Algorithms (VIII)

Yu Yu

Shanghai Jiaotong University

Review of the Previous Lecture

Chapter 4. Paths in graphs

Distances

DFS does not necessarily find the shortest paths.

Definition

The **distance** between two nodes is the length of the shortest path between them.

Breadth-first search

The algorithm

BFS(G, s)

Input: Graph $G = (V, E)$, directed or undirected; vertex $s \in V$

Output: For all vertices u reachable from s, dist(u) is set to the distance from s to u.

```
for all u ∈ V do
        dist(u) = ∞
dist(s) = 0
Q = [s] (queue containing just s)
while Q is not empty do
        u = eject(Q)
        for all edge (u, v) ∈ E do
                if dist(v) = ∞ then
                        inject(Q, v)
                        dist(v) = dist(u) + 1
```

Correctness and efficiency

Lemma

For each $d = 0, 1, 2, \ldots$, there is a moment at which (1) all nodes at distance $\leq d$ from s have their distances correctly set; (2) all other nodes have their distances set to ∞; and (3) the queue contains exactly the nodes at distance d.

Lemma

BFS has a running time of $O(|V| + |E|)$.

Lengths on edges

BFS treats all edges as having *the same length*, which is rarely true in applications where shortest paths are to be found.

Every edge $e \in E$ with a length ℓ_e.
If $e = (u, v)$, we will sometimes also write

$$\ell(u, v) \quad \text{or} \quad \ell_{uv}.$$

Dijkstra's algorithm

An adaption of breadth-first search

BFS finds shortest paths in any graph whose edges have unit length. Can we adapt it to a more general graph $G = (V, E)$ whose *edge lengths ℓ_e are positive integers*?

A simple trick:

> *For any edge $e = (u, v)$ of E, replace it by ℓ_e edges of length 1, by adding $\ell_e - 1$ dummy nodes between u and v.*

It might take time

$$O\left(|V| + \sum_{e \in E} \ell_e\right),$$

which is bad in case we have edges with long length.

Alarm clocks

- Set an alarm clock for node s at time 0.
- Repeat until there are no more alarms:
  Say the next alarm goes off at time T, for node y. Then:
  - The distance from s to u is T.
  - For each neighbor v of u in G:
    - If there is no alarm yet for v, set one for time $T + \ell(u, v)$.
    - If v's alarm is set for later than $T + \ell(u, v)$, then reset it to this earlier time.

Priority queue

Priority queue is a data structure usually implemented by *heap*.

- *Insert.* Add a new element to the set.
- *Decrease-key.* Accommodate the decrease in key value of a particular element.
- *Delete-min.* Return the element with the smallest key, and remove it from the set.
- *Make-queue.* Build a priority queue out of the given elements, with the given key values. (In many implementations, this is significantly faster than inserting the elements one by one.)

Dijkstra's shortest-path algorithm

DIJKSTRA(G, ℓ, s)

Input: Graph $G = (V, E)$, directed or undirected;
positive edge length $\{\ell_e \mid e \in E\}$; vertex $s \in V$

Output: For all vertices u reachable from s, $\texttt{dist}(u)$ is set
to the distance from s to u.

```
for all u ∈ V do
        dist(u) = ∞
        prev(u) = nil
dist(s) = 0
H = makequeue(V) (using dist-values as keys)
while H is not empty do
        u = deletemin(H)
        for all edge (u, v) ∈ E do
                if dist(v) > dist(u) + ℓ(u, v) then
                        dist(v) = dist(u) + ℓ(u, v)
                        prev(v) = u
                        decreasekey(H, v)
```

An alternative derivation

Initialize $\texttt{dist}(s) = 0$, other $\texttt{dist}(\cdot)$ to ∞
$R = \{\ \}$ (the "known region")
while $R \neq V$ **do**
    Pick the node $v \notin R$ with smallest $\texttt{dist}(\cdot)$
    Add v to R
    for all edge $(v, z) \in E$ **do**
        if $\texttt{dist}(z) > \texttt{dist}(v) + \ell(v, z)$ **then**
            $\texttt{dist}(z) = \texttt{dist}(v) + \ell(v, z)$

Key property

At the end of each iteration of the while loop, the following conditions hold:

(1) there is a value d such that all nodes in R are at distance $\leq d$ from s and all nodes outside R are at distance $\geq d$ from s;

(2) for every node u, the value $\texttt{dist}(u)$ is the length of the shortest path from s to u whose intermediate nodes are constrained to be in R (if no such path exists, the value is ∞).

Running time

Since `makequeue` takes at most as long as $|V|$ `insert` operations, we get a total of $|V|$ `deletemin` and $|V| + |E|$ `insert`/`decreasekey` operations.

The time needed for these varies by implementation; for instance, a *binary heap* gives an overall running time of

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

Which heap is best?

Implementation	`deletemin`	`insert`/ `decreasekey`	$\lvert V\rvert \times$ `deletemin`+ $(\lvert V\rvert + \lvert E\rvert) \times$ `insert`
Array	$O(\lvert V\rvert)$	$O(1)$	$O(\lvert V\rvert^2)$
Binary heap	$O(\log \lvert V\rvert)$	$O(\log \lvert V\rvert)$	$O((\lvert V\rvert + \lvert E\rvert) \log \lvert V\rvert)$
d-ary heap	$O(\frac{d \log \lvert V\rvert}{\log d})$	$O(\frac{\log \lvert V\rvert}{\log d})$	$O(\frac{(d\lvert V\rvert + \lvert E\rvert) \log \lvert V\rvert}{\log d})$
Fibonacci heap	$O(\log \lvert V\rvert)$	$O(1)$ (amortized)	$O(\lvert V\rvert \log \lvert V\rvert + \lvert E\rvert)$

Priority queue implementations

Array

The simplest implementation of a priority queue is as an *unordered array* of key values for all potential elements (the vertices of the graph, in the case of Dijkstra's algorithm).

Initially, these values are set to ∞.

An `insert` or `decreasekey` is fast, because it just involves adjusting a key value, an $O(1)$ operation.

To `deletemin`, on the other hand, requires a linear-time scan of the list.

Binary heap

Here elements are stored in *a complete binary tree*.

In addition, a special ordering constraint is enforced:

> *the key value of any node of the tree is less than or equal to that of its children.*

In particular, therefore, the root always contains the smallest element.

To `insert`, place the new element at the bottom of the tree (in the first available position), and let it "*bubble up*."
The number of swaps is at most the height of the tree $\lfloor \log_2 n \rfloor$, when there are n elements.

A `decreasekey` is similar, except the element is already in the tree, so we let it bubble up from its current position.

To `deletemin`, return the root value.
To then remove this element from the heap, take the last node in the tree (in the rightmost position in the bottom row) and place it at the root.
Then let it "*sift down*." Again this takes $O(\log n)$ time.

d-ary heap

A *d*-**ary heap** is identical to a binary heap, except that nodes have *d children*.

This reduces the height of a tree with n elements to

$$\Theta(\log_d n) = \Theta((\log n)/(\log d))$$

`Inserts` are therefore speeded up by a factor of $\Theta(\log d)$.
`Deletemin` operations, however, take a little longer, namely $O(d \log_d n)$.

Shortest paths in the presence of negative edges

Negative edges

Negative edges

Dijkstra's algorithm works in part because the shortest path from the starting point s to any node v *must pass exclusively through nodes that are closer than* v.

Negative edges

Dijkstra's algorithm works in part because the shortest path from the starting point s to any node v *must pass exclusively through nodes that are closer than* v*.*

This no longer holds when edge lengths can be negative.

Negative edges

Dijkstra's algorithm works in part because the shortest path from the starting point s to any node v *must pass exclusively through nodes that are closer than* v.

This no longer holds when edge lengths can be negative.

What needs to be changed in order to accommodate this new complication?

Negative edges

Dijkstra's algorithm works in part because the shortest path from the starting point s to any node v *must pass exclusively through nodes that are closer than* v*.*

This no longer holds when edge lengths can be negative.

What needs to be changed in order to accommodate this new complication?
A crucial invariant of Dijkstra's algorithm is that the `dist` values it maintains are always either *overestimates* or *exactly correct*.

Negative edges

Dijkstra's algorithm works in part because the shortest path from the starting point s to any node v *must pass exclusively through nodes that are closer than v.*

This no longer holds when edge lengths can be negative.

What needs to be changed in order to accommodate this new complication? A crucial invariant of Dijkstra's algorithm is that the `dist` values it maintains are always either *overestimates* or *exactly correct*.

They start off at ∞, and the only way they ever change is by updating along an edge:

$\underline{\textsc{update}}((u, v) \in E)$

$\texttt{dist}(v) = \min\{\texttt{dist}(v), \texttt{dist}(u) + \ell(u, v)\}$

Update

Update

$\underline{\textsc{Update}}((u, v) \in E)$

$\texttt{dist}(v) = \min\{\texttt{dist}(v), \texttt{dist}(u) + \ell(u, v)\}$

Update

$\underline{\text{UPDATE}}((u, v) \in E)$

$\texttt{dist}(v) = \min\{\texttt{dist}(v), \texttt{dist}(u) + \ell(u, v)\}$

1. It gives the correct distance to v in the particular case where u is the *second-last* node in the shortest path to v, and $\texttt{dist}(u)$ is correctly set.

Update

$\underline{\text{UPDATE}}((u, v) \in E)$

$\texttt{dist}(v) = \min\{\texttt{dist}(v), \texttt{dist}(u) + \ell(u, v)\}$

1. It gives the correct distance to v in the particular case where u is the *second-last* node in the shortest path to v, and $\texttt{dist}(u)$ is correctly set.

2. It will never make $\texttt{dist}(v)$ too small, and in this sense it is *safe*. For instance, a slew of extraneous `update`'s can't hurt.

Update

$\underline{\text{UPDATE}}((u, v) \in E)$

$\texttt{dist}(v) = \min\{\texttt{dist}(v), \texttt{dist}(u) + \ell(u, v)\}$

1. It gives the correct distance to v in the particular case where u is the *second-last* node in the shortest path to v, and $\texttt{dist}(u)$ is correctly set.
2. It will never make $\texttt{dist}(v)$ too small, and in this sense it is *safe*. For instance, a slew of extraneous `update`'s can't hurt.

Let

$$s \to u_1 \to u_2 \to u_3 \to \cdots \to u_k \to t$$

be a shortest path from s to t.

Update

$\underline{\text{UPDATE}}((u, v) \in E)$

$\texttt{dist}(v) = \min\{\texttt{dist}(v), \texttt{dist}(u) + \ell(u, v)\}$

1. It gives the correct distance to v in the particular case where u is the *second-last* node in the shortest path to v, and $\texttt{dist}(u)$ is correctly set.

2. It will never make $\texttt{dist}(v)$ too small, and in this sense it is *safe*. For instance, a slew of extraneous `update`'s can't hurt.

Let

$$s \to u_1 \to u_2 \to u_3 \to \cdots \to u_k \to t$$

be a shortest path from s to t.
This path can have at most $|V| - 1$ edges (why?).

Update

$\underline{\text{UPDATE}}((u, v) \in E)$

$\texttt{dist}(v) = \min\{\texttt{dist}(v), \texttt{dist}(u) + \ell(u, v)\}$

1. It gives the correct distance to v in the particular case where u is the *second-last* node in the shortest path to v, and $\texttt{dist}(u)$ is correctly set.

2. It will never make $\texttt{dist}(v)$ too small, and in this sense it is *safe*. For instance, a slew of extraneous `update`'s can't hurt.

Let

$$s \rightarrow u_1 \rightarrow u_2 \rightarrow u_3 \rightarrow \cdots \rightarrow u_k \rightarrow t$$

be a shortest path from s to t.
This path can have at most $|V| - 1$ edges (why?). If the sequence of updates performed includes $(s, u_1), (u_1, u_2), \ldots, (u_k, t)$, *in that order* (though not necessarily consecutively), then by 1 the distance to t will be correctly computed.

Update

$\underline{\text{UPDATE}}((u, v) \in E)$

$\mathtt{dist}(v) = \min\{\mathtt{dist}(v), \mathtt{dist}(u) + \ell(u, v)\}$

1. It gives the correct distance to v in the particular case where u is the *second-last* node in the shortest path to v, and $\mathtt{dist}(u)$ is correctly set.
2. It will never make $\mathtt{dist}(v)$ too small, and in this sense it is *safe*. For instance, a slew of extraneous `update`'s can't hurt.

Let

$$s \to u_1 \to u_2 \to u_3 \to \cdots \to u_k \to t$$

be a shortest path from s to t.
This path can have at most $|V| - 1$ edges (why?). If the sequence of updates performed includes $(s, u_1), (u_1, u_2), \ldots, (u_k, t)$, *in that order* (though not necessarily consecutively), then by 1 the distance to t will be correctly computed.
It doesn't matter what other updates occur on these edges, or what happens in the rest of the graph, because updates are *safe*.

Bellman-Ford algorithm

Bellman-Ford algorithm

But still, if we don't know all the shortest paths beforehand, how can we be sure to update the right edges in the right order?

Bellman-Ford algorithm

But still, if we don't know all the shortest paths beforehand, how can we be sure to update the right edges in the right order?
We simply update all the edges, $|V| - 1$ *times*!

Bellman-Ford algorithm

But still, if we don't know all the shortest paths beforehand, how can we be sure to update the right edges in the right order?
We simply update all the edges, $|V| - 1$ *times*!

SHORTEST-PATHS(G, ℓ, s)

Input: Graph $G = (V, E)$;
edge lengths $\{\ell_e \mid e \in E\}$ with no negative cycles, vertex $s \in V$

Output: For all vertices u reachable from s, $\texttt{dist}(u)$ is set to the distance from s to u.

1. **for all** $u \in V$ **do**
2. $\quad\texttt{dist}(u) = \infty$
3. $\quad\texttt{prev}(u) = \texttt{nil}$
4. $\texttt{dist}(s) = 0$
5. repeat $|V| - 1$ times:
6. $\quad$**for all** $e \in E$ **do**
7. $\quad\quad\texttt{update}(e)$

Bellman-Ford algorithm

But still, if we don't know all the shortest paths beforehand, how can we be sure to update the right edges in the right order?
We simply update all the edges, $|V| - 1$ *times*!

SHORTEST-PATHS(G, ℓ, s)

Input: Graph $G = (V, E)$;
edge lengths $\{\ell_e \mid e \in E\}$ with no negative cycles, vertex $s \in V$

Output: For all vertices u reachable from s, $\texttt{dist}(u)$ is set to the distance from s to u.

```
for all u ∈ V do
        dist(u) = ∞
        prev(u) = nil
dist(s) = 0
repeat |V| − 1 times:
        for all e ∈ E do
                update(e)
```

Running time: $O(|V| \cdot |E|)$.

Negative cycles

Negative cycles

If the graph has a *negative cycle*, then it doesn't make sense to even ask about shortest path.

Negative cycles

If the graph has a *negative cycle*, then it doesn't make sense to even ask about shortest path.

How to detect the existence of negative cycles:

Negative cycles

If the graph has a *negative cycle*, then it doesn't make sense to even ask about shortest path.

How to detect the existence of negative cycles:

Instead of stopping after $|V| - 1$ iterations, perform one extra round.

Negative cycles

If the graph has a *negative cycle*, then it doesn't make sense to even ask about shortest path.

How to detect the existence of negative cycles:

Instead of stopping after $|V| - 1$ iterations, perform one extra round.
There is a negative cycle if and only if some dist value is reduced during this final round.

Shortest paths in dags

There are two subclasses of graphs that automatically exclude the possibility of negative cycles:

There are two subclasses of graphs that automatically exclude the possibility of negative cycles:

graphs without negative edges, and

There are two subclasses of graphs that automatically exclude the possibility of negative cycles:

graphs without negative edges, and *graphs without cycles*.

There are two subclasses of graphs that automatically exclude the possibility of negative cycles:

graphs without negative edges, and *graphs without cycles*.

We already know how to efficiently handle the former.

There are two subclasses of graphs that automatically exclude the possibility of negative cycles:

graphs without negative edges, and *graphs without cycles*.

We already know how to efficiently handle the former. We will now see how the single-source shortest-path problem can be solved in just *linear time* on directed acyclic graphs.

There are two subclasses of graphs that automatically exclude the possibility of negative cycles:

graphs without negative edges, and *graphs without cycles*.

We already know how to efficiently handle the former. We will now see how the single-source shortest-path problem can be solved in just *linear time* on directed acyclic graphs.

As before, we need to perform a sequence of updates that includes every shortest path as a subsequence.

There are two subclasses of graphs that automatically exclude the possibility of negative cycles:

graphs without negative edges, and *graphs without cycles*.

We already know how to efficiently handle the former. We will now see how the single-source shortest-path problem can be solved in just *linear time* on directed acyclic graphs.

As before, we need to perform a sequence of updates that includes every shortest path as a subsequence.

In any path of a dag, the vertices appear in increasing linearized order.

A single-source shortest-path algorithm for directed acyclic graphs

A single-source shortest-path algorithm for directed acyclic graphs

DAG-SHORTEST-PATHS(G, ℓ, s)

Input: Dag $G = (V, E)$;
edge lengths $\{\ell_e \mid e \in E\}$, vertex $s \in V$

Output: For all vertices u reachable from s, $\texttt{dist}(u)$ is set to the distance from s to u.

for all $u \in V$ **do**
    $\texttt{dist}(u) = \infty$
    $\texttt{prev}(u) = \texttt{nil}$
$\texttt{dist}(s) = 0$
Linearize G
for each $u \in V$ in linearized order **do**
    for all edges $(u, v) \in E$ **do**
        $\texttt{update}(e)$

A single-source shortest-path algorithm for directed acyclic graphs

```
DAG-SHORTEST-PATHS(G, ℓ, s)
  Input:    Dag G = (V, E);
            edge lengths {ℓ_e | e ∈ E}, vertex s ∈ V
  Output:   For all vertices u reachable from s, dist(u) is set
            to the distance from s to u.

  for all u ∈ V do
        dist(u) = ∞
        prev(u) = nil
  dist(s) = 0
  Linearize G
  for each u ∈ V in linearized order do
        for all edges (u, v) ∈ E do
              update(e)
```

Notice that our scheme doesn't require edges to be positive.

A single-source shortest-path algorithm for directed acyclic graphs

DAG-SHORTEST-PATHS(G, ℓ, s)

Input: Dag $G = (V, E)$;
edge lengths $\{\ell_e \mid e \in E\}$, vertex $s \in V$

Output: For all vertices u reachable from s, `dist`(u) is set to the distance from s to u.

for all $u \in V$ **do**
    `dist`$(u) = \infty$
    `prev`$(u) =$ `nil`
`dist`$(s) = 0$
Linearize G
for each $u \in V$ in linearized order **do**
    for all edges $(u, v) \in E$ **do**
        `update`(e)

Notice that our scheme doesn't require edges to be positive.
In particular, we can find longest paths in a dag by the same algorithm: *just negate all edge lengths*.

Chapter 5. Greedy algorithms

Minimum spanning trees

Building a network

Building a network

Suppose you are asked to network a collection of computers by linking selected pairs of them.

Building a network

Suppose you are asked to network a collection of computers by linking selected pairs of them.

This translates into a graph problem in which

- nodes are computers,
- undirected edges are potential links, each with a *maintenance cost*.

Building a network

Suppose you are asked to network a collection of computers by linking selected pairs of them.

This translates into a graph problem in which

- nodes are computers,
- undirected edges are potential links, each with a *maintenance cost*.

The goal is to

- pick enough of these edges that the nodes are *connected*,
- the total maintenance cost is *minimum*.

Properties of the optimal solutions

Properties of the optimal solutions

Lemma (1)

Removing a cycle edge cannot disconnect a graph.

Properties of the optimal solutions

Lemma (1)

Removing a cycle edge cannot disconnect a graph.

So the solution must be connected and acyclic: undirected graphs of this kind are called **trees**.

Properties of the optimal solutions

Lemma (1)

Removing a cycle edge cannot disconnect a graph.

So the solution must be connected and acyclic: undirected graphs of this kind are called **trees**. A tree with *minimum total weight*, is a **minimum spanning tree**.

Properties of the optimal solutions

Lemma (1)

Removing a cycle edge cannot disconnect a graph.

So the solution must be connected and acyclic: undirected graphs of this kind are called **trees**. A tree with *minimum total weight*, is a **minimum spanning tree**.

Input:	An undirected graph $G = (V, E)$; edge weights w_e
Output:	A tree $T = (V, E')$ with $E' \subseteq E$ that *minimizes*

$$\text{weight}(T) = \sum_{e \in E'} w_e.$$

Trees

Trees

Lemma (2)

A tree on n nodes has $n - 1$ edges.

Trees

Lemma (2)

A tree on n nodes has $n-1$ edges.

Lemma (3)

Any connected, undirected graph $G=(V,E)$ with $|E|=|V|-1$ is a tree.

Trees

Lemma (2)

A tree on n nodes has $n-1$ edges.

Lemma (3)

Any connected, undirected graph $G = (V, E)$ with $|E| = |V| - 1$ is a tree.

Lemma (4)

An undirected graph is a tree if and only if there is a unique *path between any pair of nodes.*

A greedy approach

A greedy approach

Kruskal's minimum spanning tree algorithm starts with the empty graph and then selects edges from E according to the following rule.

A greedy approach

Kruskal's minimum spanning tree algorithm starts with the empty graph and then selects edges from E according to the following rule.

Repeatedly add the next lightest edge that doesn't produce a cycle.

A greedy approach

Kruskal's minimum spanning tree algorithm starts with the empty graph and then selects edges from E according to the following rule.

Repeatedly add the next lightest edge that doesn't produce a cycle.

The correctness of Kruskal's method follows from a certain *cut property*.

The cut property

The cut property

Lemma (Cut property)

Suppose edges X are part of a minimum spanning tree (MST) of $G = (V, E)$. Pick any subset of nodes S for which X does not cross between S and $V \setminus S$, and let e be the lightest edge across this partition. Then

$$X \cup \{e\}$$

is part of some MST.

The cut property

Lemma (Cut property)

Suppose edges X are part of a minimum spanning tree (MST) of $G = (V, E)$. Pick any subset of nodes S for which X does not cross between S and $V \setminus S$, and let e be the lightest edge across this partition. Then

$$X \cup \{e\}$$

is part of some MST.

A **cut** is any partition of the vertices into two groups, S and $V \setminus S$.

The cut property

Lemma (**Cut property**)

Suppose edges X are part of a minimum spanning tree (MST) of $G = (V, E)$. Pick any subset of nodes S for which X does not cross between S and $V \setminus S$, and let e be the lightest edge across this partition. Then

$$X \cup \{e\}$$

is part of some MST.

A **cut** is any partition of the vertices into two groups, S and $V \setminus S$.

The cut property says that it is always safe to add the lightest edge across any cut (that is, between a vertex in S and one in $V \setminus S$), provided X has no edges across the cut.

Proof of the cut property

Edges X are part of some MST T; if the new edge e also happens to be part of T, then there is nothing to prove.

Proof of the cut property

Edges X are part of some MST T; if the new edge e also happens to be part of T, then there is nothing to prove.

So assume e is not in T.

Proof of the cut property

Edges X are part of some MST T; if the new edge e also happens to be part of T, then there is nothing to prove.

So assume e is not in T. We will construct a different MST T' containing $X \cup \{e\}$ by altering T slightly, changing just one of its edges.

Proof of the cut property

Edges X are part of some MST T; if the new edge e also happens to be part of T, then there is nothing to prove.

So assume e is not in T. We will construct a different MST T' containing $X \cup \{e\}$ by altering T slightly, changing just one of its edges.

Add edge e to T.

Proof of the cut property

Edges X are part of some MST T; if the new edge e also happens to be part of T, then there is nothing to prove.

So assume e is not in T. We will construct a different MST T' containing $X \cup \{e\}$ by altering T slightly, changing just one of its edges.

Add edge e to T. Since T is connected, it already has a path between the endpoints of e, so adding e creates a cycle.

Proof of the cut property

Edges X are part of some MST T; if the new edge e also happens to be part of T, then there is nothing to prove.

So assume e is not in T. We will construct a different MST T' containing $X \cup \{e\}$ by altering T slightly, changing just one of its edges.

Add edge e to T. Since T is connected, it already has a path between the endpoints of e, so adding e creates a cycle. *This cycle must also have some other edge e' across the cut $(S, V \setminus S)$.*

Proof of the cut property

Edges X are part of some MST T; if the new edge e also happens to be part of T, then there is nothing to prove.

So assume e is not in T. We will construct a different MST T' containing $X \cup \{e\}$ by altering T slightly, changing just one of its edges.

Add edge e to T. Since T is connected, it already has a path between the endpoints of e, so adding e creates a cycle. *This cycle must also have some other edge e' across the cut $(S, V \setminus S)$.*

If we now remove e'

$$T' = T \cup \{e\} \setminus \{e'\}$$

which we will show to be a tree.

Proof of the cut property

Edges X are part of some MST T; if the new edge e also happens to be part of T, then there is nothing to prove.

So assume e is not in T. We will construct a different MST T' containing $X \cup \{e\}$ by altering T slightly, changing just one of its edges.

Add edge e to T. Since T is connected, it already has a path between the endpoints of e, so adding e creates a cycle. *This cycle must also have some other edge e' across the cut $(S, V \setminus S)$.*

If we now remove e'

$$T' = T \cup \{e\} \setminus \{e'\}$$

which we will show to be a tree.

T' is connected by Lemma (1), since e' is a cycle edge.

Proof of the cut property

Edges X are part of some MST T; if the new edge e also happens to be part of T, then there is nothing to prove.

So assume e is not in T. We will construct a different MST T' containing $X \cup \{e\}$ by altering T slightly, changing just one of its edges.

Add edge e to T. Since T is connected, it already has a path between the endpoints of e, so adding e creates a cycle. *This cycle must also have some other edge e' across the cut $(S, V \setminus S)$.*

If we now remove e'

$$T' = T \cup \{e\} \setminus \{e'\}$$

which we will show to be a tree.

T' is connected by Lemma (1), since e' is a cycle edge. And it has the same number of edges as T; so by Lemmas (2) and (3), it is also a tree.

Proof of the cut property (cont'd)

Proof of the cut property (cont'd)

T' is a minimum spanning tree:

Proof of the cut property (cont'd)

T' is a minimum spanning tree:

$$\text{weight}(T') = \text{weight}(T) + w(e) - w(e').$$

Proof of the cut property (cont'd)

T' is a minimum spanning tree:

$$\text{weight}(T') = \text{weight}(T) + w(e) - w(e').$$

Both e and e' cross between S and $V \setminus S$, and e is the lightest edge of this type.

Proof of the cut property (cont'd)

T' is a minimum spanning tree:

$$\text{weight}(T') = \text{weight}(T) + w(e) - w(e').$$

Both e and e' cross between S and $V \setminus S$, and e is the lightest edge of this type. Therefore $w(e) \leq w(e')$, and

$$\text{weight}(T') \leq \text{weight}(T).$$

Proof of the cut property (cont'd)

T' is a minimum spanning tree:

$$\text{weight}(T') = \text{weight}(T) + w(e) - w(e').$$

Both e and e' cross between S and $V \setminus S$, and e is the lightest edge of this type. Therefore $w(e) \leq w(e')$, and

$$\text{weight}(T') \leq \text{weight}(T).$$

Since T is an MST, it must be the case that $\text{weight}(T') = \text{weight}(T)$ and that T' is also an MST.

Kruskal's algorithm

Kruskal's algorithm

KRUSKAL(G, w)

Input: A connected undirected graph $G = (V, E)$ with edge weight w_e

Output: A minimum spanning tree defined by the edges X.

```
for all u ∈ V do
        makeset(u)
X = {}
Sort the edges E by weight
for all edge {u, v} ∈ E in increasing order of weight do
        if find(u) ≠ find(v) then
                add edge {u, v} to X
                union(u, v)
```

Kruskal's algorithm

```
KRUSKAL(G, w)
  Input:    A connected undirected graph G = (V, E) with edge weight w_e
  Output:   A minimum spanning tree defined by the edges X.

for all u ∈ V do
        makeset(u)
X = {}
Sort the edges E by weight
for all edge {u, v} ∈ E in increasing order of weight do
        if find(u) ≠ find(v) then
                add edge {u, v} to X
                union(u, v)
```

$\lvert V \rvert$	`makeset`(x)	create a singleton set containing x
$2 \cdot \lvert E \rvert$	`find`(x)	find the set that x belongs to
$\lvert V \rvert - 1$	`union`(x, y)	merge the sets containing x and y

A data structure for disjoint sets

A data structure for disjoint sets

Union by rank

A data structure for disjoint sets

Union by rank

We store a set is by *a directed tree*.

A data structure for disjoint sets

Union by rank

We store a set is by *a directed tree*. Nodes of the tree are elements of the set, arranged in no particular order, and each has parent pointers that eventually lead up to the *root* of the tree.

A data structure for disjoint sets

Union by rank

We store a set is by *a directed tree*. Nodes of the tree are elements of the set, arranged in no particular order, and each has parent pointers that eventually lead up to the *root* of the tree.
This root element is a convenient *representative*, or *name*, for the set.

A data structure for disjoint sets

Union by rank

We store a set is by *a directed tree*. Nodes of the tree are elements of the set, arranged in no particular order, and each has parent pointers that eventually lead up to the *root* of the tree.
This root element is a convenient *representative*, or *name*, for the set. It is distinguished from the other elements by the fact that its parent pointer is a *self-loop*.

Union by rank

We store a set is by *a directed tree*. Nodes of the tree are elements of the set, arranged in no particular order, and each has parent pointers that eventually lead up to the *root* of the tree.
This root element is a convenient *representative*, or *name*, for the set. It is distinguished from the other elements by the fact that its parent pointer is a *self-loop*.

In addition to a parent pointer π, each node also has a **rank** that, for the time being, should be interpreted as the height of the subtree hanging from that node.

Union by rank

Union by rank

$\underline{\text{MAKESET}}(x)$

$\pi(x) = x$
$\texttt{rank}(x) = 0$

Union by rank

MAKESET(x)

$\pi(x) = x$
$\texttt{rank}(x) = 0$

FIND(x)

while $x \neq \pi(x)$ **do** $x = \pi(x)$
return x

Union by rank

$\underline{\text{MAKESET}}(x)$

$\pi(x) = x$
$\texttt{rank}(x) = 0$

$\underline{\text{FIND}}(x)$

while $x \neq \pi(x)$ **do** $x = \pi(x)$
return x

`makeset` is a constant-time operation.

Union by rank

MAKESET(x)

$\pi(x) = x$
$\texttt{rank}(x) = 0$

FIND(x)

while $x \neq \pi(x)$ **do** $x = \pi(x)$
return x

`makeset` is a constant-time operation.

`find` follows parent pointers to the root of the tree and therefore takes time *proportional to the height of the tree*.

Union by rank

$\underline{\text{MAKESET}}(x)$

$\pi(x) = x$
$\texttt{rank}(x) = 0$

$\underline{\text{FIND}}(x)$

while $x \neq \pi(x)$ **do** $x = \pi(x)$
return x

`makeset` is a constant-time operation.

`find` follows parent pointers to the root of the tree and therefore takes time *proportional to the height of the tree*.

The tree actually gets built via the third operation, `union`, and so we must make sure that this procedure keeps trees *shallow*.

Union

Union

<u>UNION</u>(x, y)

$r_x = \texttt{find}(x)$
$r_y = \texttt{find}(y)$
if $r_x = r_y$ **then** return
if $\texttt{rank}(r_x) > \texttt{rank}(r_y)$
    then $\pi(r_u) = r_x$
else
    $\pi(r_x) = r_y$
    if $\texttt{rank}(r_x) = \texttt{rank}(r_y)$ **then** $\texttt{rank}(r_y) = \texttt{rank}(r_y) + 1$

Properties

Properties

Lemma (1)

For any non-root x, $rank(x) < rank(\pi(x))$

Properties

Lemma (1)

For any non-root x, $rank(x) < rank(\pi(x))$

Lemma (2)

Any root node of rank k has least 2^k *nodes in its tree.*

Properties

Lemma (1)

For any non-root x, $\mathit{rank}(x) < \mathit{rank}(\pi(x))$

Lemma (2)

Any root node of rank k *has least* 2^k *nodes in its tree.*

Lemma (3)

If there are n *elements overall, there can be at most* $n/2^k$ *nodes of* rank k.