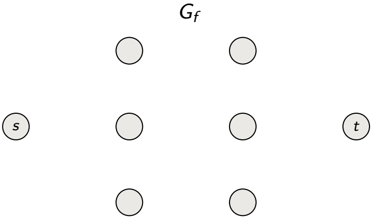


The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

1. Initialize flow f to 0
2. **while exists an augmenting path p in the residual network G_f**
3. augment flow f along p
4. **return f**

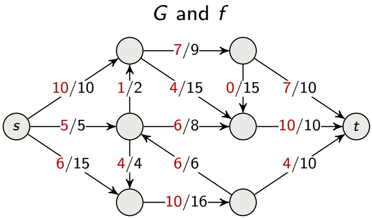


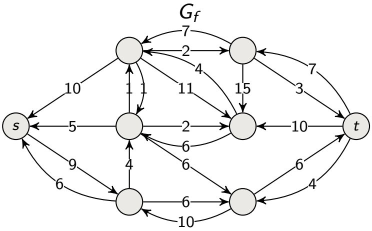


The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

1. Initialize flow f to 0
2. **while exists an augmenting path p in the residual network G_f**
3. augment flow f along p
4. **return** f

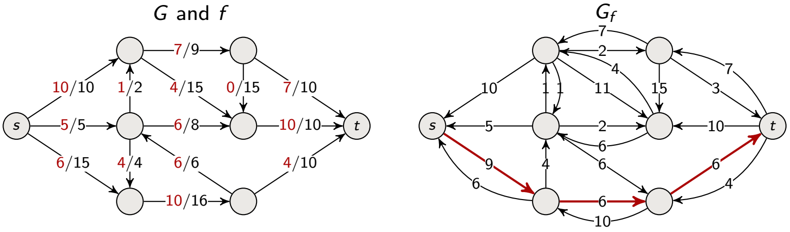




The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

1. Initialize flow f to 0
2. **while exists an augmenting path p in the residual network G_f**
3. augment flow f along p
4. **return** f

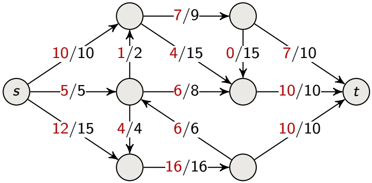


The Ford-Fulkerson Method'54

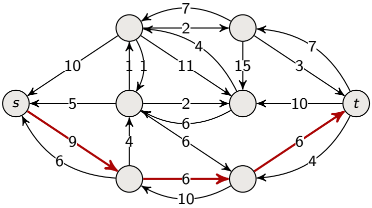
FORD-FULKERSON-METHOD(G, s, t):

1. Initialize flow f to 0
2. **while** exists an augmenting path p in the residual network G_f
3. **augment flow f along p**
4. **return** f

G and f



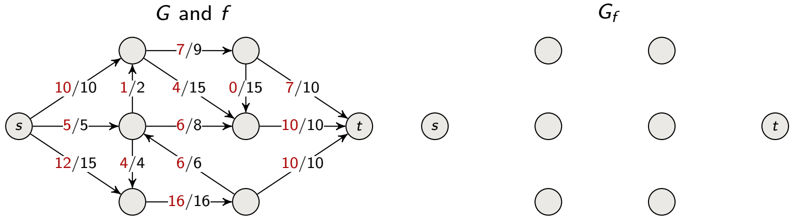
G_f



The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

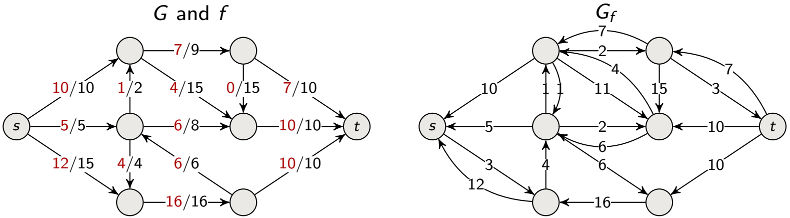
1. Initialize flow f to 0
2. **while exists an augmenting path p in the residual network G_f**
3. augment flow f along p
4. **return** f



The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

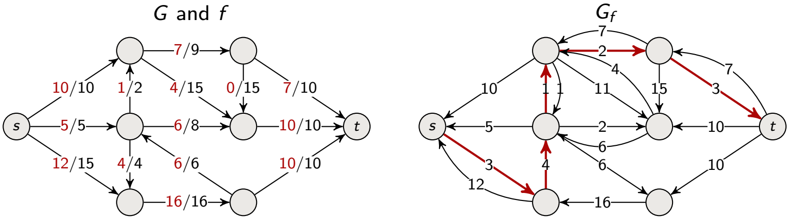
1. Initialize flow f to 0
2. **while exists an augmenting path p in the residual network G_f**
3. augment flow f along p
4. **return** f



The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

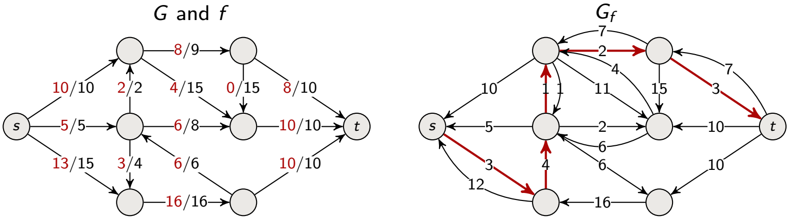
1. Initialize flow f to 0
2. **while exists an augmenting path p in the residual network G_f**
3. augment flow f along p
4. **return** f



The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

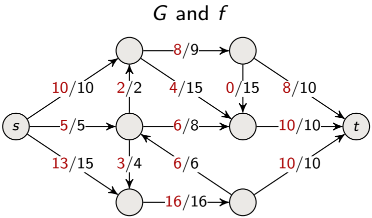
1. Initialize flow f to 0
2. **while** exists an augmenting path p in the residual network G_f
3. **augment flow f along p**
4. **return** f

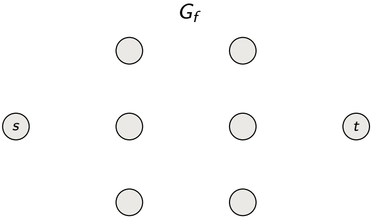


The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

1. Initialize flow f to 0
2. **while exists an augmenting path p in the residual network G_f**
3. augment flow f along p
4. **return** f

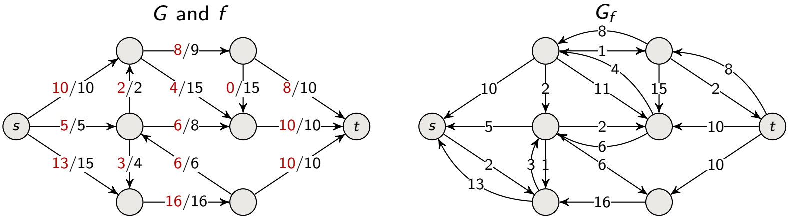




The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

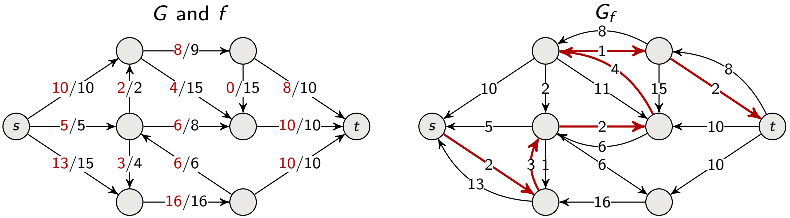
1. Initialize flow f to 0
2. **while exists an augmenting path p in the residual network G_f**
3. augment flow f along p
4. **return** f



The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

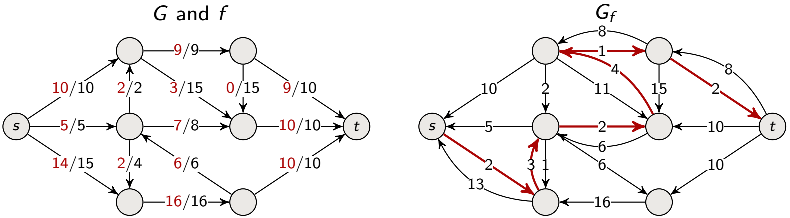
1. Initialize flow f to 0
2. **while exists an augmenting path p in the residual network G_f**
3. augment flow f along p
4. **return** f



The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

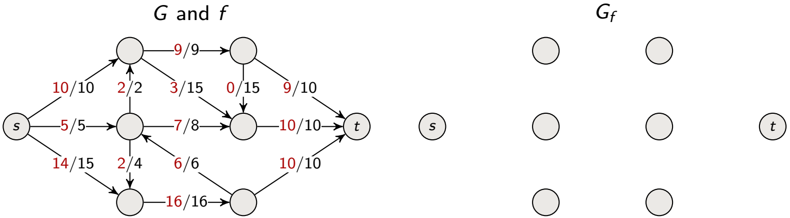
1. Initialize flow f to 0
2. **while** exists an augmenting path p in the residual network G_f
3. **augment flow f along p**
4. **return** f



The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

1. Initialize flow f to 0
2. **while** exists an augmenting path p in the residual network G_f
3. augment flow f along p
4. **return** f

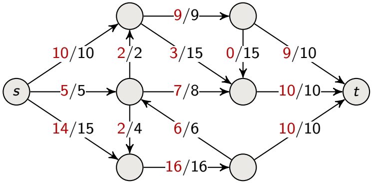


The Ford-Fulkerson Method'54

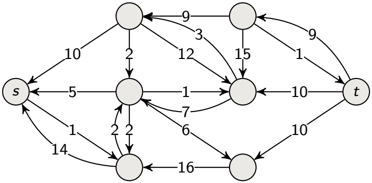
FORD-FULKERSON-METHOD(G, s, t):

1. Initialize flow f to 0
2. **while** exists an augmenting path p in the residual network G_f
3. augment flow f along p
4. **return** f

G and f



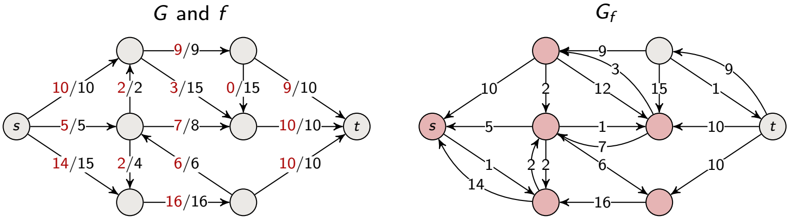
G_f



The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

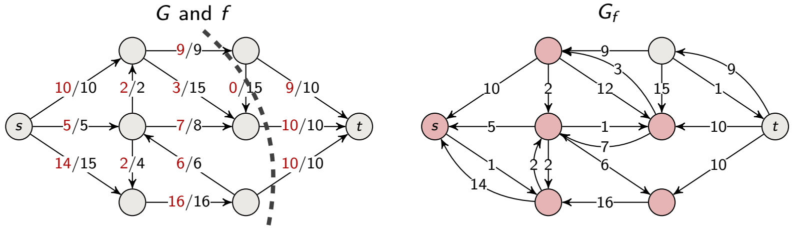
1. Initialize flow f to 0
2. **while** exists an augmenting path p in the residual network G_f
3. augment flow f along p
4. **return** f



The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

1. Initialize flow f to 0
2. **while** exists an augmenting path p in the residual network G_f
3. augment flow f along p
4. **return** f



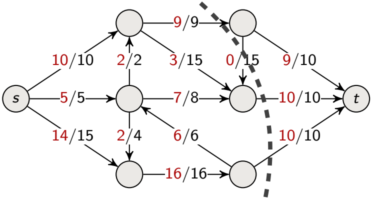
The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

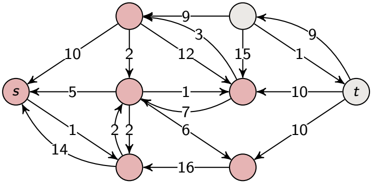
1. Initialize flow f to 0
2. **while** exists an augmenting path p in the residual network G_f
3. augment flow f along p
4. **return** f

No augmenting path, flow of value 29 and cut of capacity 29

G and f



G_f



Running Time

Might not terminate. However, if we either take the shortest path or the fattest path then this will not happen if the capacities are integers without proof

Running Time

Might not terminate. However, if we either take the shortest path or the fattest path then this will not happen if the capacities are integers without proof

Augmenting path	Number of iterations
BFS shortest path	$\leq \frac{1}{2} E \cdot V$
Fattest path	$\leq E \cdot \log(E \cdot U)$

Running Time

Might not terminate. However, if we either take the shortest path or the fattest path then this will not happen if the capacities are integers without proof

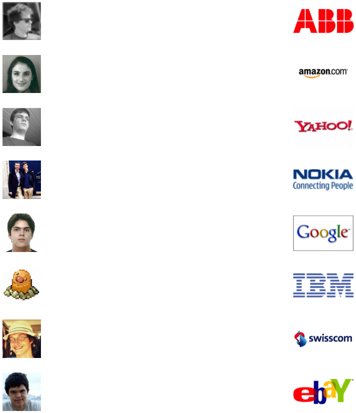
Augmenting path	Number of iterations
BFS shortest path	$\leq \frac{1}{2} E \cdot V$
Fattest path	$\leq E \cdot \log(E \cdot U)$

- U is the maximum flow value
- Fattest path: choose augmenting path with largest minimum capacity (bottleneck)

APPLICATIONS OF MAX-FLOW

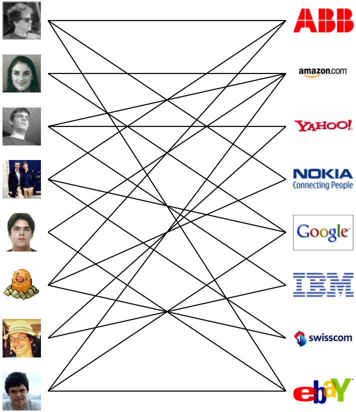
Bipartite matching

- N students apply for M jobs



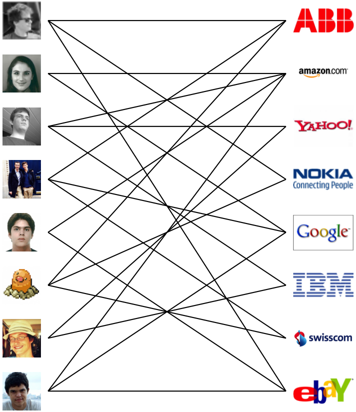
Bipartite matching

- N students apply for M jobs
- Each get several offers

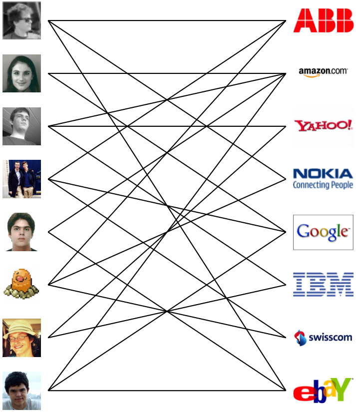


Bipartite matching

- N students apply for M jobs
- Each get several offers
- Is there a way to match all students to jobs? obviously M has to be at least equal to N

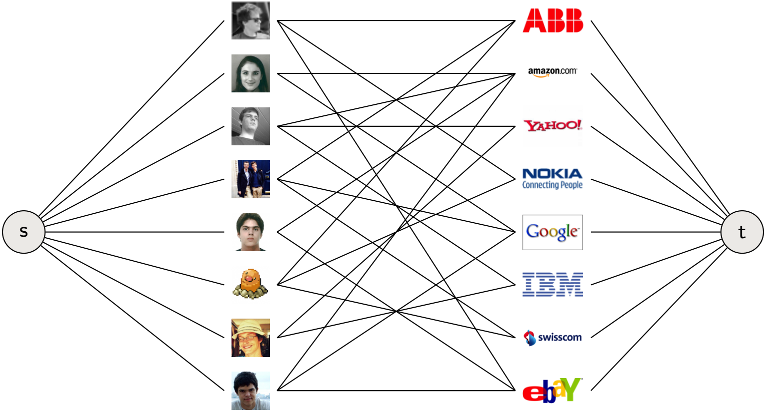


Bipartite matching as flow problem



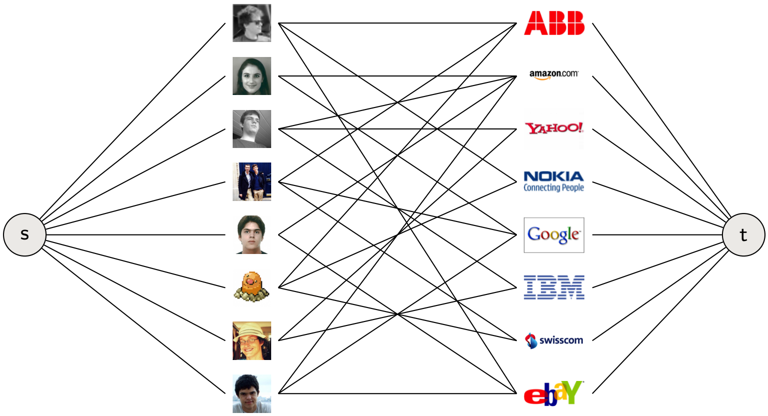
Bipartite matching as flow problem

- Add source s and sink t with edges from s to students and from jobs to t



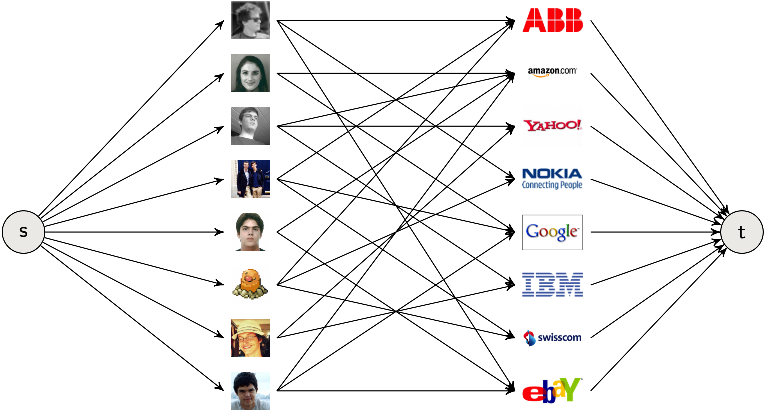
Bipartite matching as flow problem

- Add source s and sink t with edges from s to students and from jobs to t
- All edges have capacity one



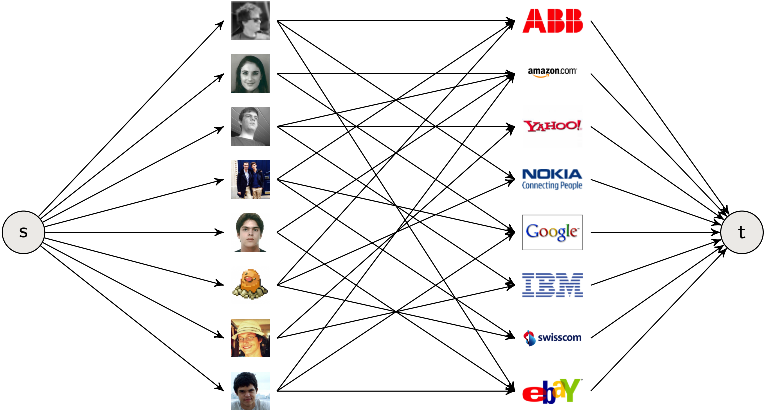
Bipartite matching as flow problem

- Add source s and sink t with edges from s to students and from jobs to t
- All edges have capacity one
- Direction is from left to right



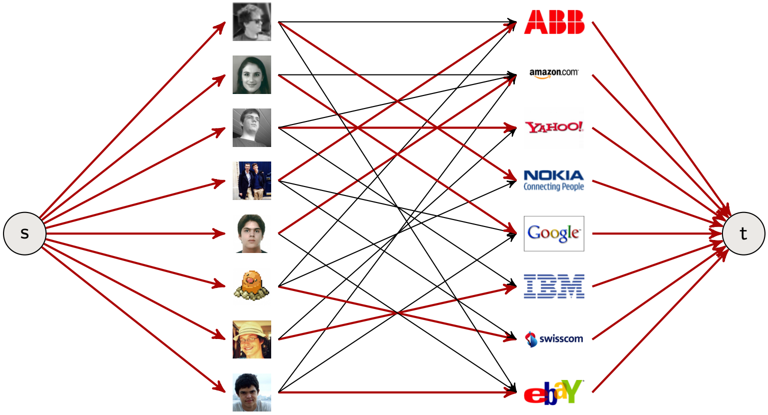
Bipartite matching as flow problem

- Run the Ford-Fulkerson method



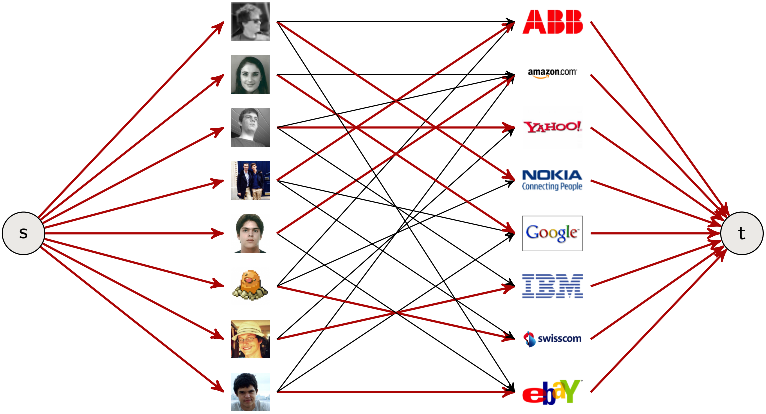
Bipartite matching as flow problem

- Run the Ford-Fulkerson method



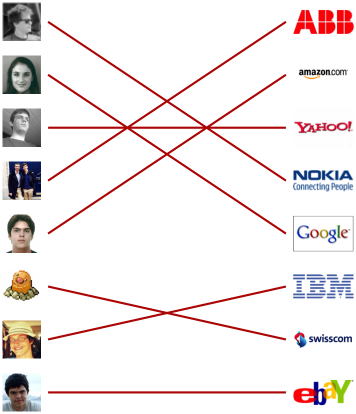
Bipartite matching as flow problem

- Run the Ford-Fulkerson method
- Matching is complete



Bipartite matching as flow problem

- Run the Ford-Fulkerson method
- Matching is complete



Why does it work?

Every matching defines a flow of value equal to the number of edges in matching

- Put flow 1 on
  - Edges of the matching
  - Edges from s to matched student nodes
  - Edges from matched job nodes to t
- Put flow 0 on all other edges

Works because flow conservation is equivalent to: no student is matched more than once, no job is matched more than once

Why does it work?

Every flow during the algorithm defines a matching of size equal to its value

- Flows obtained by Ford-Fulkerson are integer valued if capacities are integral, so value on every edge is 0 or 1
- Edges between students and jobs with flow 1 are a matching by flow conservation
  - There cannot be more than one edge with flow 1 from a student node
  - There cannot be more than one edge with flow 1 into a job node

Why does it work?

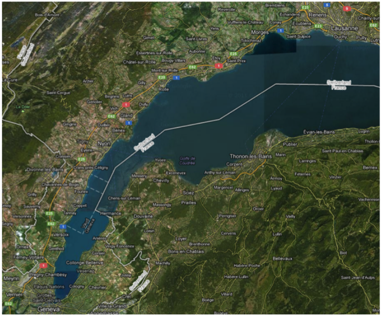
Every flow during the algorithm defines a matching of size equal to its value

- Flows obtained by Ford-Fulkerson are integer valued if capacities are integral, so value on every edge is 0 or 1
- Edges between students and jobs with flow 1 are a matching by flow conservation
  - There cannot be more than one edge with flow 1 from a student node
  - There cannot be more than one edge with flow 1 into a job node

So, maximum flow is a maximum matching!

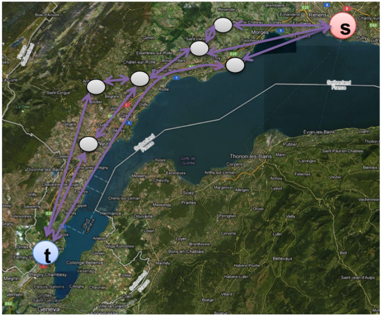
Edge-disjoint paths

- You want to travel to a nice location these winter holidays
- You need to drive from Lausanne to Geneva airport
- Winter season $\Rightarrow$ risk that roads are closed
- How many different routes can you take that does not share a common road?



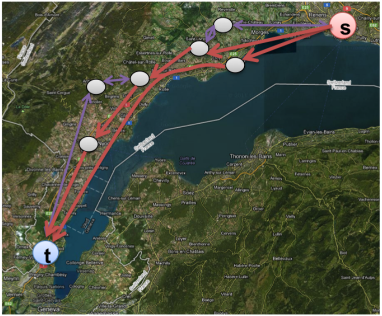
Edge-disjoint paths as flow network

- s = Lausanne
- t = Geneva airport
- An edge capacity of 1 in both directions for each road
- (make anti-parallel using gadgets)



Solution

- max-flow = # edge-disjoint paths
- min-cut = min #roads to be closed so that there is no route from Lausanne to Geneva airport



DATA STRUCTURES FOR DISJOINT SETS

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 we ask for the representative twice without modifying the set, we get the same answer both times

Operations

Operations

MAKE-SET(x): make a new set $S_i = \{x\}$, and add S_i to $\mathcal{S}$

Operations

$\textsc{Make-Set}(x)$: make a new set $S_i = \{x\}$, and add S_i to $\mathcal{S}$

$\textsc{Union}(x, y)$: 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 in $S_x \cup S_y$, often the representative of one of S_x and S_y
- Destroys S_x and S_y (since sets must be disjoint)

Operations

$\textsc{Make-Set}(x)$: make a new set $S_i = \{x\}$, and add S_i to $\mathcal{S}$

$\textsc{Union}(x, y)$: if $x \in S_x, y \in S_y$, then $\mathcal{S} = \mathcal{S} - S_x - S_y \cup \{S_x \cup S_y\}$

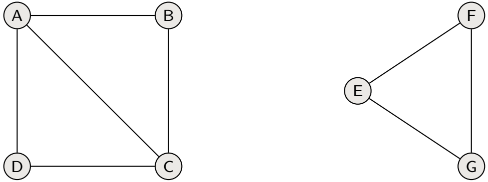
- Representative of new set is any member in $S_x \cup S_y$, often the representative of one of S_x and S_y
- Destroys S_x and S_y (since sets must be disjoint)

$\textsc{Find}(x)$: return representative of set containing x

Example application: connected components

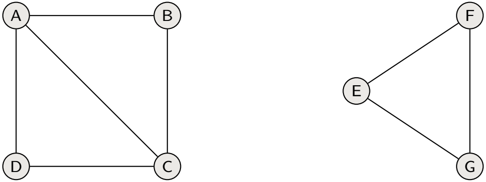
For a graph $G = (V, E)$, vertices u, v are in same connected component if and only if there is a path between them.

- Connected components partition vertices into equivalence classes



Connected components

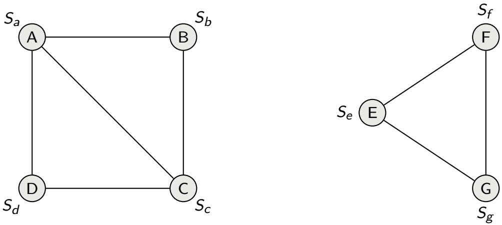
CONNECTED-COMPONENTS(G)
for each vertex $v \in G.V$
    MAKE-SET(v)
for each edge $(u, v) \in G.E$
    if FIND-SET(u) $\neq$ FIND-SET(v)
        UNION(u, v)



Connected components

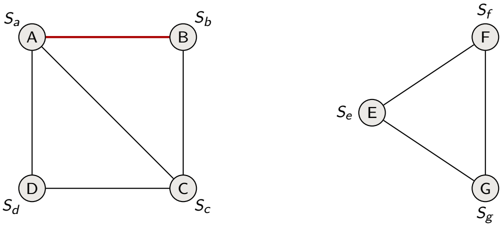
CONNECTED-COMPONENTS(G)

for each vertex $v \in G.V$
  MAKE-SET(v)
for each edge $(u, v) \in G.E$
  if FIND-SET$(u) \neq$ FIND-SET(v)
    UNION(u, v)



Connected components

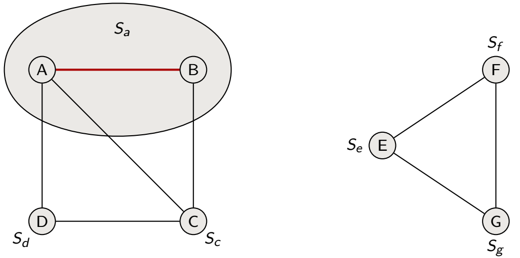
CONNECTED-COMPONENTS(G)
  for each vertex $v \in G.V$
    MAKE-SET(v)
  for each edge $(u, v) \in G.E$
    if FIND-SET(u) $\neq$ FIND-SET(v)
      UNION(u, v)



Connected components

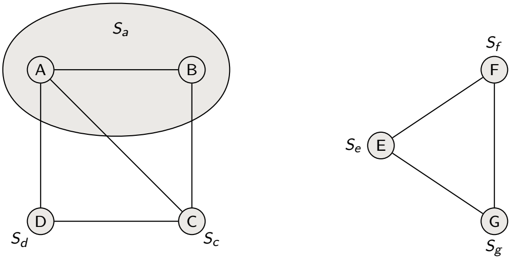
CONNECTED-COMPONENTS(G)

for each vertex $v \in G.V$
  MAKE-SET(v)
for each edge $(u, v) \in G.E$
  if FIND-SET$(u) \neq$ FIND-SET(v)
    UNION(u, v)



Connected components

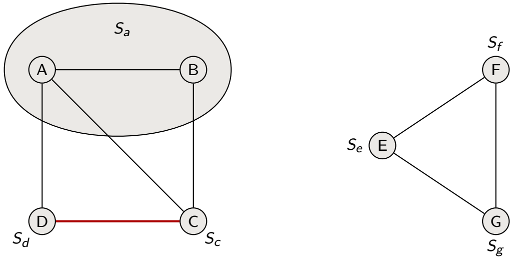
CONNECTED-COMPONENTS(G)
  for each vertex $v \in G.V$
    MAKE-SET(v)
  for each edge $(u, v) \in G.E$
    if FIND-SET$(u) \neq$ FIND-SET(v)
      UNION(u, v)



Connected components

CONNECTED-COMPONENTS(G)

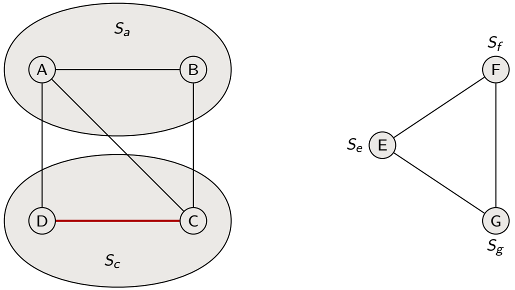
for each vertex $v \in G.V$
  MAKE-SET(v)
for each edge $(u, v) \in G.E$
  if FIND-SET$(u) \neq$ FIND-SET(v)
    UNION(u, v)



Connected components

CONNECTED-COMPONENTS(G)

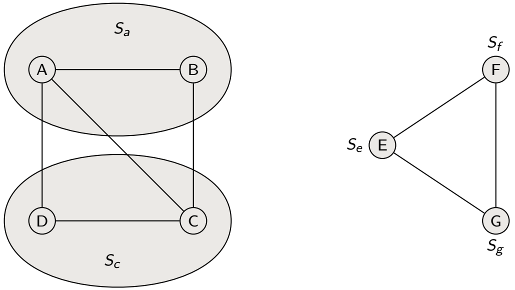
for each vertex $v \in G.V$
  MAKE-SET(v)
for each edge $(u, v) \in G.E$
  if FIND-SET$(u) \neq$ FIND-SET(v)
    UNION(u, v)



Connected components

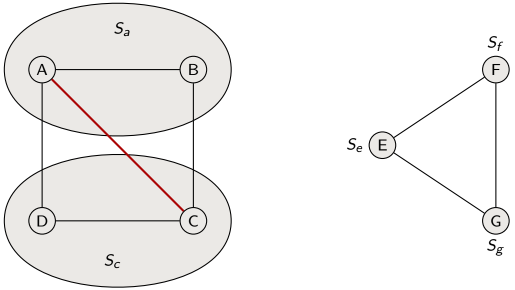
CONNECTED-COMPONENTS(G)

for each vertex $v \in G.V$
  MAKE-SET(v)
for each edge $(u, v) \in G.E$
  if FIND-SET$(u) \neq$ FIND-SET(v)
    UNION(u, v)



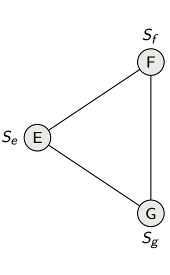
Connected components

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



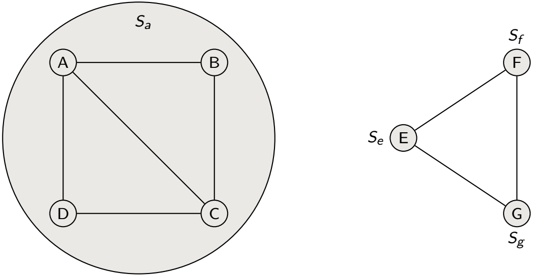
Connected components

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



Connected components

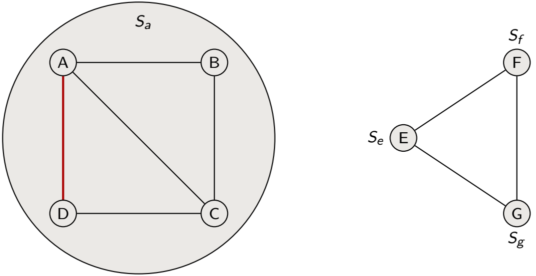
CONNECTED-COMPONENTS(G)
 for each vertex $v \in G.V$
  MAKE-SET(v)
 for each edge $(u, v) \in G.E$
  if FIND-SET$(u) \neq$ FIND-SET(v)
   UNION(u, v)



Connected components

CONNECTED-COMPONENTS(G)

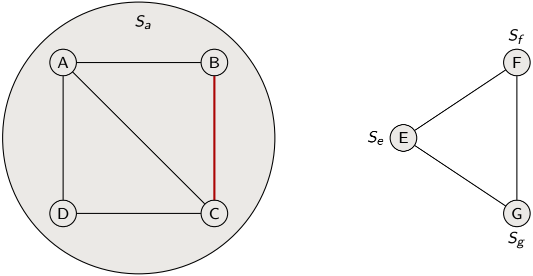
for each vertex $v \in G.V$
    MAKE-SET(v)
for each edge $(u, v) \in G.E$
    if FIND-SET$(u) \neq$ FIND-SET(v)
        UNION(u, v)



Connected components

CONNECTED-COMPONENTS(G)

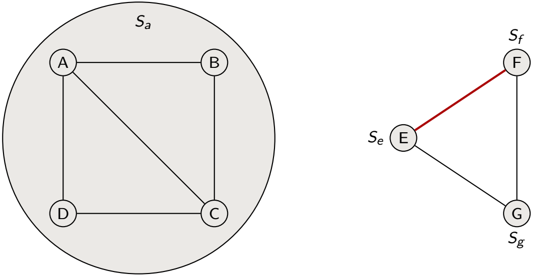
for each vertex $v \in G.V$
  MAKE-SET(v)
for each edge $(u, v) \in G.E$
  if FIND-SET$(u) \neq$ FIND-SET(v)
    UNION(u, v)



Connected components

CONNECTED-COMPONENTS(G)

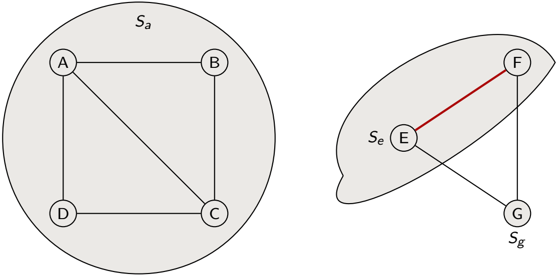
for each vertex $v \in G.V$
  MAKE-SET(v)
for each edge $(u, v) \in G.E$
  if FIND-SET$(u) \neq$ FIND-SET(v)
    UNION(u, v)



Connected components

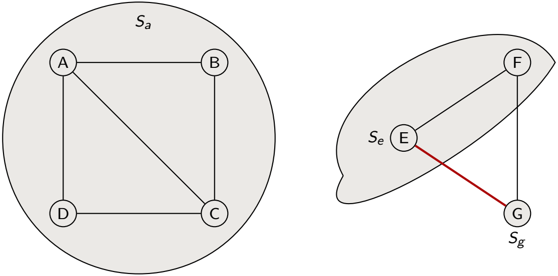
CONNECTED-COMPONENTS(G)

for each vertex $v \in G.V$
    MAKE-SET(v)
for each edge $(u, v) \in G.E$
    if FIND-SET$(u) \neq$ FIND-SET(v)
        UNION(u, v)



Connected components

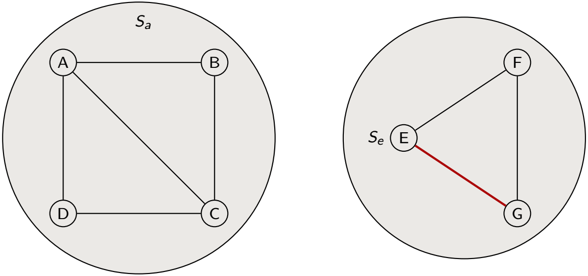
CONNECTED-COMPONENTS(G)
  for each vertex $v \in G.V$
    MAKE-SET(v)
  for each edge $(u, v) \in G.E$
    if FIND-SET(u) $\neq$ FIND-SET(v)
      UNION(u, v)



Connected components

CONNECTED-COMPONENTS(G)

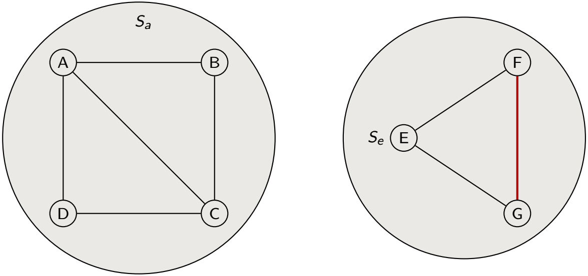
```
for each vertex v ∈ G.V
    MAKE-SET(v)
for each edge (u, v) ∈ G.E
    if FIND-SET(u) ≠ FIND-SET(v)
        UNION(u, v)
```



Connected components

CONNECTED-COMPONENTS(G)

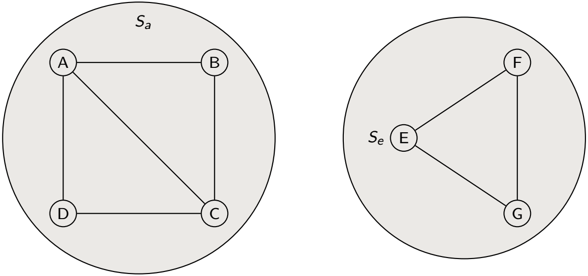
for each vertex $v \in G.V$
    MAKE-SET(v)
for each edge $(u, v) \in G.E$
    if FIND-SET$(u) \neq$ FIND-SET(v)
        UNION(u, v)



Connected components

CONNECTED-COMPONENTS(G)

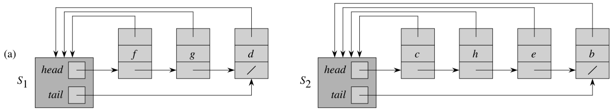
```
for each vertex v ∈ G.V
    MAKE-SET(v)
for each edge (u, v) ∈ G.E
    if FIND-SET(u) ≠ FIND-SET(v)
        UNION(u, v)
```



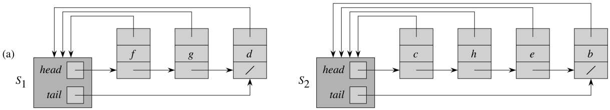
Linked list representation

List representation

- Each set is a single linked list represented by a set object that has
  - a pointer to the *head* of the list (assumed to be the representative
  - a pointer to the *tail* of the list
- Each object in the list has attributes for the *set member*, *pointer to the set object* and *next*

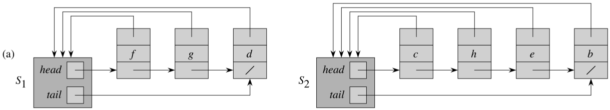


Make-Set and Find



Make-Set and Find

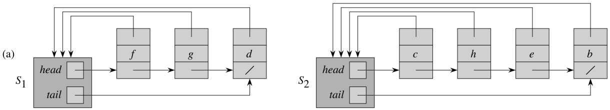
MAKE-SET(x): Create a single ton list in time $\Theta(1)$



Make-Set and Find

MAKE-SET(x): Create a single ton list in time $\Theta(1)$

FIND(x): follow the pointer back to the list object, and then follow the *head* pointer to the representative (time $\Theta(1)$)



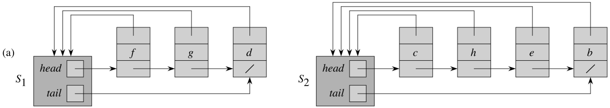
Union

A couple of ways of doing it

Union

A couple of ways of doing it

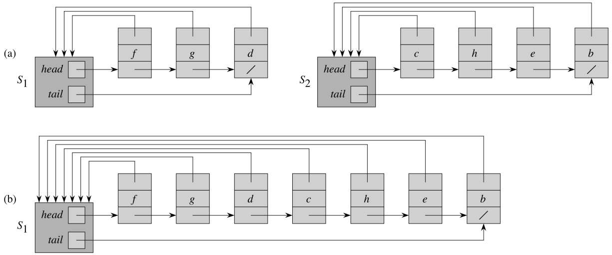
1. Append y's list onto the end of x's list. Use x's tail pointer to find the end.
   - Need to update the pointer back to the set object for every node on y's list.



Union

A couple of ways of doing it

1. Append y's list onto the end of x's list. Use x's tail pointer to find the end.
   - Need to update the pointer back to the set object for every node on y's list.



Union

A couple of ways of doing it

1. Append y's list onto the end of x's list. Use x's tail pointer to find the end.
   - Need to update the pointer back to the set object for every node on y's list.
   - If appending a large list onto a small list, it can take a while

Union

A couple of ways of doing it

1. Append y's list onto the end of x's list. Use x's tail pointer to find the end.
   - Need to update the pointer back to the set object for every node on y's list.
   - If appending a large list onto a small list, it can take a while

Operation	Number of objects updated
MAKE-SET(x_1)	1
MAKE-SET(x_2)	1
$\vdots$	$\vdots$
MAKE-SET(x_n)	1
UNION(x_2, x_1)	1
UNION(x_3, x_2)	2
UNION(x_4, x_3)	3
$\vdots$	$\vdots$
UNION(x_n, x_{n-1})	$n-1$

Union

A couple of ways of doing it

Union

A couple of ways of doing it

2. **Weighted-union heuristic** Always append the smaller list to the larger list (break ties arbitrarily)

Union

A couple of ways of doing it

2. **Weighted-union heuristic** Always append the smaller list to the larger list (break ties arbitrarily)

Theorem

With weighted-union heuristic, a sequence of m operations on n elements take $O(m + n \log n)$ time.

Weighted-union heuristic

Theorem

With weighted-union heuristic, a sequence of m operations on n elements take $O(m + n \lg n)$ time.

Proof sketch MAKE-SET and FIND still takes constant time.

Weighted-union heuristic

Theorem

With weighted-union heuristic, a sequence of m operations on n elements take $O(m + n \lg n)$ time.

Proof sketch MAKE-SET and FIND still takes constant time. How many times can each objects' representative pointer be updated? It must be in the smaller set each time

Weighted-union heuristic

Theorem

With weighted-union heuristic, a sequence of m operations on n elements take $O(m + n \lg n)$ time.

Proof sketch MAKE-SET and FIND still takes constant time. How many times can each objects' representative pointer be updated? It must be in the smaller set each time

times updated	size of resulting set
1	≥ 2

Weighted-union heuristic

Theorem

With weighted-union heuristic, a sequence of m operations on n elements take $O(m + n \lg n)$ time.

Proof sketch MAKE-SET and FIND still takes constant time. How many times can each objects' representative pointer be updated? It must be in the smaller set each time

times updated	size of resulting set
1	≥ 2
2	≥ 4

Weighted-union heuristic

Theorem

With weighted-union heuristic, a sequence of m operations on n elements take $O(m + n \lg n)$ time.

Proof sketch MAKE-SET and FIND still takes constant time. How many times can each objects' representative pointer be updated? It must be in the smaller set each time

times updated	size of resulting set
1	≥ 2
2	≥ 4
3	≥ 8

Weighted-union heuristic

Theorem

With weighted-union heuristic, a sequence of m operations on n elements take $O(m + n \lg n)$ time.

Proof sketch MAKE-SET and FIND still takes constant time. How many times can each objects' representative pointer be updated? It must be in the smaller set each time

times updated	size of resulting set
1	≥ 2
2	≥ 4
3	≥ 8
$\vdots$	$\vdots$

Weighted-union heuristic

Theorem

With weighted-union heuristic, a sequence of m operations on n elements take $O(m + n \lg n)$ time.

Proof sketch MAKE-SET and FIND still takes constant time. How many times can each objects' representative pointer be updated? It must be in the smaller set each time

times updated	size of resulting set
1	≥ 2
2	≥ 4
3	≥ 8
$\vdots$	$\vdots$
k	$\geq 2^k$

Weighted-union heuristic

Theorem

With weighted-union heuristic, a sequence of m operations on n elements take $O(m + n \lg n)$ time.

Proof sketch MAKE-SET and FIND still takes constant time. How many times can each objects' representative pointer be updated? It must be in the smaller set each time

times updated	size of resulting set
1	≥ 2
2	≥ 4
3	≥ 8
$\vdots$	$\vdots$
k	$\geq 2^k$
$\vdots$	$\vdots$

Weighted-union heuristic

Theorem

With weighted-union heuristic, a sequence of m operations on n elements take $O(m + n \lg n)$ time.

Proof sketch MAKE-SET and FIND still takes constant time. How many times can each objects' representative pointer be updated? It must be in the smaller set each time

times updated	size of resulting set
1	≥ 2
2	≥ 4
3	≥ 8
$\vdots$	$\vdots$
k	$\geq 2^k$
$\vdots$	$\vdots$
$\log n$	$\geq n$

Weighted-union heuristic

Theorem

With weighted-union heuristic, a sequence of m operations on n elements take $O(m + n \lg n)$ time.

Proof sketch MAKE-SET and FIND still takes constant time. How many times can each objects' representative pointer be updated? It must be in the smaller set each time

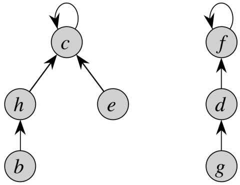
times updated	size of resulting set
1	≥ 2
2	≥ 4
3	≥ 8
$\vdots$	$\vdots$
k	$\geq 2^k$
$\vdots$	$\vdots$
$\log n$	$\geq n$

Therefore, each representative is updated $\leq \log n$ times

Disjoint-set forest

Forest of trees

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

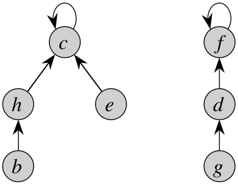


(a)

Forest of trees

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

MAKE-SET(x): Make a single-node tree



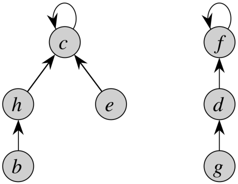
(a)

Forest of trees

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

MAKE-SET(x): Make a single-node tree

FIND(x): follow pointers to the root



(a)

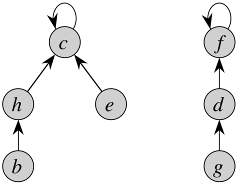
Forest of trees

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

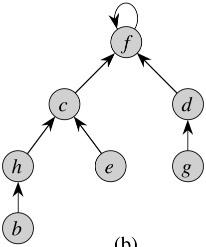
MAKE-SET(x): Make a single-node tree

FIND(x): follow pointers to the root

UNION(x, y): make one root a child of another



(a)



(b)

Great heuristics

Union by rank: make the root of the smaller tree a child of the root of the larger tree

Great heuristics

Union by rank: make the root of the smaller tree a child of the root of the larger tree

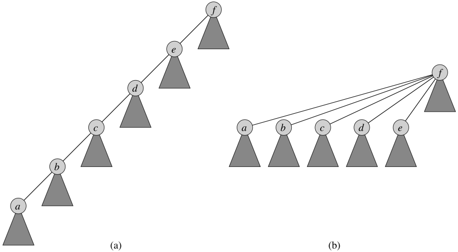
- Don't actually use size
- Use rank, which is an upper bound on height of node
- Make the root with the smaller rank a child of the root with the larger rank

Great heuristics

Union by rank: make the root of the smaller tree 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 a child of the root with the larger rank

Path compression: **Find path =** nodes visited during FIND on the trip to the root, make all nodes on the find path direct children to root.



Pseudocode of Make-Set and Find-Set

Make-Set(x)

1. $x.p = x$
2. $x.rank = 0$

Pseudocode of Make-Set and Find-Set

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

```
FIND-SET(x)
1. if x ≠ x.p
2.         x.p = FIND-SET(x.p)
3. return x.p
```

Pseudocode of Union

Union(x, y)

1. Link(Find-Set(x), Find-Set(y))

Pseudocode of Union

```
UNION(x, y)

LINK(FIND-SET(x), FIND-SET(y))
```

```
LINK(x, y)

if x.rank > y.rank
    y.p = x
else x.p = y
    if x.rank == y.rank
        y.rank = y.rank + 1
```

Running time

If use both union by rank and path compression,

$$O(m \cdot \alpha(n))$$

where $\alpha(n)$ is an extremely slowly growing function:

Running time

If use both union by rank and path compression,

$$O(m \cdot \alpha(n))$$

where $\alpha(n)$ is an extremely slowly growing function:

n	$\alpha(n)$

Running time

If use both union by rank and path compression,

$$O(m \cdot \alpha(n))$$

where $\alpha(n)$ is an extremely slowly growing function:

n	$\alpha(n)$
$0-2$	0

Running time

If use both union by rank and path compression,

$$O(m \cdot \alpha(n))$$

where $\alpha(n)$ is an extremely slowly growing function:

n	$\alpha(n)$
$0-2$	0
3	1

Running time

If use both union by rank and path compression,

$$O(m \cdot \alpha(n))$$

where $\alpha(n)$ is an extremely slowly growing function:

n	$\alpha(n)$
$0 - 2$	0
3	1
$4 - 7$	2

Running time

If use both union by rank and path compression,

$$O(m \cdot \alpha(n))$$

where $\alpha(n)$ is an extremely slowly growing function:

n	$\alpha(n)$
$0 - 2$	0
3	1
$4 - 7$	2
$8 - 2047$	3

Running time

If use both union by rank and path compression,

$$O(m \cdot \alpha(n))$$

where $\alpha(n)$ is an extremely slowly growing function:

n	$\alpha(n)$
$0-2$	0
3	1
$4-7$	2
$8-2047$	3
$2047- \gg 10^8$	4

Running time

If use both union by rank and path compression,

$$O(m \cdot \alpha(n))$$

where $\alpha(n)$ is an extremely slowly growing function:

n	$\alpha(n)$
$0 - 2$	0
3	1
$4 - 7$	2
$8 - 2047$	3
$2047 - \gg 10^8$	4

- $\alpha(n) \leq 5$ for any practical purpose
- The bound $O(m \cdot \alpha(n))$ is tight

Running time of connected components

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

Running time of connected components

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

- V elements
- $\leq V + 3E$ operations on Union-Find data structure

Running time of connected components

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

- V elements
- $\leq V + 3E$ operations on Union-Find data structure
- Total running time if implemented as linked list with weighted-union heuristic

Running time of connected components

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

- V elements
- $\leq V + 3E$ operations on Union-Find data structure
- Total running time if implemented as linked list with weighted-union heuristic

$$O(V \log V + E)$$

Running time of connected components

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

- V elements
- $\leq V + 3E$ operations on Union-Find data structure
- Total running time if implemented as linked list with weighted-union heuristic

$$O(V \log V + E)$$

- Total running time if implemented as forest with union-by-rank and path-compression

Running time of connected components

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

- V elements
- $\leq V + 3E$ operations on Union-Find data structure
- Total running time if implemented as linked list with weighted-union heuristic

$$O(V \log V + E)$$

- Total running time if implemented as forest with union-by-rank and path-compression

$$O((V + E)\alpha(V)) \approx O(V + E)$$

MINIMUM SPANNING TREES

Origin of today's lecture

Otakar Boruvka (1926)

- Electrical power company in western Moravia in Brno
- Most economical construction of electrical power network
- Concrete engineering problem led to what is now a cornerstone problem-solving model in combinatorial optimization

A spanning tree of a graph

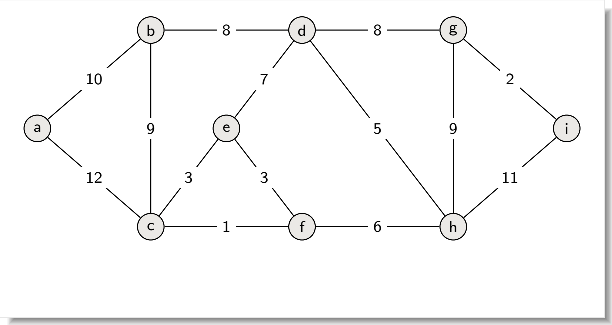
A set **T** of edges that is

- Acyclic
- Spanning (connects all vertices)

A spanning tree of a graph

A set **T** of edges that is

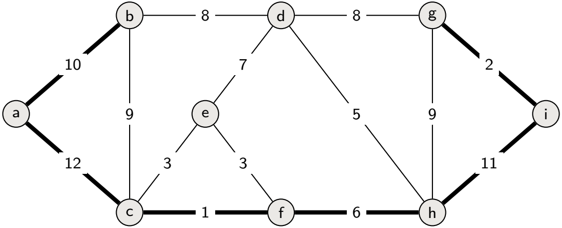
- Acyclic
- Spanning (connects all vertices)



A spanning tree of a graph

A set **T** of edges that is

- Acyclic
- Spanning (connects all vertices)

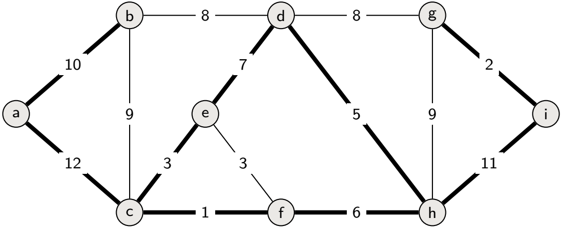


Acyclic, but not connected

A spanning tree of a graph

A set **T** of edges that is

- Acyclic
- Spanning (connects all vertices)

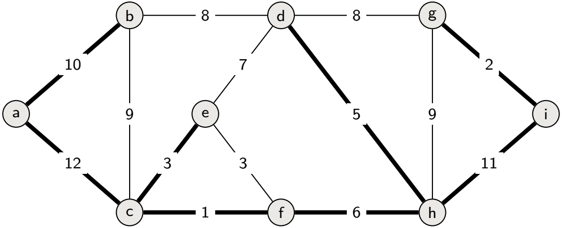


Connected, but not acyclic

A spanning tree of a graph

A set **T** of edges that is

- Acyclic
- Spanning (connects all vertices)

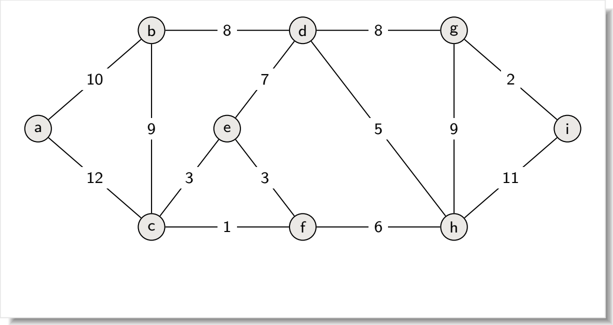


Acyclic and connected = spanning tree

Minimum spanning tree (MST)

INPUT: an undirected graph $G = (V, E)$ with weight $w(u, v)$ for each edge $(u, v) \in E$

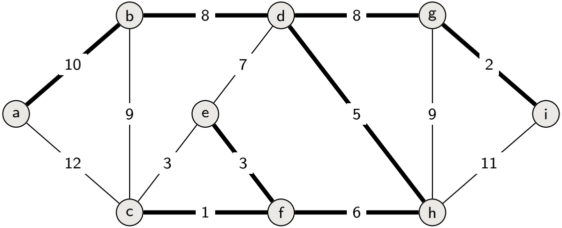
OUTPUT: a spanning tree of minimum total weight



Minimum spanning tree (MST)

INPUT: an undirected graph $G = (V, E)$ with weight $w(u, v)$ for each edge $(u, v) \in E$

OUTPUT: a spanning tree of minimum total weight



Spanning tree of weight $10 + 8 + 1 + 3 + 8 + 5 + 6 + 2 = 43$

EXAMPLE APPLICATIONS

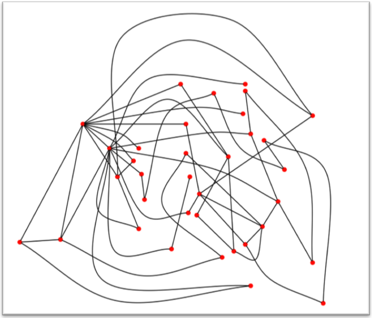
Example 1: Communication networks

A multinational company wants to lease communication lines between its various locations

Example 1: Communication networks

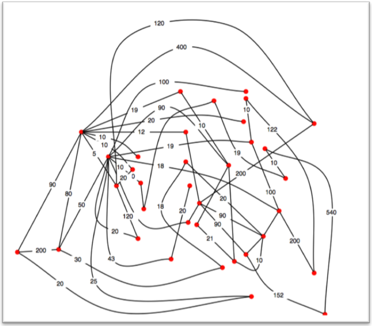
A multinational company wants to lease communication lines between its various locations

Example 1: Communication networks



A multinational company wants to lease communication lines between its various locations

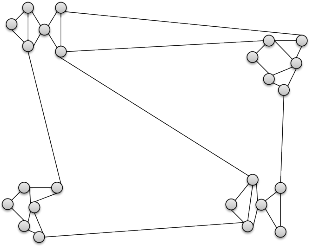
Example 1: Communication networks



A multinational company wants to lease communication lines between its various locations

Solution given by a MST on the graph

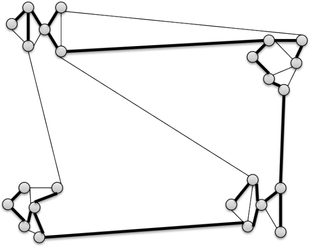
Example 2: Clustering



Edge weights equal to distance of nodes

Find: "cluster" of nodes   Possible solution: Find MST. Eliminate "fat" edges

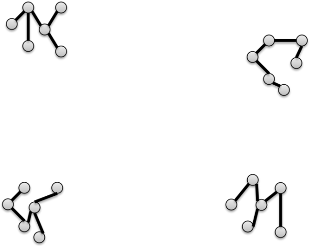
Example 2: Clustering



Edge weights equal to distance of nodes

Find: “cluster” of nodes   Possible solution: Find MST. Eliminate “fat” edges

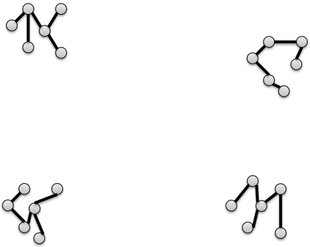
Example 2: Clustering



Edge weights equal to distance of nodes

Find: "cluster" of nodes   **Possible solution:** Find MST. Eliminate "fat" edges

Example 2: Clustering

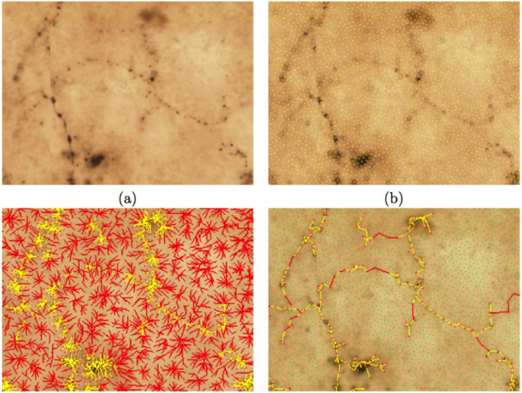


Edge weights equal to
distance of nodes

Find: "cluster" of nodes   Possible solution: Find MST. Eliminate "fat" edges

Note: this is a "heuristic" algorithm. Needs analysis

Example 3: Dendritic structures in the brain



(a) (b) (c) (d)

Example 4: Phylogenetic trees

Infer evolutionary relationships among various biological species

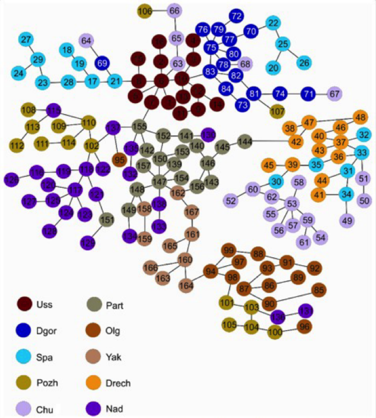
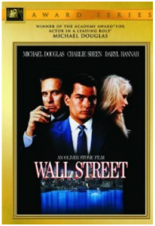


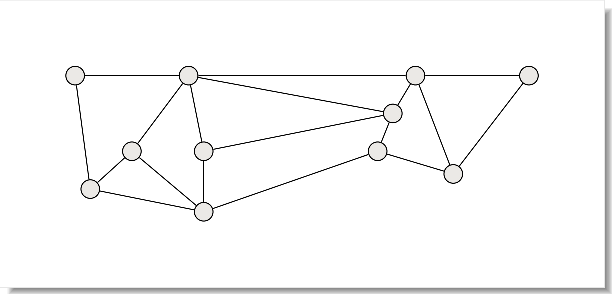
Figure 2 MS-tree representing phylogenetic relationships among representative *Panax ginseng* populations. Length of lines is proportional to the Euclidean distances among plants. Length of scale line is equal to 50 units of Euclidean distances



ALGORITHMS FOR MST

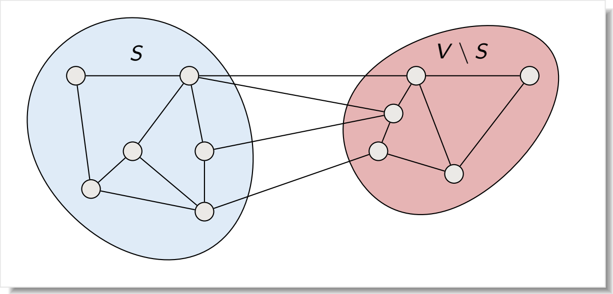
"Greed is good. Greed is right. Greed works. Greed clarifies, cuts through and captures the essence of the evolutionary spirit."
- Gordon Gecko

Cuts



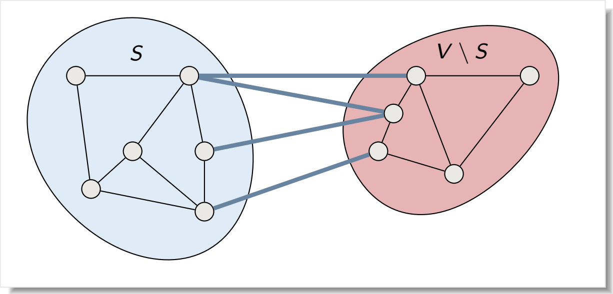
Cuts

- A **cut** $(S, V \setminus S)$ is a partition of the vertices into two nonempty disjoint sets S and $V \setminus S$



Cuts

- A **cut** $(S, V \setminus S)$ is a partition of the vertices into two nonempty disjoint sets S and $V \setminus S$
- A **crossing edge** is an edge connecting vertex S to vertex in $V \setminus S$

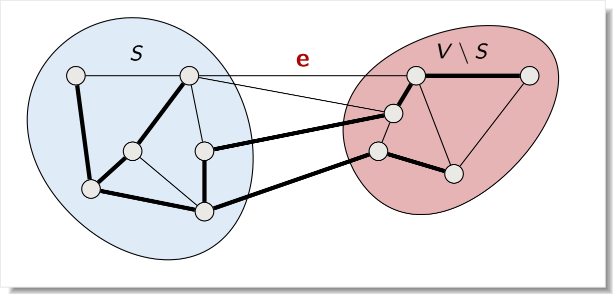


Cut property

Consider a cut $(S, V \setminus S)$ and let

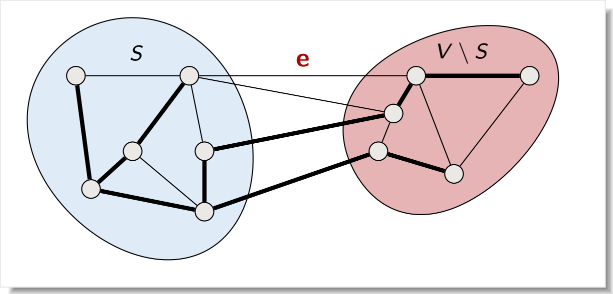
- T be a tree on S which is part of a MST
- e be a crossing edge of minimum weight

Then there is MST of G containing e and T



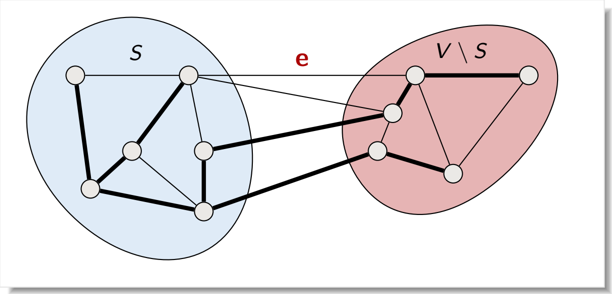
Cut property

Proof. If e is already in MST we are done.



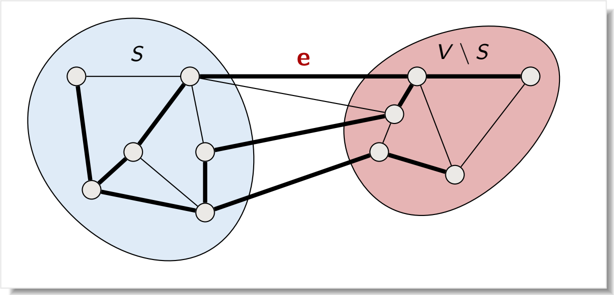
Cut property

Proof. If e is already in MST we are done.
Otherwise add e to the MST



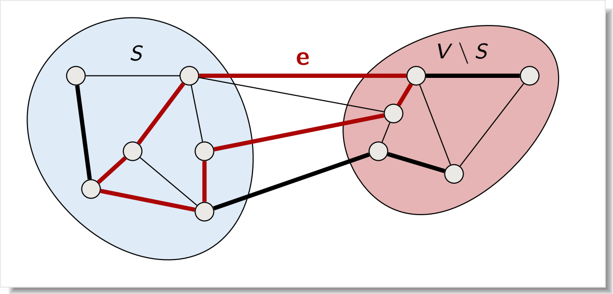
Cut property

Proof. If e is already in MST we are done.
Otherwise add e to the MST



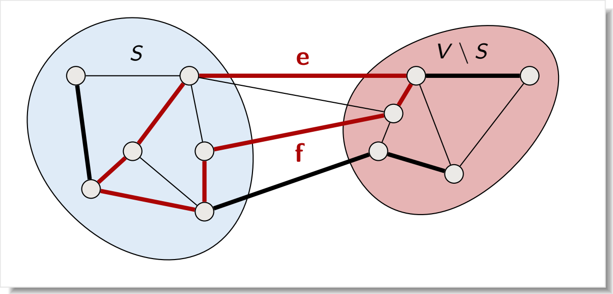
Cut property

Proof. If e is already in MST we are done.
Otherwise add e to the MST
This creates a cycle



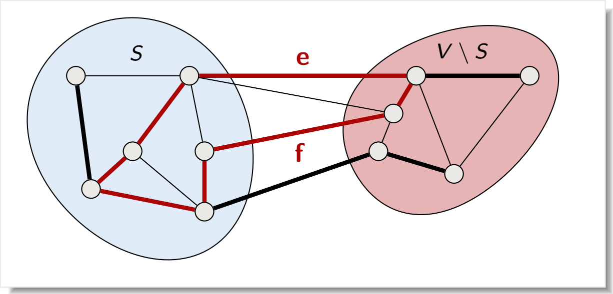
Cut property

Proof. If e is already in MST we are done.
Otherwise add e to the MST
This creates a cycle
At least one other crossing edge f in cycle



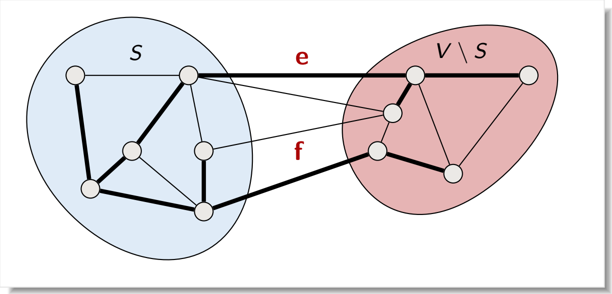
Cut property

Proof. If e is already in MST we are done.
Otherwise add e to the MST
This creates a cycle
At least one other crossing edge f in cycle
$w(f) \geq w(e)$ (actually must be equal)



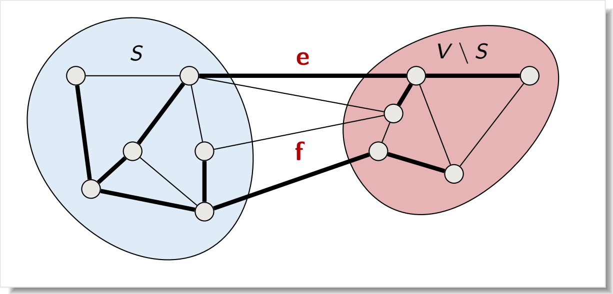
Cut property

Proof. If e is already in MST we are done.
Otherwise add e to the MST
This creates a cycle
At least one other crossing edge f in cycle
$w(f) \geq w(e)$ (actually must be equal)
Replace f by e in MST
This gives new MST which contains T and e

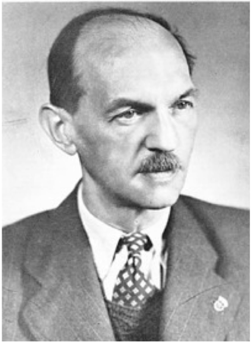


Cut property

Proof. If e is already in MST we are done.
Otherwise add e to the MST
This creates a cycle
At least one other crossing edge f in cycle
$w(f) \geq w(e)$ (actually must be equal)
Replace f by e in MST
This gives new MST which contains T and e



Prim's algorithm



Vojitech Jarnik
1897 - 1970

Robert Prim
1921-

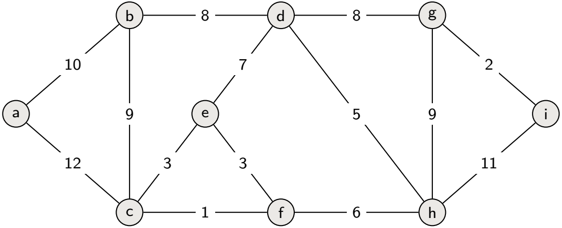
Edsger Dijkstra
1930 - 2002

Prim's algorithm

Start with any vertex v, set tree T to singleton v

Greedily grow tree T:

at each step add to T a minimum weight crossing edge with respect to the cut induced by T

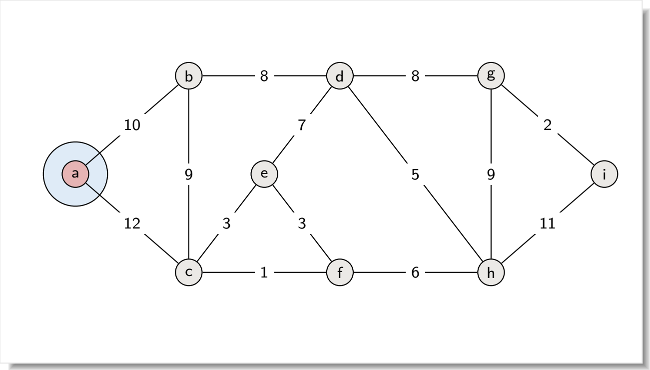


Prim's algorithm

Start with any vertex v, set tree T to singleton v

Greedily grow tree T:

at each step add to T a minimum weight crossing edge with respect to the cut induced by T

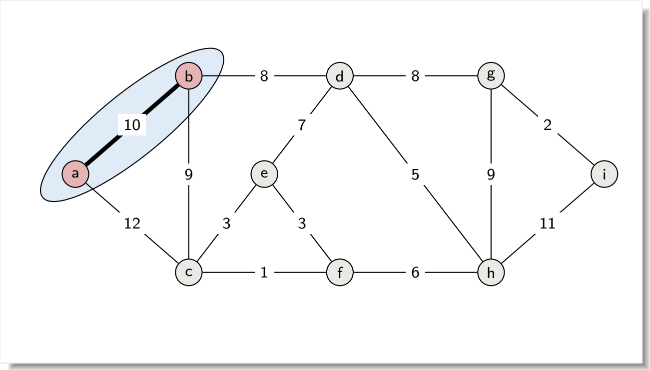


Prim's algorithm

Start with any vertex v, set tree T to singleton v

Greedily grow tree T:

at each step add to T a minimum weight crossing edge with respect to the cut induced by T

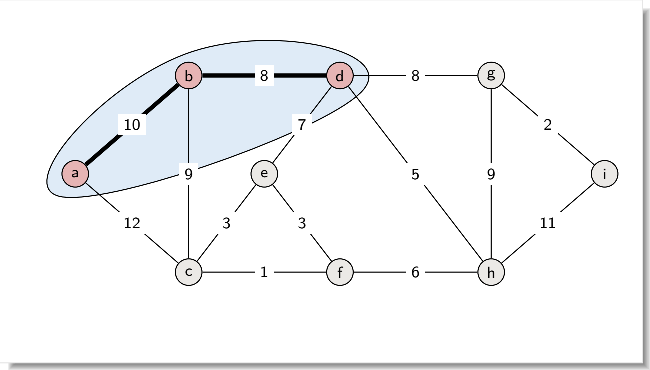


Prim's algorithm

Start with any vertex v, set tree T to singleton v

Greedily grow tree T:

at each step add to T a minimum weight crossing edge with respect to the cut induced by T

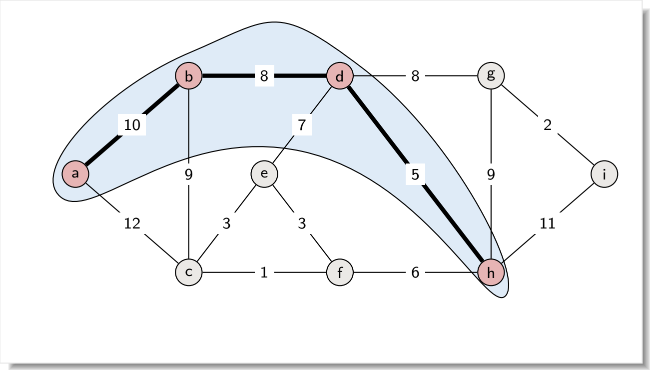


Prim's algorithm

Start with any vertex v, set tree T to singleton v

Greedily grow tree T:

at each step add to T a minimum weight crossing edge with respect to the cut induced by T

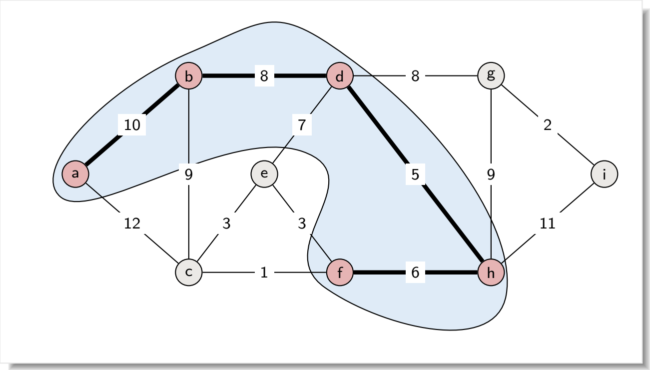


Prim's algorithm

Start with any vertex v, set tree T to singleton v

Greedily grow tree T:

at each step add to T a minimum weight crossing edge with respect to the cut induced by T

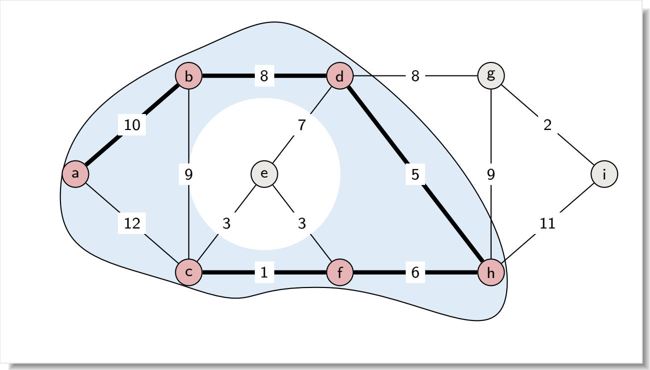


Prim's algorithm

Start with any vertex v, set tree T to singleton v

Greedily grow tree T:

at each step add to T a minimum weight crossing edge with respect to the cut induced by T

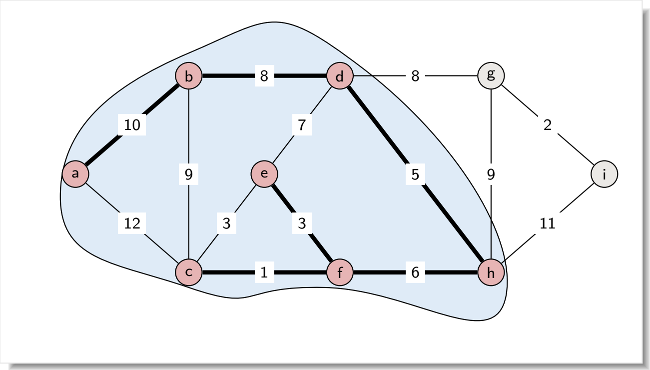


Prim's algorithm

Start with any vertex v, set tree T to singleton v

Greedily grow tree T:

at each step add to T a minimum weight crossing edge with respect to the cut induced by T

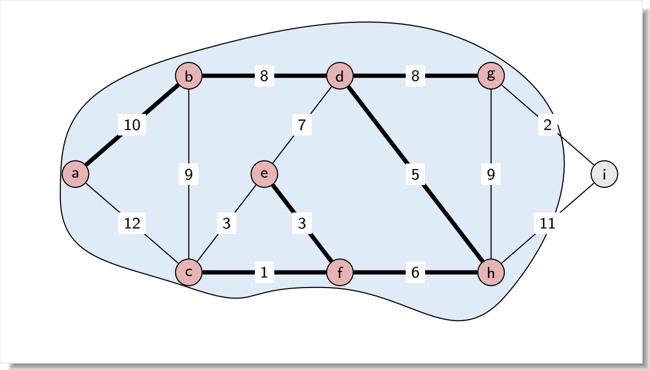


Prim's algorithm

Start with any vertex v, set tree T to singleton v

Greedily grow tree T:

at each step add to T a minimum weight crossing edge with respect to the cut induced by T

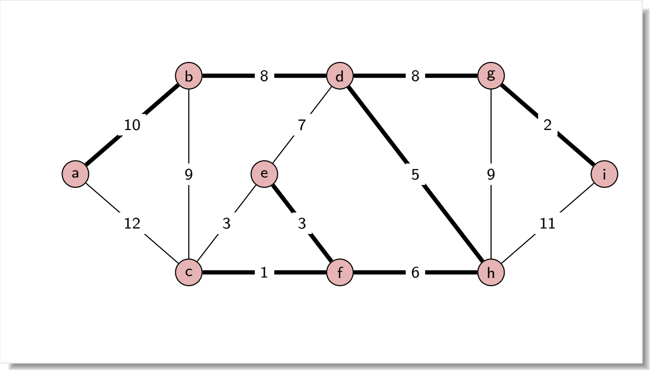


Prim's algorithm

Start with any vertex v, set tree T to singleton v

Greedily grow tree T:

at each step add to T a minimum weight crossing edge with respect to the cut induced by T

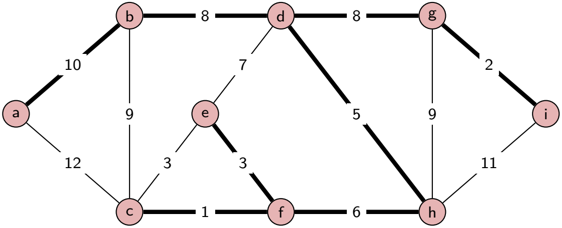


Prim's algorithm

Start with any vertex v, set tree T to singleton v

Greedily grow tree T:

at each step add to T a minimum weight crossing edge with respect to the cut induced by T



Minimum spanning tree of weight $10 + 8 + 5 + 6 + 3 + 1 + 8 + 2 = 43$

Why does it work?

T is always a subtree of a MST

Proof by induction on number of nodes in T. Final T is MST by this result

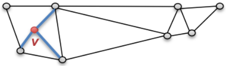
Why does it work?

T is always a subtree of a MST

Proof by induction on number of nodes in T. Final T is MST by this result

Base case: trivial

Singleton v is part of a MST



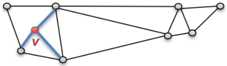
Why does it work?

T is always a subtree of a MST

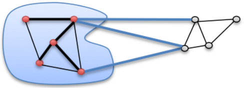
Proof by induction on number of nodes in T. Final T is MST by this result

Base case: trivial

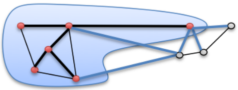
Singleton v is part of a MST



Inductive step: use cut property



In MST by hypothesis



In MST by cut property

Implementation challenge

How do we find minimum crossing edge at every iteration?

Implementation challenge

How do we find minimum crossing edge at every iteration?

Check all outgoing edges:

- O(E) comparisons at every iteration
- O(E V) running time in total

Implementation challenge

How do we find minimum crossing edge at every iteration?

Check all outgoing edges:

- O(E) comparisons at every iteration
- O(E V) running time in total

More clever solution:

- For every node w, keep value $dist(w)$ that measures the "distance" of w from current tree
- When a new node u is added to tree, check whether neighbors of u decreases their distance to tree; if so, decrease distance
- Maintain a min-priority queue for the nodes and their distances

Implementation and Analysis

```
PRIM(G, w, r)
  Q = ∅
  for each u ∈ G.V
      u.key = ∞
      u.π = NIL
      INSERT(Q, u)
  DECREASE-KEY(Q, r, 0)        // r.key = 0
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      for each v ∈ G.Adj[u]
          if v ∈ Q and w(u, v) < v.key
              v.π = u
              DECREASE-KEY(Q, v, w(u, v))
```

Implementation and Analysis

```
PRIM(G, w, r)
  Q = ∅
  for each u ∈ G.V
      u.key = ∞
      u.π = NIL
      INSERT(Q, u)
  DECREASE-KEY(Q, r, 0)      // r.key = 0
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      for each v ∈ G.Adj[u]
          if v ∈ Q and w(u, v) < v.key
              v.π = u
              DECREASE-KEY(Q, v, w(u, v))
```

- Initialize Q and first **for** loop: $O(V \lg V)$

Implementation and Analysis

```
PRIM(G, w, r)
  Q = ∅
  for each u ∈ G.V
      u.key = ∞
      u.π = NIL
      INSERT(Q, u)
  DECREASE-KEY(Q, r, 0)      // r.key = 0
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      for each v ∈ G.Adj[u]
          if v ∈ Q and w(u, v) < v.key
              v.π = u
              DECREASE-KEY(Q, v, w(u, v))
```

- Initialize Q and first **for** loop: $O(V \lg V)$
- Decrease key of r: $O(\lg V)$

Implementation and Analysis

```
PRIM(G, w, r)
  Q = ∅
  for each u ∈ G.V
      u.key = ∞
      u.π = NIL
      INSERT(Q, u)
  DECREASE-KEY(Q, r, 0)      // r.key = 0
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      for each v ∈ G.Adj[u]
          if v ∈ Q and w(u, v) < v.key
              v.π = u
              DECREASE-KEY(Q, v, w(u, v))
```

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

Implementation and Analysis

```
PRIM(G, w, r)
  Q = ∅
  for each u ∈ G.V
      u.key = ∞
      u.π = NIL
      INSERT(Q, u)
  DECREASE-KEY(Q, r, 0)        // r.key = 0
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      for each v ∈ G.Adj[u]
          if v ∈ Q and w(u, v) < v.key
              v.π = u
              DECREASE-KEY(Q, v, w(u, v))
```

- Initialize Q and first **for** loop: $O(V \lg V)$
- Decrease key of r: $O(\lg V)$
- **while** loop: V EXTRACT-MIN calls $\Rightarrow O(V \lg V)$
  $\leq E$ DECREASE-KEY calls $\Rightarrow O(E \lg V)$
- Total: $O(E \lg V)$ (can be made $O(E + V \lg V)$ with careful queue implementation)

Summary

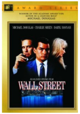
- Greedy is good (sometimes)
- Prim's algorithm

  Min-priority queue for implementation

- Next time Kruskal's algortihm

  Union-Find for implementation

- Many applications



Kruskal's algorithm

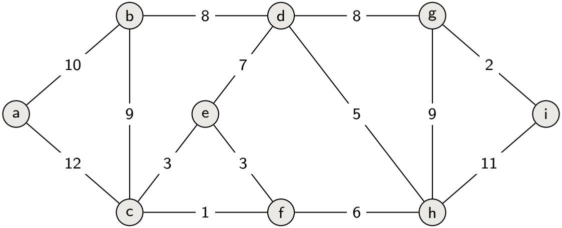
```
KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ G.V
      MAKE-SET(v)
  sort the edges of G.E into nondecreasing order by weight w
  for each (u, v) taken from the sorted list
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
  return A
```

Kruskal's algorithm

Start from empty forest T

Greedily maintain forest T which will become MST at the end:

at each step add cheapest edge that does not create a cycle

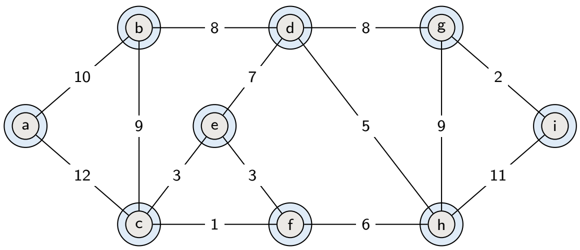


Kruskal's algorithm

Start from empty forest T

Greedily maintain forest T which will become MST at the end:

at each step add cheapest edge that does not create a cycle

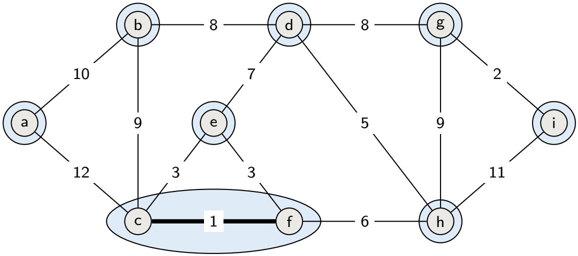


Kruskal's algorithm

Start from empty forest T

Greedily maintain forest T which will become MST at the end:

at each step add cheapest edge that does not create a cycle

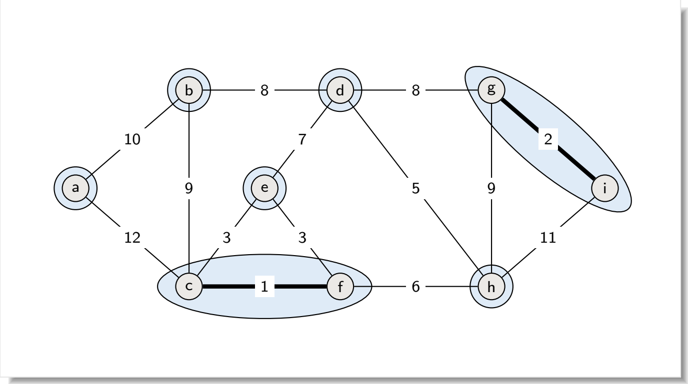


Kruskal's algorithm

Start from empty forest T

Greedily maintain forest T which will become MST at the end:

at each step add cheapest edge that does not create a cycle

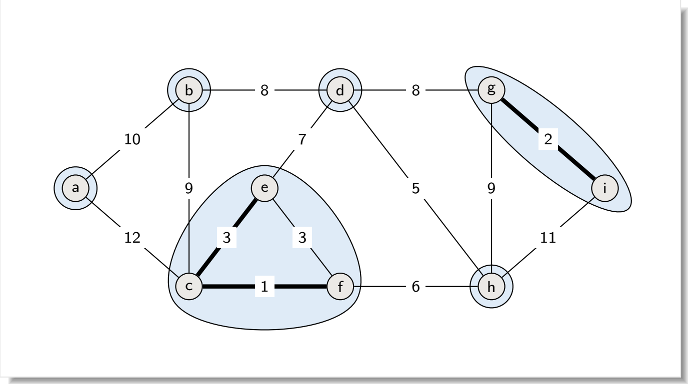


Kruskal's algorithm

Start from empty forest T

Greedily maintain forest T which will become MST at the end:

at each step add cheapest edge that does not create a cycle

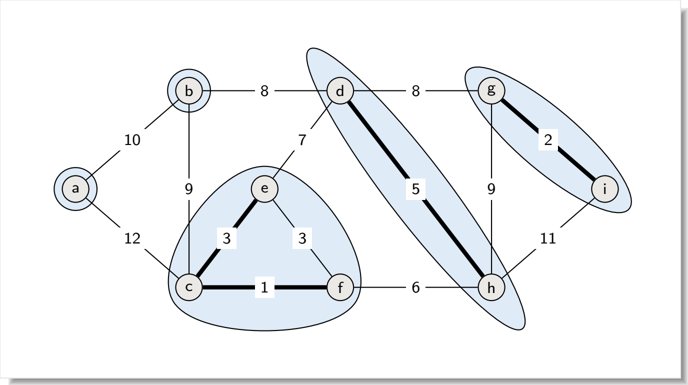


Kruskal's algorithm

Start from empty forest T

Greedily maintain forest T which will become MST at the end:

at each step add cheapest edge that does not create a cycle

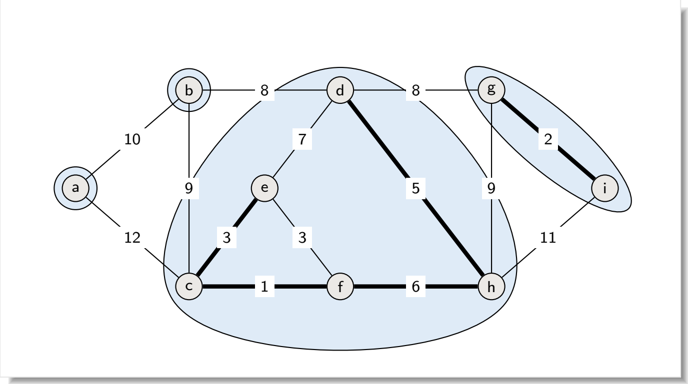


Kruskal's algorithm

Start from empty forest T

Greedily maintain forest T which will become MST at the end:

at each step add cheapest edge that does not create a cycle

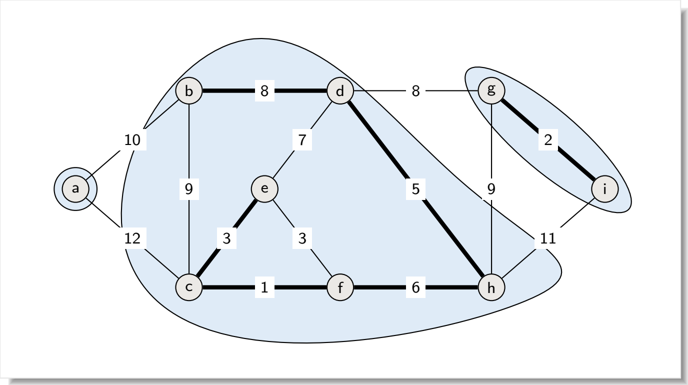


Kruskal's algorithm

Start from empty forest T

Greedily maintain forest T which will become MST at the end:

at each step add cheapest edge that does not create a cycle

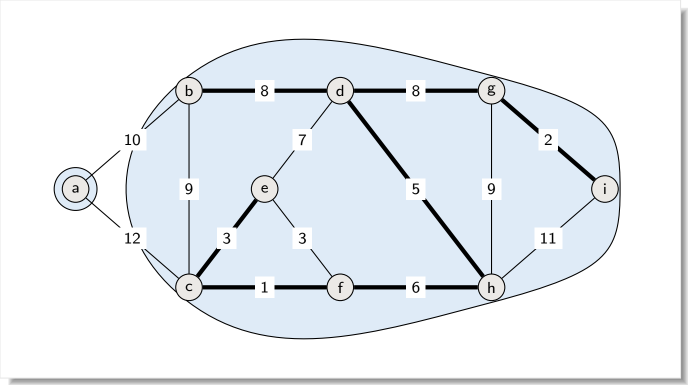


Kruskal's algorithm

Start from empty forest T

Greedily maintain forest T which will become MST at the end:

at each step add cheapest edge that does not create a cycle

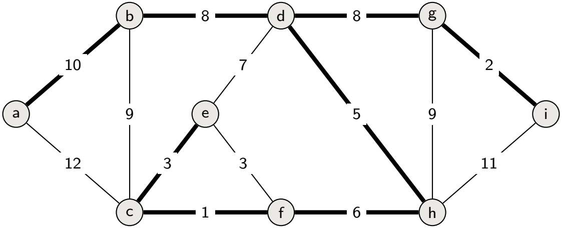


Kruskal's algorithm

Start from empty forest T

Greedily maintain forest T which will become MST at the end:

at each step add cheapest edge that does not create a cycle

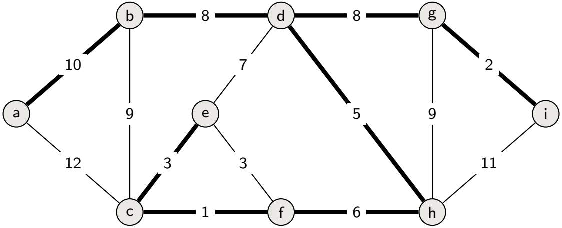


Kruskal's algorithm

Start from empty forest T

Greedily maintain forest T which will become MST at the end:

at each step add cheapest edge that does not create a cycle



Minimum spanning tree of weight $1 + 2 + 3 + 5 + 6 + 8 + 8 + 10 = 43$

Why does it work?

Claim: T is always a sub-forest of a MST

Proof by induction on the number of components/edges in T

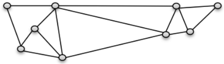
Why does it work?

Claim: T is always a sub-forest of a MST

Proof by induction on the number of components/edges in T

Base case: trivial

T is a union of singleton vertices



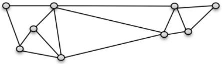
Why does it work?

Claim: T is always a sub-forest of a MST

Proof by induction on the number of components/edges in T

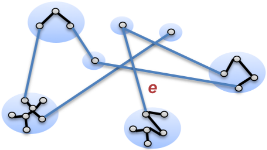
Base case: trivial

T is a union of singleton vertices



Inductive step:

1. By hypothesis, current ***T*** is a sub-forest of a MST,



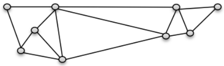
Why does it work?

Claim: T is always a sub-forest of a MST

Proof by induction on the number of components/edges in T

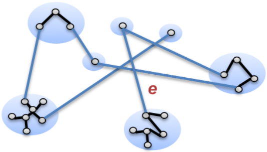
Base case: trivial

T is a union of singleton vertices



Inductive step:

1. By hypothesis, current ***T*** is a sub-forest of a MST,
2. Edge ***e*** is an edge of minimum weight that doesn't create a cycle



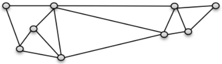
Why does it work?

Claim: T is always a sub-forest of a MST

Proof by induction on the number of components/edges in T

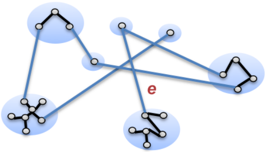
Base case: trivial

T is a union of singleton vertices



Inductive step:

1. By hypothesis, current ***T*** is a sub-forest of a MST,
2. Edge **e** is an edge of minimum weight that doesn't create a cycle
3. *Suppose* **e** creates a cycle with MST



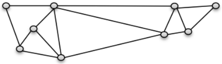
Why does it work?

Claim: T is always a sub-forest of a MST

Proof by induction on the number of components/edges in T

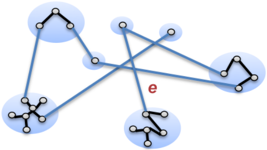
Base case: trivial

T is a union of singleton vertices



Inductive step:

1. By hypothesis, current ***T*** is a sub-forest of a MST,
2. Edge ***e*** is an edge of minimum weight that doesn't create a cycle
3. *Suppose* ***e*** creates a cycle with MST
4. Replace an edge (with larger weight) along this cycle by ***e***



An MST since weight did not increase!

Implementation challenge

In each iteration, we need to check whether cheapest edge creates a cycle

Implementation challenge

In each iteration, we need to check whether cheapest edge creates a cycle

This is the same thing as checking whether its endpoints belong to the same component $\Rightarrow$ **use disjoint sets (union-find) data structure**

Implementation challenge

In each iteration, we need to check whether cheapest edge creates a cycle

This is the same thing as checking whether its endpoints belong to the same component ⇒ **use disjoint sets (union-find) data structure**

Let the connected components denote sets

- Initially each singleton is a set
- When edge (u, v) is added to T, make union of the two connected components/sets

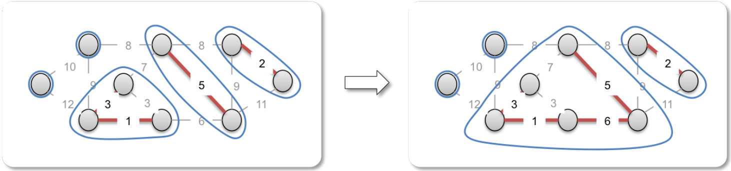
Implementation challenge

In each iteration, we need to check whether cheapest edge creates a cycle

This is the same thing as checking whether its endpoints belong to the same component ⇒ **use disjoint sets (union-find) data structure**

Let the connected components denote sets

- Initially each singleton is a set
- When edge (u, v) is added to T, make union of the two connected components/sets



Implementation and Analysis

```
KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ G.V
      MAKE-SET(v)
  sort the edges of G.E into nondecreasing order by weight w
  for each (u, v) taken from the sorted list
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
  return A
```

Implementation and Analysis

```
KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ G.V
      MAKE-SET(v)
  sort the edges of G.E into nondecreasing order by weight w
  for each (u, v) taken from the sorted list
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
  return A
```

- Initialize A:

Implementation and Analysis

```
KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ G.V
      MAKE-SET(v)
  sort the edges of G.E into nondecreasing order by weight w
  for each (u, v) taken from the sorted list
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
  return A
```

- Initialize A: $O(1)$

Implementation and Analysis

```
KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ G.V
      MAKE-SET(v)
  sort the edges of G.E into nondecreasing order by weight w
  for each (u, v) taken from the sorted list
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
  return A
```

- Initialize A: $O(1)$
- First **for** loop:

Implementation and Analysis

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

- Initialize A: $O(1)$
- First **for** loop: V MAKE-SETS

Implementation and Analysis

```
KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ G.V
      MAKE-SET(v)
  sort the edges of G.E into nondecreasing order by weight w
  for each (u, v) taken from the sorted list
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
  return A
```

- Initialize A: $O(1)$
- First **for** loop: V MAKE-SETS
- Sort E :

Implementation and Analysis

```
KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ G.V
      MAKE-SET(v)
  sort the edges of G.E into nondecreasing order by weight w
  for each (u, v) taken from the sorted list
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
  return A
```

- Initialize A: $O(1)$
- First **for** loop: V MAKE-SETS
- Sort E : $O(E \lg E)$

Implementation and Analysis

```
KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ G.V
      MAKE-SET(v)
  sort the edges of G.E into nondecreasing order by weight w
  for each (u, v) taken from the sorted list
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
  return A
```

- Initialize A: $O(1)$
- First **for** loop: V MAKE-SETS
- Sort E : $O(E \lg E)$
- Second **for** loop:

Implementation and Analysis

```
KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ G.V
      MAKE-SET(v)
  sort the edges of G.E into nondecreasing order by weight w
  for each (u, v) taken from the sorted list
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
  return A
```

- 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

Implementation and Analysis

```
KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ G.V
      MAKE-SET(v)
  sort the edges of G.E into nondecreasing order by weight w
  for each (u, v) taken from the sorted list
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
  return A
```

- 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 time: $O((V + E)\alpha(V)) + O(E \lg E) = O(E \lg E) = O(E \lg V)$

Implementation and Analysis

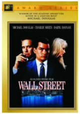
```
KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ G.V
      MAKE-SET(v)
  sort the edges of G.E into nondecreasing order by weight w
  for each (u, v) taken from the sorted list
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
  return A
```

- 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 time: $O((V+E)\alpha(V)) + O(E \lg E) = O(E \lg E) = O(E \lg V)$

  If edges already sorted time is $O(E\alpha(V))$ which is almost linear

Summary

- Greedy is good (sometimes)



- Prim's algorithm

  Min-priority queue for implementation

- Kruskal's algortihm

  Union-Find for implementation

- Many applications