Algorithms and Data Structures

Minimum Spanning Trees - Greedy Algorithms Running Time

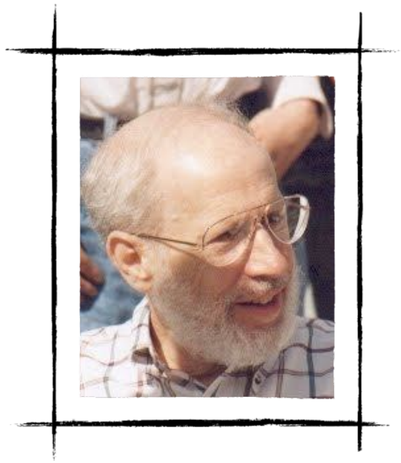
Minimum Spanning Tree

$G'=(V, T)$ is a spanning tree and the problem is called
the Minimum Spanning Tree problem.

Consider a *connected* graph $G=(V, E)$, such that for every edge $e=\{v,w\}$ of E, there is an associated positive cost c_e.

Goal: Find a subset T of E so that the graph $G'=(V, T)$ is connected and the total cost $\sum_{e \in T} c_e$ is minimised.

Kruskal’s Algorithm



Start with an empty set of edges T.

Add one edge to T.

Which one?

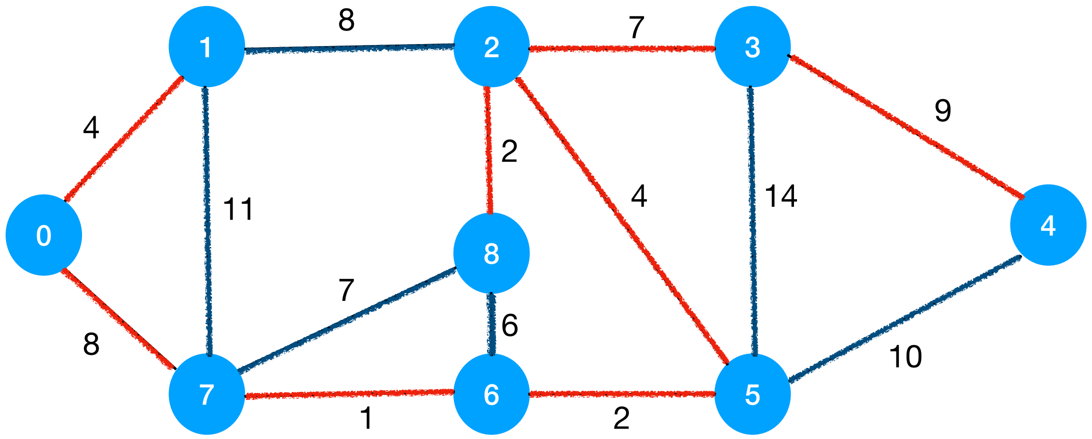
The one with the minimum cost c_e.

We continue like this.

Do we always add the new edge e to T ?

Only if we don’t introduce any cycles.

Example



Prim's Algorithm

Start with an empty set of edges T.

Start with a node s.

Add an edge $e=\{s,w\}$ to T.

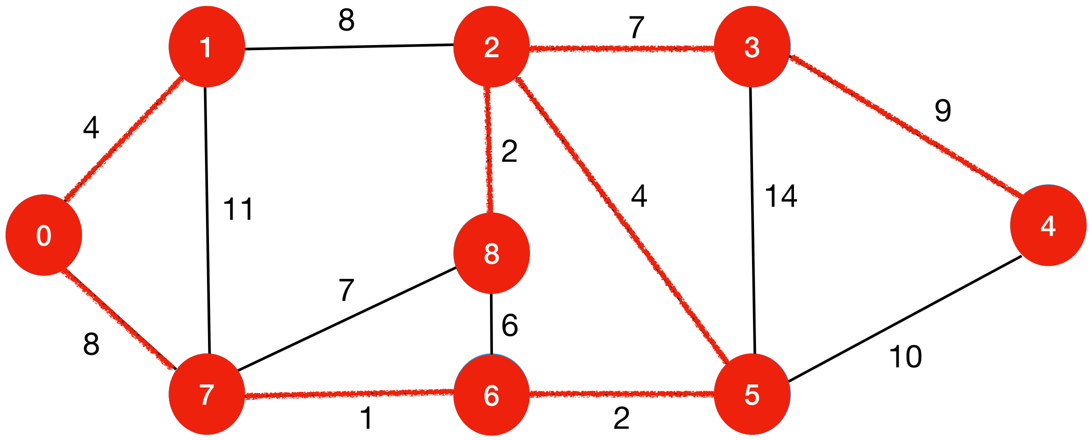
Which one?

The one with the minimum cost c_e.

We continue like this.

We only consider edges to neighbours that are not in the spanning tree.

Example



Prim’s algorithm running time

We add nodes to the expanding spanning tree S.

We need to figure out which node to add next.

We need to know the attachment cost of each node:

$$a(v) = \min_{e=\{u,v\}:u\in S} c_e$$

Naive solution: For every step run over all candidates.

$\Theta(n^2)$

Priority Queues

Priority queue: A data structure that maintains

A set of elements S.

Each with an associated value, key(v).

The values denote *priorities*.

For Min-Priority Queues, the elements with the smallest values are those with the highest priority.

Priority Queue Operations

Insert(Q, v) inserts a new item v in the priority queue.

FindMin(Q) finds the element with the maximum priority (the smallest value) in the priority queue and returns it (but does not remove it).

ExtractMin(Q) finds the element with the maximum priority (smallest value) in the priority queue, returns it, and deletes it from the queue.

Priority Queue Operations

Insert(Q, v) inserts a new item v in the priority queue.

FindMin(Q) finds the element with the maximum priority (the smallest value) in the priority queue and returns it (but does not remove it).

ExtractMin(Q) finds the element with the maximum priority (smallest value) in the priority queue, returns it, and deletes it from the queue.

ChangeKey(Q, v, α) sets key$(v) = \alpha$.

Priority Queues

The Priority Queue is an *abstract* data type.

In reality, we have to implement it with known data structures.

Many implementations exists, the usual one is with *heaps*.

We will not cover this here; it was covered in IADS last year.

e.g. see KT Chapter 2.5, CLRS Chapter 6.5. (but you would have to also read 6.1 - 6.3).

Priority Queue Operations

Insert(Q, v) inserts a new item v in the priority queue. $O(\lg n)$

FindMin(Q) finds the element with the maximum priority (the smallest value) in the priority queue and returns it (but does not remove it). $O(1)$

ExtractMin(Q) finds the element with the maximum priority (smallest value) in the priority queue, returns it, and deletes it from the queue. $O(\lg n)$

ChangeKey(Q, v, α) sets key$(v) = \alpha$. $O(\lg n)$

Prim's algorithm running time

We add nodes to the expanding spanning tree S.

We need to figure out which node to add next.

We need to know the attachment cost of each node:

$$a(v) = \min_{e=\{u,v\}:u\in S} c_e$$

Running time: $O(m \log n)$

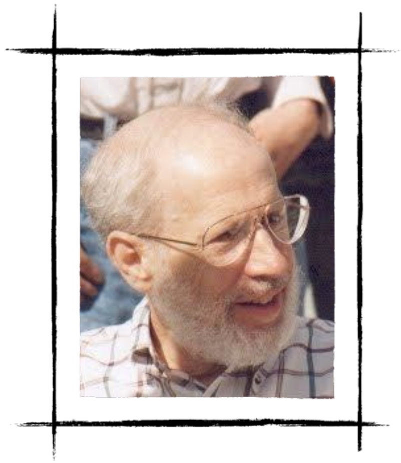
PQ solution: Insert the nodes in a PQ, with the attachment cost as the key.

Run ExtractMin(Q) to find the next node. How many times? $n-1$

ChangeKey(Q, v, α) to update the attachment cost. How many times?

At most once per edge, thus $\leq m$

Kruskal’s Algorithm



Start with an empty set of edges T.

Add one edge to T.

Which one?

The one with the minimum cost c_e.

We continue like this.

What is the tricky part here?

Do we always add the new edge e to T ?

Only if we don’t introduce any cycles.

Identifying connected components

Assume that we consider edge $e = \{v, w\}$ for possible inclusion in our spanning tree.

If v and w are in different connected components, we are good (why?)

Otherwise we should not add this edge (why?)

How can we find the connected component of v?

Graph Traversal (Search)

We would like to go over all the possible nodes of an (undirected) graph.

There are different ways of doing that.

Two systematic ways:

Depth-First Search

Breadth-First Search

KT Chapter 3.2.

CLRS Chapter 20.2, 20.3

Implementation idea

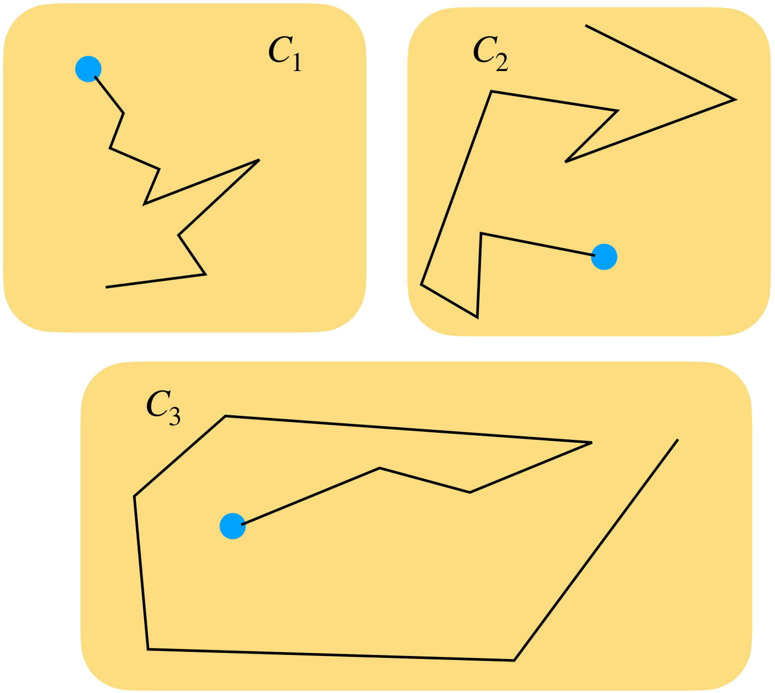
Run DFS/BFS from v and see if w is part of its connected component.

Equivalently: see if the DFS/BFS from v reaches w.

Running time: DFS/BFS takes time $\Theta(m+n)$, and we have to do that for every edge → $\Omega(m^2)$.

We actually don’t have to compute the connected component for every edge! We can compute all the connected components in $O(m+n)$ time.

Finding all connected components



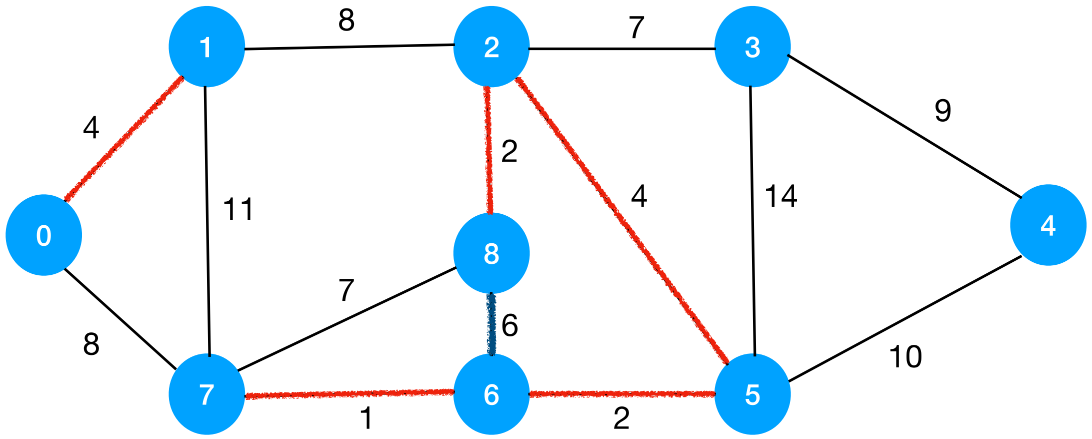
Implementation idea

We actually don’t have to compute the connected component for every edge! We can compute all the connected components in $O(m+n)$ time.

So, for a candidate edge $e = \{w, v\}$ we can check in $O(1)$ time whether the endpoints are in the same connected component or not.

But on which graph?

Example



Implementation idea

We actually don't have to compute the connected component for every edge! We can compute all the connected components in $O(m+n)$ time.

So, for a candidate edge $e = \{w, v\}$ we can check in $O(1)$ time whether the endpoints are in the same connected component or not.

But on which graph?

$G = (V, T)$, where T is the set of edges of the spanning tree that we are developing.

This changes over time!

How many times can it change? $\Omega(m)$ times

Overall running time: $\Omega(m^2)$

The Union-Find Data Structure

An *abstract* data structure which maintains disjoint sets
(e.g., here connected components of a graph).

Its operations will allow us to *find* the set containing an element u, and to *merge* two sets into a single set
(e.g., when we add edges so that now two nodes are part of the same component, when they were not before).

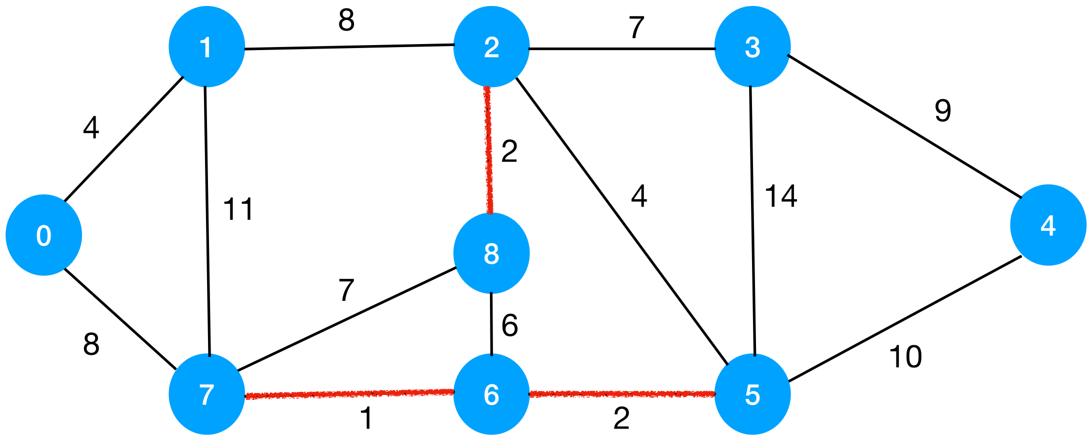
Union-Find Operations

MakeUnionFind(S) creates a new Union-Find data structure where every element in S is a singleton set, i.e., $\{v_1\}, \{v_2,\}, \ldots \{v_k\}$ for $S = \{v_1, v_2, \ldots, v_k\}$

Find(u) returns the name of the set containing element u.

Union(A, B) changes the Union-Find data structure by merging the sets A and B into a single set.

Example



MakeUnionFind(V)

Union($\{6,7\}, \{5\}$)

Union($\{6\}, \{7\}$)

Union($\{2\}, \{8\}$)

Kruskal’s algorithm using Union-Find

We first sort the edges in terms of non-decreasing cost.

We run MakeUnionFind(V) to initialise the components of $G = (V, T)$ to each contain one node.

When considering an edge $e = \{w, v\}$, we check if Find(w) = Find(v).

If yes, we ignore the edge and continue with the next one.

If no, we add the edge to the spanning tree and run Union(Find(u), Find(v)) to merge the two components.

Union-Find Operations

MakeUnionFind(S) creates a new Union-Find data structure where every element in S is a singleton set, i.e., $\{v_1\}, \{v_2,\}, \ldots \{v_k\}$ for $S = \{v_1, v_2, \ldots, v_k\}$ $T\left(\text{MakeUnionFind}(v)\right)$

Find(u) returns the name of the set containing element u. $T\left(\text{Find}(u)\right)$

Union(A, B) changes the Union-Find data structure by merging the sets A and B into a single set. $T\left(\text{Union}(A, B)\right)$

Kruskal’s algorithm using Union-Find

We first sort the edges in terms of non-decreasing cost.

We run MakeUnionFind(V) to initialise the components of $G = (V, T)$ to each contain one node.

When considering an edge $e = \{w, v\}$, we check if Find(w) = Find(v).

If yes, we ignore the edge and continue with the next one.

If no, we add the edge to the spanning tree and run Union(Find(u), Find(v)) to merge the two components.

Kruskal's algorithm
Running time

We first sort the edges in terms of non-decreasing cost.

$O(m \log m) = O(m \log n)$ (why?)

We run MakeUnionFind(V) to initialise the components of $G = (V, T)$ to each contain one node. $T\left(\text{MakeUnionFind}(v)\right)$

When considering an edge $e = \{w, v\}$, we check if Find(w) = Find(v).

$\leq 2m \cdot T\left(\text{Find}(u)\right)$

If yes, we ignore the edge and continue with the next one.

If no, we add the edge to the spanning tree and run Union(Find(u), Find(v)) to merge the two components.

$\leq (n-1) \cdot T\left(\text{Union}(A, B)\right) \leq m \cdot T\left(\text{Union}(A, B)\right)$

Union-Find Operations

MakeUnionFind(S) creates a new Union-Find data structure where every element in S is a singleton set, i.e., $\{v_1\}, \{v_2,\}, \ldots \{v_k\}$ for $S = \{v_1, v_2, \ldots, v_k\}$ $T\left(\text{MakeUnionFind}(v)\right)$

Find(u) returns the name of the set containing element u. $T\left(\text{Find}(u)\right)$

Union(A, B) changes the Union-Find data structure by merging the sets A and B into a single set. $T\left(\text{Union}(A, B)\right)$

Suffices to show $T\left(\text{Find}(u)\right)$, and $T\left(\text{Union}(A, B)\right)$ are $O(\log n)$ and $T\left(\text{MakeUnionFind}(v)\right)$ is $O(m \log n)$

The Union-Find Data Structure

An *abstract* data structure which maintains disjoint sets
(e.g., here connected components of a graph).

Its operations will allow us to *find* the set containing an element u, and to *merge* two sets into a single set
(e.g., when we add edges so that now two nodes are part of the same component, when they were not before).

We need to implement it using actual data structures.

A First Attempt

Define an *array of sets* called Component, where Component$[u]$ is the set containing element u.
The size of Component is $n = |S|$.

MakeUnionFind(S): Setup Component and initialise Component$[u] = u$ for all $u \in S$.

Time: $O(n)$

Find(u): Simply return Component$[u]$

Time: $O(1)$

A First Attempt

Define an *array of sets* called Component, where Component$[u]$ is the set containing element u.
The size of Component is $n = |S|$.

Union(A, B): Update Component$[u]$ to $A \cup B$ for every element $u \in A$ or $u \in B$.

Time: $O(n)$

Suffices to show $T\left(\text{Find}(u)\right)$, and $T\left(\text{Union}(A, B)\right)$ are $O(\log n)$
and $T\left(\text{MakeUnionFind}(v)\right)$ is $O(m \log n)$

Let’s optimise a bit

Optimisation #1: For each set, keep a list of elements it contains.

Update takes time $O(|A| + |B|)$ rather than $O(n)$

Optimisation #2: Use the largest of A and B as the name for $A \cup B$ (keep the sizes in an array size$[\,\cdot\,]$).

Update takes time $O(|B|)$ assuming $|A| \geq |B|$.

Still, worst case for Union(A, B) is $O(n)$, when e.g., $|A| = \Omega(n)$ and $|B| = \Omega(n)$.

Let’s optimise a bit

Still, worst case for Union(A, B) is $O(n)$, when e.g., $|A| = \Omega(n)$ and $|B| = \Omega(n)$.

In a sequence of k Union(A, B) operations, how often does this really happen?

Intuition: There can only be a few sets of very large size, so all the other Union(A, B) operations should be pretty cheap.

Lemma: Sequence of Union(A, B) operations

Lemma: Any sequence of k Union(A, B) operations takes $O(k \log k)$ time.

Proof: Consider some element v for which Component$[v]$ gets updated throughout the sequence of k operations.

After k operations (…) elements still belong to their own singleton sets.

Lemma: Sequence of Union(A, B) operations

Lemma: Any sequence of k Union(A, B) operations takes $O(k \log k)$ time.

Proof: Consider some element v for which Component$[v]$ gets updated throughout the sequence of k operations.

After k operations at least $n - 2k$ elements still belong to their own singleton sets.

What is the largest size that the set in which v belongs can have during the sequence?

Lemma: Sequence of Union(A, B) operations

What is the largest size that the set in which v belongs can have during the sequence?

The maximum size it can reach is $2k$, since at least $n - 2k$ elements did not participate in the merge.

Every time Component$[v]$ is updated, the size of the set containing v at least doubles.

Note: It is important here to use our naming by the largest of the two sets that are merged.

How many updates to Component$[v]$? At most $\log_2(2k)$ updates.

Lemma: Sequence of Union(A, B) operations

Lemma: Any sequence of k Union(A, B) operations takes $O(k \log k)$ time.

Proof: At most $\log_2(2k)$ updates to Component$[v]$.

At most $2k$ elements participating in updates.

Time: $O(k \log k)$

Union-Find Operations

MakeUnionFind(S) creates a new Union-Find data structure where every element in S is a singleton set, i.e., $\{v_1\}, \{v_2,\}, \ldots \{v_k\}$ for $S = \{v_1, v_2, \ldots, v_k\}$ $T\left(\text{MakeUnionFind}(v)\right)$

Find(u) returns the name of the set containing element u. $T\left(\text{Find}(u)\right)$

Union(A, B) changes the Union-Find data structure by merging the sets A and B into a single set. $T\left(\text{Union}(A, B)\right)$

Suffices to show $T\left(\text{Find}(u)\right)$, and $T\left(\text{Union}(A, B)\right)$ are $O(\log n)$
and $T\left(\text{MakeUnionFind}(v)\right)$ is $O(m \log n)$

Kruskal’s algorithm
Running time

We first sort the edges in terms of non-decreasing cost.

$O(m \log m) = O(m \log n)$

We run MakeUnionFind(V) to initialise the components of $G = (V, T)$ to each contain one node. $T\left(\text{MakeUnionFind}(v)\right)$

When considering an edge $e = \{w, v\}$, we check if Find(w) = Find(v).

$\leq 2m \cdot T\left(\text{Find}(u)\right)$

If yes, we ignore the edge and continue with the next one.

If no, we add the edge to the spanning tree and run Union(Find(u), Find(v)) to merge the two components.

$\leq (n-1) \cdot T\left(\text{Union}(A, B)\right)$

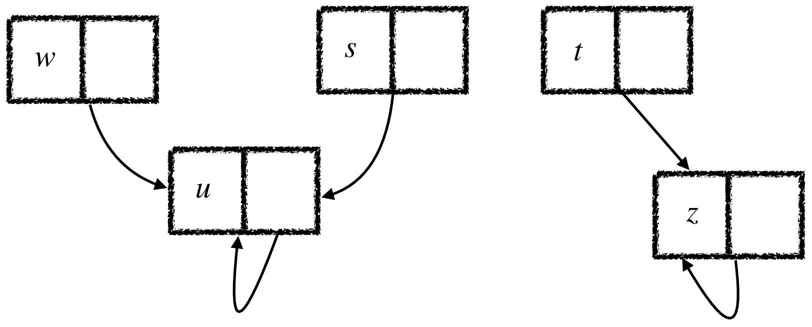
$\leq m \cdot T\left(\text{Union}(A, B)\right)$

$T(m\ \text{Union}(A, B)\ \text{operations}) = O(m \log m) = O(m \log n)$

A Better Implementation

Naming: Name a set S by the name of one of its elements v.

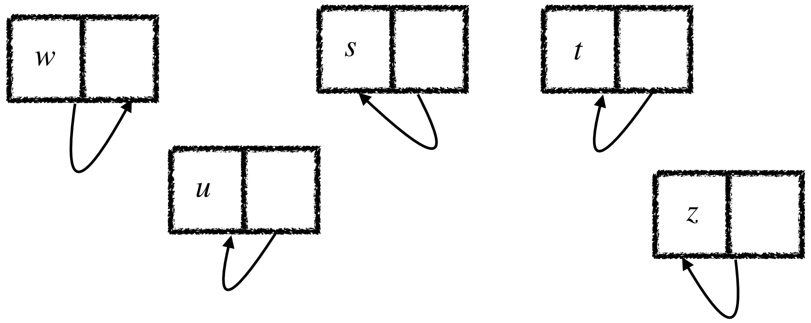
Pointers: Every element v points to some element u (possibly the same).



A Better Implementation

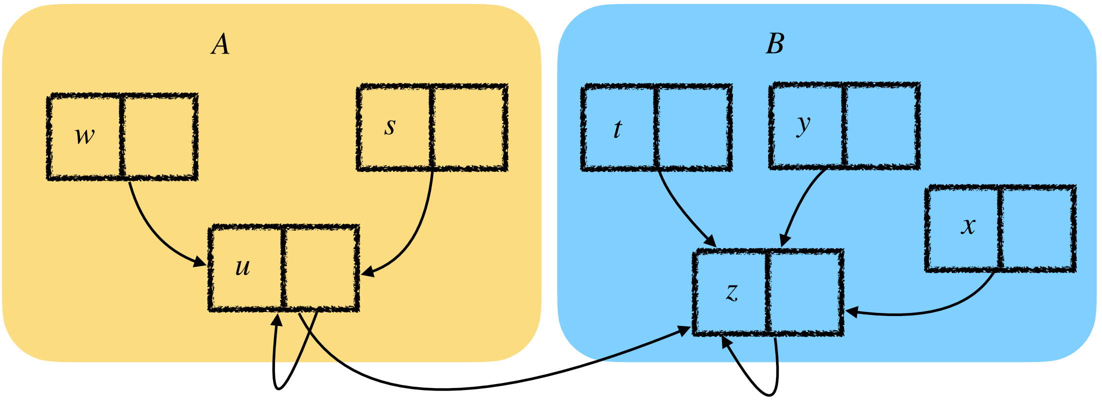
MakeUnionFind(S): Every element v points to itself.

$O(n)$ time



A Better Implementation

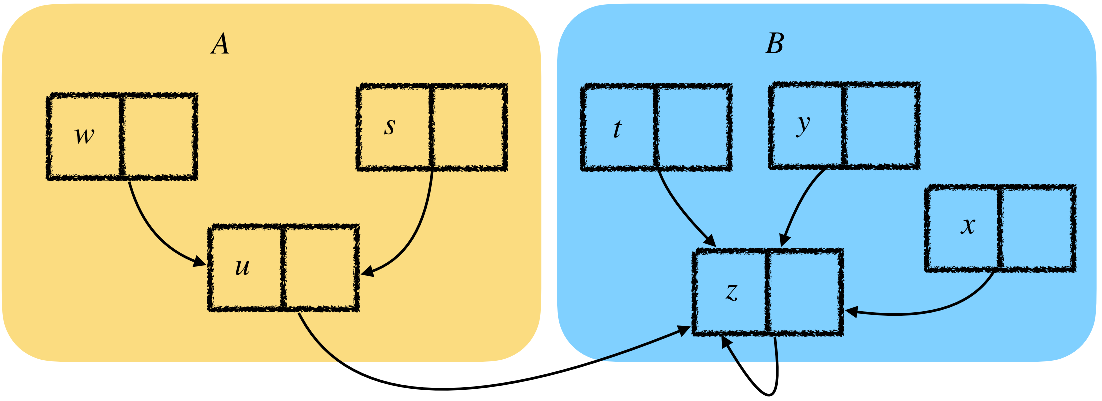
Union(A, B): Redirect the pointer of the smallest set to the largest set.



How much time needed for **Union**(A, B)? $O(1)$ time

A Better Implementation

Find(u): Follow the arrows to find the name of the set.



Bounding the time of Find(u)

How many times does an arrow get redirected?

i.e., how many times does a set chance its name?

Every time the set containing u changes name, it must be merged with a larger set, so its size *at least doubles* (by our naming convention).

Initially the set containing u has size 1.

In the end it has size at most n.

So, at most how many name changes? $O(\log n)$ changes

Union-Find Operations

MakeUnionFind(S) creates a new Union-Find data structure where every element in S is a singleton set, i.e., $\{v_1\}, \{v_2,\}, \ldots \{v_k\}$ for $S = \{v_1, v_2, \ldots, v_k\}$

$T\left(\text{MakeUnionFind}(v)\right) = O(n)$

Find(u) returns the name of the set containing element u.

$T\left(\text{Find}(u)\right) = O(\log n)$

Union(A, B) changes the Union-Find data structure by merging the sets A and B into a single set.

$T\left(\text{Union}(A, B)\right) = O(1)$

Suffices to show $T\left(\text{Find}(u)\right)$, and $T\left(\text{Union}(A, B)\right)$ are $O(\log n)$
and $T\left(\text{MakeUnionFind}(v)\right)$ is $O(m \log n)$

An even better implementation?

The pointer-based implementation can be made even better, using a similar argument as before, bounding the running time of a sequence of Find(u) operations rather than a single operation.

Details only if you are very interested: KT pp 197-199.

A Final Remark

In this course we have focused (and we will mostly focus) on algorithms.

But sometimes the right use of data structure can make our algorithm more efficient.

You can think of a data structure as a “part of the algorithm”, which can be abstracted from the more high-level ideas.

Data structures is a very big chapter in itself and an active area of research.