Disjoint sets and minimum spanning trees

Ashley Montanaro
ashley@cs.bris.ac.uk

Department of Computer Science, University of Bristol
Bristol, UK

11 November 2013

Introduction

- In this lecture we will start by discussing a data structure used for maintaining disjoint subsets of some bigger set.

Introduction

- In this lecture we will start by discussing a data structure used for maintaining disjoint subsets of some bigger set.
- This has a number of applications, including to maintaining connected components of a graph, and to finding minimum spanning trees in undirected graphs.

Introduction

- In this lecture we will start by discussing a data structure used for maintaining disjoint subsets of some bigger set.
- This has a number of applications, including to maintaining connected components of a graph, and to finding minimum spanning trees in undirected graphs.
- We will then discuss two algorithms for finding minimum spanning trees: an algorithm by Kruskal based on disjoint-set structures, and an algorithm by Prim which is similar to Dijkstra's algorithm.

Introduction

- In this lecture we will start by discussing a data structure used for maintaining disjoint subsets of some bigger set.
- This has a number of applications, including to maintaining connected components of a graph, and to finding minimum spanning trees in undirected graphs.
- We will then discuss two algorithms for finding minimum spanning trees: an algorithm by Kruskal based on disjoint-set structures, and an algorithm by Prim which is similar to Dijkstra's algorithm.
- In both cases, we will see that efficient implementations of data structures give us efficient algorithms.

Disjoint-set data structure

A disjoint-set data structure maintains a collection $\mathcal{S} = \{S_1, \ldots, S_k\}$ of disjoint subsets of some larger "universe" U.

Disjoint-set data structure

A disjoint-set data structure maintains a collection $\mathcal{S} = \{S_1, \dots, S_k\}$ of disjoint subsets of some larger "universe" U.

The data structure supports the following operations:

1. MakeSet(x): create a new set whose only member is x. As the sets are disjoint, we require that x is not contained in any of the other sets.

Disjoint-set data structure

A disjoint-set data structure maintains a collection $\mathcal{S} = \{S_1, \dots, S_k\}$ of disjoint subsets of some larger "universe" U.

The data structure supports the following operations:

1. MakeSet(x): create a new set whose only member is x. As the sets are disjoint, we require that x is not contained in any of the other sets.
2. Union(x, y): combine the sets containing x and y (call these S_x, S_y) to replace them with a new set $S_x \cup S_y$.

Disjoint-set data structure

A disjoint-set data structure maintains a collection $\mathcal{S} = \{S_1, \dots, S_k\}$ of disjoint subsets of some larger "universe" U.

The data structure supports the following operations:

1. MakeSet(x): create a new set whose only member is x. As the sets are disjoint, we require that x is not contained in any of the other sets.
2. Union(x, y): combine the sets containing x and y (call these S_x, S_y) to replace them with a new set $S_x \cup S_y$.
3. FindSet(x): returns the identity of the unique set containing x.

Disjoint-set data structure

A disjoint-set data structure maintains a collection $\mathcal{S} = \{S_1, \dots, S_k\}$ of disjoint subsets of some larger "universe" U.

The data structure supports the following operations:

1. MakeSet(x): create a new set whose only member is x. As the sets are disjoint, we require that x is not contained in any of the other sets.
2. Union(x, y): combine the sets containing x and y (call these S_x, S_y) to replace them with a new set $S_x \cup S_y$.
3. FindSet(x): returns the identity of the unique set containing x.

The identity of a set is just some unique identifier for that set – for example, the identity of one of the elements in the set.

Example

Operation	Returns	$\mathcal{S}$
(start)		(empty)

Example

Operation	Returns	$\mathcal{S}$
(start)		(empty)
MakeSet(a)		$\{a\}$

Example

Operation	Returns	$\mathcal{S}$
(start)		(empty)
MakeSet(a)		$\{a\}$
MakeSet(b)		$\{a\}, \{b\}$

Example

Operation	Returns	$\mathcal{S}$
(start)		(empty)
MakeSet(a)		$\{a\}$
MakeSet(b)		$\{a\}, \{b\}$
FindSet(b)	b	$\{a\}, \{b\}$

Ashley Montanaro
ashley@cs.bris.ac.uk

Example

Operation	Returns	$\mathcal{S}$
(start)		(empty)
MakeSet(a)		$\{a\}$
MakeSet(b)		$\{a\}, \{b\}$
FindSet(b)	b	$\{a\}, \{b\}$
Union(a, b)		$\{a, b\}$

Example

Operation	Returns	$\mathcal{S}$
(start)		(empty)
MakeSet(a)		$\{a\}$
MakeSet(b)		$\{a\}, \{b\}$
FindSet(b)	b	$\{a\}, \{b\}$
Union(a, b)		$\{a, b\}$
FindSet(b)	a	$\{a, b\}$

Example

Operation	Returns	$\mathcal{S}$
(start)		(empty)
MakeSet(a)		$\{a\}$
MakeSet(b)		$\{a\}, \{b\}$
FindSet(b)	b	$\{a\}, \{b\}$
Union(a, b)		$\{a, b\}$
FindSet(b)	a	$\{a, b\}$
FindSet(a)	a	$\{a, b\}$

Example

Operation	Returns	$\mathcal{S}$
(start)		(empty)
MakeSet(a)		$\{a\}$
MakeSet(b)		$\{a\}, \{b\}$
FindSet(b)	b	$\{a\}, \{b\}$
Union(a, b)		$\{a, b\}$
FindSet(b)	a	$\{a, b\}$
FindSet(a)	a	$\{a, b\}$
MakeSet(c)		$\{a, b\}, \{c\}$

Implementation

- A simple way to implement a disjoint-set data structure is as an array of linked lists.

Implementation

- A simple way to implement a disjoint-set data structure is as an array of linked lists.
- We have a linked list for each disjoint set. Each element *elem* in the list stores a pointer *elem.next* to the next element in the list, and the set element itself, *elem.data*.

Implementation

- A simple way to implement a disjoint-set data structure is as an array of linked lists.
- We have a linked list for each disjoint set. Each element *elem* in the list stores a pointer *elem.next* to the next element in the list, and the set element itself, *elem.data*.
- We also have an array A corresponding to the universe, with each entry in the array containing a pointer to the linked list corresponding to the set in which it occurs.

Implementation

- A simple way to implement a disjoint-set data structure is as an array of linked lists.
- We have a linked list for each disjoint set. Each element *elem* in the list stores a pointer *elem.next* to the next element in the list, and the set element itself, *elem.data*.
- We also have an array A corresponding to the universe, with each entry in the array containing a pointer to the linked list corresponding to the set in which it occurs.

Then to implement:

- MakeSet(x), we create a new list and set x's pointer to that list.

Implementation

- A simple way to implement a disjoint-set data structure is as an array of linked lists.
- We have a linked list for each disjoint set. Each element *elem* in the list stores a pointer *elem.next* to the next element in the list, and the set element itself, *elem.data*.
- We also have an array A corresponding to the universe, with each entry in the array containing a pointer to the linked list corresponding to the set in which it occurs.

Then to implement:

- MakeSet(x), we create a new list and set x's pointer to that list.
- FindSet(x), we return the first element in the list to which x points.

Implementation

- A simple way to implement a disjoint-set data structure is as an array of linked lists.
- We have a linked list for each disjoint set. Each element *elem* in the list stores a pointer *elem.next* to the next element in the list, and the set element itself, *elem.data*.
- We also have an array A corresponding to the universe, with each entry in the array containing a pointer to the linked list corresponding to the set in which it occurs.

Then to implement:

- MakeSet(x), we create a new list and set x's pointer to that list.
- FindSet(x), we return the first element in the list to which x points.
- Union(x, y), we append y's list to x's list and update the pointers of everything in y's list to point to to x's list.

Implementation

In more detail:

MakeSet(x)

1. $A[x] \leftarrow$ new linked list
2. $elem \leftarrow$ new list element
3. $elem.data \leftarrow x$
4. $A[x].head \leftarrow elem$
5. $A[x].tail \leftarrow elem$

Implementation

In more detail:

MakeSet(x)

1. $A[x] \leftarrow$ new linked list
2. $elem \leftarrow$ new list element
3. $elem.data \leftarrow x$
4. $A[x].head \leftarrow elem$
5. $A[x].tail \leftarrow elem$

FindSet(x)

1. return $A[x].head.data$

Implementation

Union(x, y)

1. $A[x].tail.next \leftarrow A[y].head$
2. $A[x].tail \leftarrow A[y].tail$

Implementation

Union(x, y)

1. $A[x].tail.next \leftarrow A[y].head$
2. $A[x].tail \leftarrow A[y].tail$
3. $elem \leftarrow A[y].head$
4. while $elem \neq nil$
5. $\quad A[elem.data] \leftarrow A[x]$
6. $\quad elem \leftarrow elem.next$

Example

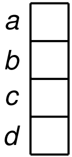
Imagine we have a universe $U = \{a, b, c, d\}$. The initial configuration of the array A (corresponding to $\mathcal{S} = \emptyset$) is

a	
b	
c	
d	

Ashley Montanaro
ashley@cs.bris.ac.uk

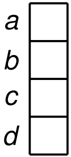
Example

Imagine we have a universe $U = \{a, b, c, d\}$. The initial configuration of the array A (corresponding to $\mathcal{S} = \emptyset$) is



Then the following sequence of updates occurs:

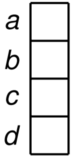
MakeSet(a)



Ashley Montanaro
ashley@cs.bris.ac.uk

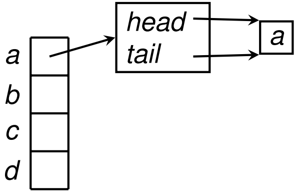
Example

Imagine we have a universe $U = \{a, b, c, d\}$. The initial configuration of the array A (corresponding to $\mathcal{S} = \emptyset$) is



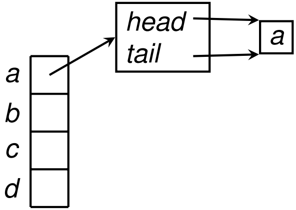
Then the following sequence of updates occurs:

MakeSet(a)



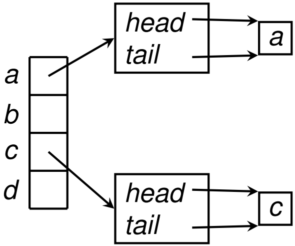
Example

MakeSet(c)



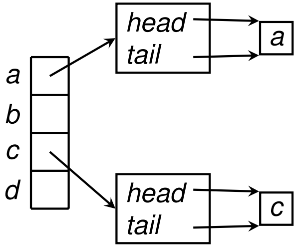
Example

MakeSet(c)



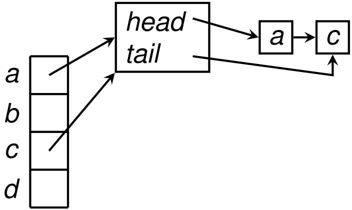
Example

Union(a, c)



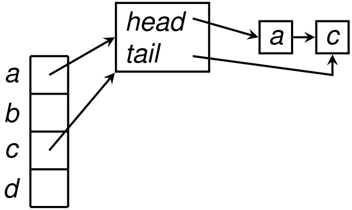
Example

Union(a, c)



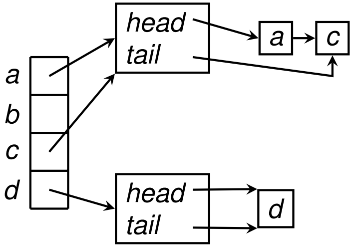
Example

MakeSet(d)



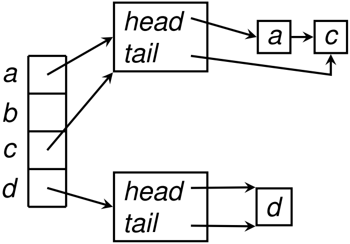
Example

MakeSet(d)



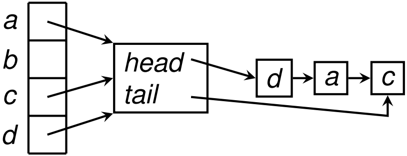
Example

Union(d, c)



Example

Union(d, c)



Improvement: the weighted-union heuristic

- MakeSet and FindSet take time $O(1)$ but Union might take time $\Theta(n)$ for a universe of size n.

Improvement: the weighted-union heuristic

- MakeSet and FindSet take time $O(1)$ but Union might take time $\Theta(n)$ for a universe of size n.
- Union(x, y) needs to update tail pointers in lists (constant time) but also the information of every element in y's list.

Improvement: the weighted-union heuristic

- MakeSet and FindSet take time $O(1)$ but Union might take time $\Theta(n)$ for a universe of size n.
- Union(x, y) needs to update tail pointers in lists (constant time) but also the information of every element in y's list.
- So the Union operation is slow when y's list is long and x's is short.

Improvement: the weighted-union heuristic

- MakeSet and FindSet take time $O(1)$ but Union might take time $\Theta(n)$ for a universe of size n.
- Union(x, y) needs to update tail pointers in lists (constant time) but also the information of every element in y's list.
- So the Union operation is slow when y's list is long and x's is short.
- Heuristic: always append the shorter list to the longer list.

Improvement: the weighted-union heuristic

- MakeSet and FindSet take time $O(1)$ but Union might take time $\Theta(n)$ for a universe of size n.
- Union(x, y) needs to update tail pointers in lists (constant time) but also the information of every element in y's list.
- So the Union operation is slow when y's list is long and x's is short.
- Heuristic: always append the shorter list to the longer list.
- Might still take time $\Theta(n)$ in the worst case (if both lists have the same size), but we have the following amortised analysis:

Claim

Using the linked-list representation and the above heuristic, a sequence of m MakeSet, FindSet and Union operations, n of which are MakeSet operations, uses time $O(m + n \log n)$.

Improvement: the weighted-union heuristic

Claim

Using the linked-list representation and the above heuristic, a sequence of m MakeSet, FindSet and Union operations, n of which are MakeSet operations, uses time $O(m + n \log n)$.

Proof

- MakeSet and FindSet take time $O(1)$ each, and there can be at most $n - 1$ non-trivial Union operations.

Improvement: the weighted-union heuristic

Claim

Using the linked-list representation and the above heuristic, a sequence of m MakeSet, FindSet and Union operations, n of which are MakeSet operations, uses time $O(m + n \log n)$.

Proof

- MakeSet and FindSet take time $O(1)$ each, and there can be at most $n - 1$ non-trivial Union operations.
- At each Union operation, an element's information is only updated when it was in the smaller set of the two sets.

Improvement: the weighted-union heuristic

Claim

Using the linked-list representation and the above heuristic, a sequence of m MakeSet, FindSet and Union operations, n of which are MakeSet operations, uses time $O(m + n \log n)$.

Proof

- MakeSet and FindSet take time $O(1)$ each, and there can be at most $n - 1$ non-trivial Union operations.
- At each Union operation, an element's information is only updated when it was in the smaller set of the two sets.
- So, the first time it is updated, the resulting set must have size at least 2. The second time, size at least 4. The k'th time, size at least 2^k.

Improvement: the weighted-union heuristic

Claim

Using the linked-list representation and the above heuristic, a sequence of m MakeSet, FindSet and Union operations, n of which are MakeSet operations, uses time $O(m + n \log n)$.

Proof

- MakeSet and FindSet take time $O(1)$ each, and there can be at most $n - 1$ non-trivial Union operations.
- At each Union operation, an element's information is only updated when it was in the smaller set of the two sets.
- So, the first time it is updated, the resulting set must have size at least 2. The second time, size at least 4. The k'th time, size at least 2^k.
- So each element's information is only updated at most $O(\log n)$ times.

Improvement: the weighted-union heuristic

Claim

Using the linked-list representation and the above heuristic, a sequence of m MakeSet, FindSet and Union operations, n of which are MakeSet operations, uses time $O(m + n \log n)$.

Proof

- MakeSet and FindSet take time $O(1)$ each, and there can be at most $n - 1$ non-trivial Union operations.
- At each Union operation, an element's information is only updated when it was in the smaller set of the two sets.
- So, the first time it is updated, the resulting set must have size at least 2. The second time, size at least 4. The k'th time, size at least 2^k.
- So each element's information is only updated at most $O(\log n)$ times.
- So $O(n \log n)$ updates are made in total. All other operations use time $O(1)$, so the total runtime is $O(m + n \log n)$. □

Improvements

- Another way to implement a disjoint-set structure is via a disjoint-set forest (CLRS §21.3). This structure is based on replacing the linked lists with trees.

Improvements

- Another way to implement a disjoint-set structure is via a disjoint-set forest (CLRS §21.3). This structure is based on replacing the linked lists with trees.
- One can show that using a disjoint-set forest, along with some optimisations, a sequence of m operations with n MakeSet operations runs in time $O(m\,\alpha(n))$, where $\alpha(n)$ is an extremely slowly growing function which satisfies $\alpha(n) \leq 4$ for any $n \leq 10^{80}$.

Improvements

- Another way to implement a disjoint-set structure is via a disjoint-set forest (CLRS §21.3). This structure is based on replacing the linked lists with trees.
- One can show that using a disjoint-set forest, along with some optimisations, a sequence of m operations with n MakeSet operations runs in time $O(m\,\alpha(n))$, where $\alpha(n)$ is an extremely slowly growing function which satisfies $\alpha(n) \leq 4$ for any $n \leq 10^{80}$.
- Disjoint-set forests were introduced in 1964 by Galler and Fischer but this bound was not proven until 1975 by Tarjan.

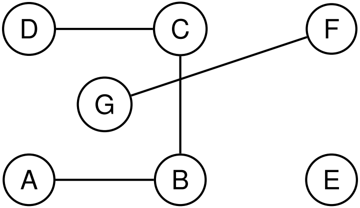
Improvements

- Another way to implement a disjoint-set structure is via a disjoint-set forest (CLRS §21.3). This structure is based on replacing the linked lists with trees.
- One can show that using a disjoint-set forest, along with some optimisations, a sequence of m operations with n MakeSet operations runs in time $O(m\,\alpha(n))$, where $\alpha(n)$ is an extremely slowly growing function which satisfies $\alpha(n) \leq 4$ for any $n \leq 10^{80}$.
- Disjoint-set forests were introduced in 1964 by Galler and Fischer but this bound was not proven until 1975 by Tarjan.
- Amazingly, it is known that this runtime bound cannot be replaced with a bound $O(m)$.

Application: computing connected components

A simple application of the disjoint-set data structure is computing connected components of an undirected graph.

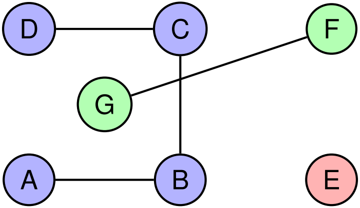
For example:



Application: computing connected components

A simple application of the disjoint-set data structure is computing connected components of an undirected graph.

For example:



Application: computing connected components

ConnectedComponents(G)

1. for each vertex $v \in G$: MakeSet(v)
2. for each edge $u \leftrightarrow v$ in arbitrary order
3.   if FindSet(u) $\neq$ FindSet(v)
4.   Union(u, v)

Application: computing connected components

ConnectedComponents(G)

1. for each vertex $v \in G$: MakeSet(v)
2. for each edge $u \leftrightarrow v$ in arbitrary order
3.   if FindSet(u) $\neq$ FindSet(v)
4.   Union(u, v)

- Time complexity: $O(E + V \log V)$ if implemented using linked lists, $O(E\,\alpha(V))$ if implemented using an optimised disjoint-set forest.

Application: computing connected components

ConnectedComponents(G)

1. for each vertex $v \in G$: MakeSet(v)
2. for each edge $u \leftrightarrow v$ in arbitrary order
3. if FindSet(u) $\neq$ FindSet(v)
4. Union(u, v)

- Time complexity: $O(E + V \log V)$ if implemented using linked lists, $O(E\,\alpha(V))$ if implemented using an optimised disjoint-set forest.
- After ConnectedComponents completes, FindSet can be used to determine whether two vertices are in the same component, in time $O(1)$.

Application: computing connected components

ConnectedComponents(G)

1. for each vertex $v \in G$: MakeSet(v)
2. for each edge $u \leftrightarrow v$ in arbitrary order
3. if FindSet(u) $\neq$ FindSet(v)
4. Union(u, v)

- Time complexity: $O(E + V \log V)$ if implemented using linked lists, $O(E\,\alpha(V))$ if implemented using an optimised disjoint-set forest.
- After ConnectedComponents completes, FindSet can be used to determine whether two vertices are in the same component, in time $O(1)$.
- This task could also be achieved using breadth-first search, but using disjoint sets allows searching and adding vertices to be carried out more efficiently in future.

Minimum spanning trees

Given a connected, undirected weighted graph G, a subgraph T is a spanning tree if:

- T is a tree (i.e. does not contain any cycles)
- Every vertex in G appears in T.

Minimum spanning trees

Given a connected, undirected weighted graph G, a subgraph T is a spanning tree if:

- T is a tree (i.e. does not contain any cycles)
- Every vertex in G appears in T.

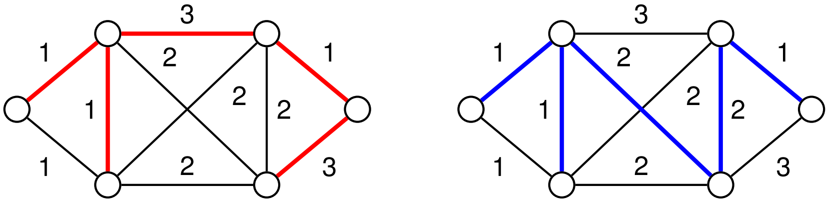
T is a minimum spanning tree (MST) if the sum of the weights of edges of T is minimal among all spanning trees of G.

Minimum spanning trees

Given a connected, undirected weighted graph G, a subgraph T is a spanning tree if:

- T is a tree (i.e. does not contain any cycles)
- Every vertex in G appears in T.

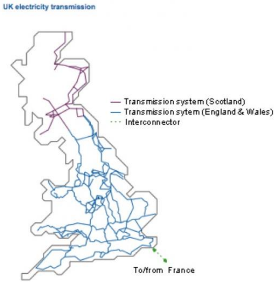
T is a minimum spanning tree (MST) if the sum of the weights of edges of T is minimal among all spanning trees of G.

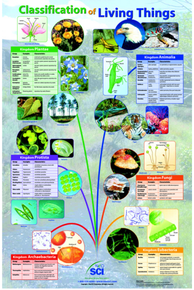


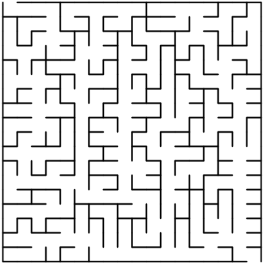
A spanning tree and a minimum spanning tree of the same graph.

MSTs: applications

- Telecommunications and utilities
- Cluster analysis
- Taxonomy
- Handwriting recognition
- Maze generation
- ...







Pics: nationalgrid.com, connecticutvalleybiological.com, Wikipedia

Ashley Montanaro
ashley@cs.bris.ac.uk

University of BRISTOL

A generic approach to MSTs

The two algorithms we will discuss for finding MSTs are both based on the following basic idea:

1. Maintain a forest (i.e. a collection of trees) F which is a subset of some minimum spanning tree.

A generic approach to MSTs

The two algorithms we will discuss for finding MSTs are both based on the following basic idea:

1. Maintain a forest (i.e. a collection of trees) F which is a subset of some minimum spanning tree.
2. At each step, add a new edge to F, maintaining the above property.

A generic approach to MSTs

The two algorithms we will discuss for finding MSTs are both based on the following basic idea:

1. Maintain a forest (i.e. a collection of trees) F which is a subset of some minimum spanning tree.
2. At each step, add a new edge to F, maintaining the above property.
3. Repeat until F is a minimum spanning tree.

A generic approach to MSTs

The two algorithms we will discuss for finding MSTs are both based on the following basic idea:

1. Maintain a forest (i.e. a collection of trees) F which is a subset of some minimum spanning tree.
2. At each step, add a new edge to F, maintaining the above property.
3. Repeat until F is a minimum spanning tree.

This approach of making a "locally optimal" choice of an edge at each step makes them greedy algorithms.

A generic approach to MSTs

The two algorithms we will discuss for finding MSTs are both based on the following basic idea:

1. Maintain a forest (i.e. a collection of trees) F which is a subset of some minimum spanning tree.
2. At each step, add a new edge to F, maintaining the above property.
3. Repeat until F is a minimum spanning tree.

This approach of making a “locally optimal” choice of an edge at each step makes them greedy algorithms.

We will discuss:

- Kruskal’s algorithm, which is based on a disjoint-set data structure.
- Prim’s algorithm, which is based on a priority queue.

A generic approach to MSTs

The two algorithms we will discuss for finding MSTs are both based on the following basic idea:

1. Maintain a forest (i.e. a collection of trees) F which is a subset of some minimum spanning tree.
2. At each step, add a new edge to F, maintaining the above property.
3. Repeat until F is a minimum spanning tree.

This approach of making a "locally optimal" choice of an edge at each step makes them greedy algorithms.

We will discuss:

- Kruskal's algorithm, which is based on a disjoint-set data structure.
- Prim's algorithm, which is based on a priority queue.

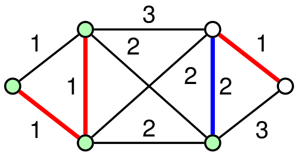
The algorithms make different choices for which new edge to add at each step.

How to choose new edges?

Cut property

Let X be a subset of some MST T. Let S be a subset of the vertices of G such that X does not contain any edges with exactly one endpoint in S. Let e be a lightest edge in G that has exactly one endpoint in S.
Then $X \cup \{e\}$ is a subset of an MST.

For example:



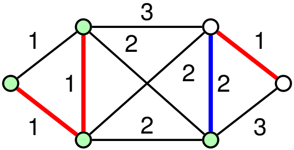
Ashley Montanaro
ashley@cs.bris.ac.uk

How to choose new edges?

Cut property

Let X be a subset of some MST T. Let S be a subset of the vertices of G such that X does not contain any edges with exactly one endpoint in S. Let e be a lightest edge in G that has exactly one endpoint in S.
Then $X \cup \{e\}$ is a subset of an MST.

For example:



Proof

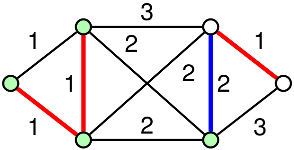
- If $e \in T$, the claim is obviously true, so assume $e \notin T$.

How to choose new edges?

Cut property

Let X be a subset of some MST T. Let S be a subset of the vertices of G such that X does not contain any edges with exactly one endpoint in S. Let e be a lightest edge in G that has exactly one endpoint in S.
Then $X \cup \{e\}$ is a subset of an MST.

For example:



Proof

- If $e \in T$, the claim is obviously true, so assume $e \notin T$.
- As T is a spanning tree, there must exist a path p in T between the endpoints of e, where p contains an edge e' with one endpoint in S. ...

Ashley Montanaro
ashley@cs.bris.ac.uk

How to choose new edges?

Cut property

Let X be a subset of some MST T. Let S be a subset of the vertices of G such that X does not contain any edges with exactly one endpoint in S. Let e be a lightest edge in G that has exactly one endpoint in S.
Then $X \cup \{e\}$ is a subset of an MST.

Proof

- Exercise: For any edge e' on the path p, if we replace e' with e in T, the resulting set T' is still a spanning tree.

How to choose new edges?

Cut property

Let X be a subset of some MST T. Let S be a subset of the vertices of G such that X does not contain any edges with exactly one endpoint in S. Let e be a lightest edge in G that has exactly one endpoint in S.
Then $X \cup \{e\}$ is a subset of an MST.

Proof

- Exercise: For any edge e' on the path p, if we replace e' with e in T, the resulting set T' is still a spanning tree.
- Further, the total weight of T' is

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

How to choose new edges?

Cut property

Let X be a subset of some MST T. Let S be a subset of the vertices of G such that X does not contain any edges with exactly one endpoint in S. Let e be a lightest edge in G that has exactly one endpoint in S.
Then $X \cup \{e\}$ is a subset of an MST.

Proof

- Exercise: For any edge e' on the path p, if we replace e' with e in T, the resulting set T' is still a spanning tree.
- Further, the total weight of T' is

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

- As e is the lightest edge with one endpoint in S, $w(e) \leq w(e')$.

Ashley Montanaro
ashley@cs.bris.ac.uk

How to choose new edges?

Cut property

Let X be a subset of some MST T. Let S be a subset of the vertices of G such that X does not contain any edges with exactly one endpoint in S. Let e be a lightest edge in G that has exactly one endpoint in S.
Then $X \cup \{e\}$ is a subset of an MST.

Proof

- Exercise: For any edge e' on the path p, if we replace e' with e in T, the resulting set T' is still a spanning tree.
- Further, the total weight of T' is

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

- As e is the lightest edge with one endpoint in S, $w(e) \leq w(e')$.
- Hence $\text{weight}(T') \leq \text{weight}(T)$, so T' is also an MST. □

Kruskal's algorithm

- The algorithm has a similar flow to the algorithm for computing connected components.
- It maintains a forest F, initially consisting of unconnected individual vertices, and a disjoint-set data structure.

Kruskal's algorithm

- The algorithm has a similar flow to the algorithm for computing connected components.
- It maintains a forest F, initially consisting of unconnected individual vertices, and a disjoint-set data structure.

Kruskal(G)

1. for each vertex $v \in G$: MakeSet(v)

Ashley Montanaro
ashley@cs.bris.ac.uk

Kruskal's algorithm

- The algorithm has a similar flow to the algorithm for computing connected components.
- It maintains a forest F, initially consisting of unconnected individual vertices, and a disjoint-set data structure.

Kruskal(G)

1. for each vertex $v \in G$: MakeSet(v)
2. sort the edges of G into non-decreasing order by weight

Kruskal's algorithm

- The algorithm has a similar flow to the algorithm for computing connected components.
- It maintains a forest F, initially consisting of unconnected individual vertices, and a disjoint-set data structure.

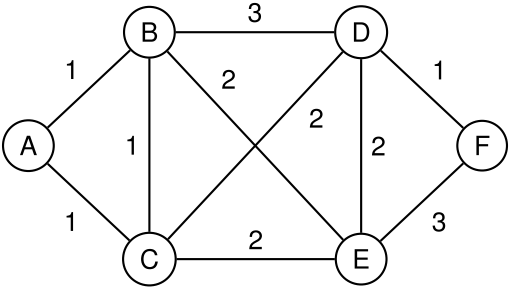
Kruskal(G)

1. for each vertex $v \in G$: MakeSet(v)
2. sort the edges of G into non-decreasing order by weight
3. for each edge $u \leftrightarrow v$ in order
4.   if FindSet(u) $\neq$ FindSet(v)
5. $F \leftarrow F \cup \{u \leftrightarrow v\}$
6.   Union(u, v)

Informally: "add the lightest edge between two components of F".

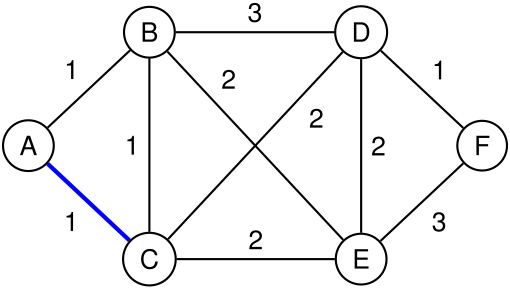
Example

We use Kruskal's algorithm to find an MST in the following graph.



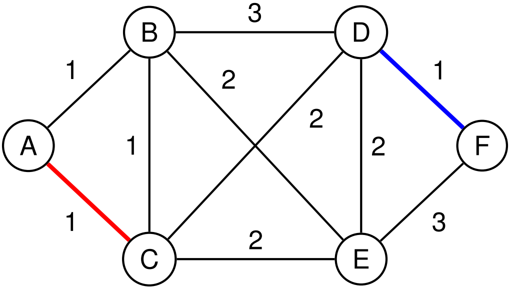
Example

First an arbitrary edge with weight 1 is picked:



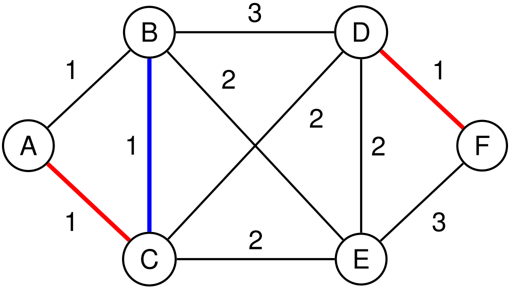
Example

Then any other edge with weight 1:



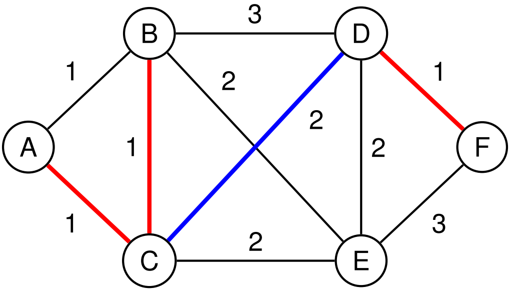
Example

Then any other edge with weight 1:



Example

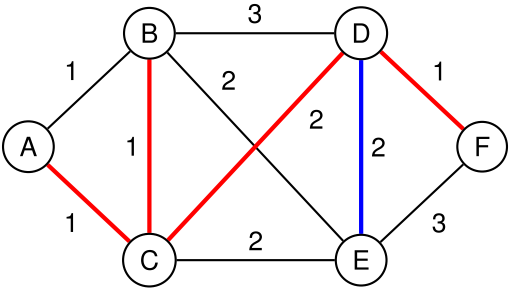
The final edge with weight 1 cannot be picked because A and B are in the same component, so one of the edges with weight 2 is chosen:



Ashley Montanaro
ashley@cs.bris.ac.uk

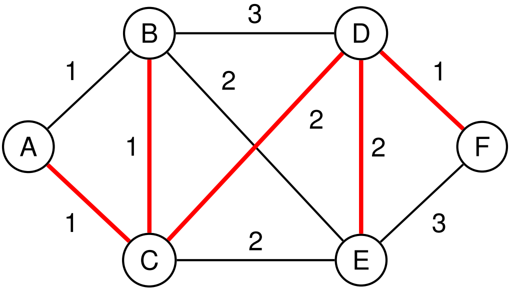
Example

Finally, one of the other edges with weight 2 is chosen and the MST is complete.



Example

Finally, one of the other edges with weight 2 is chosen and the MST is complete.



Proof of correctness

Kruskal(G)

1. for each vertex $v \in G$: MakeSet(v)
2. sort the edges of G into non-decreasing order by weight
3. for each edge $u \leftrightarrow v$ in order
4. if FindSet(u) $\neq$ FindSet(v)
5. $F \leftarrow F \cup \{u \leftrightarrow v\}$
6. Union(u, v)

Proof of correctness

- Whenever FindSet(u) $\neq$ FindSet(v), the edge $u \leftrightarrow v$ connects two trees T_1, T_2. Set $S = T_1$ in the cut property.

Proof of correctness

Kruskal(G)

1. for each vertex $v \in G$: MakeSet(v)
2. sort the edges of G into non-decreasing order by weight
3. for each edge $u \leftrightarrow v$ in order
4. if FindSet(u) $\neq$ FindSet(v)
5. $F \leftarrow F \cup \{u \leftrightarrow v\}$
6. Union(u, v)

Proof of correctness

- Whenever FindSet(u) $\neq$ FindSet(v), the edge $u \leftrightarrow v$ connects two trees T_1, T_2. Set $S = T_1$ in the cut property.
- This edge is a lightest edge with one endpoint in S.

Proof of correctness

Kruskal(G)

1. for each vertex $v \in G$: MakeSet(v)
2. sort the edges of G into non-decreasing order by weight
3. for each edge $u \leftrightarrow v$ in order
4. if FindSet(u) $\neq$ FindSet(v)
5. $F \leftarrow F \cup \{u \leftrightarrow v\}$
6. Union(u, v)

Proof of correctness

- Whenever FindSet(u) $\neq$ FindSet(v), the edge $u \leftrightarrow v$ connects two trees T_1, T_2. Set $S = T_1$ in the cut property.
- This edge is a lightest edge with one endpoint in S.
- So, by the cut property, $F \cup \{u \leftrightarrow v\}$ is a subset of an MST. $\square$

Complexity analysis of Kruskal's algorithm

Kruskal(G)

1. for each vertex $v \in G$: MakeSet(v)
2. sort the edges of G into non-decreasing order by weight
3. for each edge $u \leftrightarrow v$ in order
4. if FindSet(u) $\neq$ FindSet(v)
5. $F \leftarrow F \cup \{u \leftrightarrow v\}$
6. Union(u, v)

- V MakeSet operations
- Time $O(E \log E)$ to sort edges
- $O(E)$ FindSet and Union operations

Complexity analysis of Kruskal's algorithm

Kruskal(G)

1. for each vertex $v \in G$: MakeSet(v)
2. sort the edges of G into non-decreasing order by weight
3. for each edge $u \leftrightarrow v$ in order
4. if FindSet(u) $\neq$ FindSet(v)
5. $F \leftarrow F \cup \{u \leftrightarrow v\}$
6. Union(u, v)

- V MakeSet operations
- Time $O(E \log E)$ to sort edges
- $O(E)$ FindSet and Union operations
- So, using a disjoint-set structure implemented using an array of linked lists, we get an overall runtime of $O(E \log E)$.

Complexity analysis of Kruskal's algorithm

Kruskal(G)

1. for each vertex $v \in G$: MakeSet(v)
2. sort the edges of G into non-decreasing order by weight
3. for each edge $u \leftrightarrow v$ in order
4.   if FindSet(u) $\neq$ FindSet(v)
5. $F \leftarrow F \cup \{u \leftrightarrow v\}$
6.   Union(u, v)

- V MakeSet operations
- Time $O(E \log E)$ to sort edges
- $O(E)$ FindSet and Union operations
- So, using a disjoint-set structure implemented using an array of linked lists, we get an overall runtime of $O(E \log E)$.
- If the edges are already sorted, and we use an optimised disjoint-set forest, we can achieve $O(E\,\alpha(V))$.

Prim's algorithm

- Kruskal's algorithm maintains a forest F and uses the rule: "add the lightest edge between two components of F" at each step.
- A different approach is used by Prim's algorithm: "maintain a connected tree T and extend T with the lightest possible edge".

Prim's algorithm

- Kruskal's algorithm maintains a forest F and uses the rule: "add the lightest edge between two components of F" at each step.
- A different approach is used by Prim's algorithm: "maintain a connected tree T and extend T with the lightest possible edge".
- Prim's algorithm is based on the use of a priority queue Q.
- The flow of the algorithm is almost exactly the same as Dijkstra's algorithm; the only difference is the choice of key for the queue.
- For each vertex v, $v.key$ is the weight of the lightest edge connecting v to T.

Prim's algorithm

Prim(G)

1. for each vertex $v \in G$: $v.key \leftarrow \infty$, $v.\pi \leftarrow$ nil

Prim's algorithm

Prim(G)

1. for each vertex $v \in G$: $v.key \leftarrow \infty$, $v.\pi \leftarrow$ nil
2. choose an arbitrary vertex r
3. $r.key \leftarrow 0$

Prim's algorithm

Prim(G)

1. for each vertex $v \in G$: $v.key \leftarrow \infty$, $v.\pi \leftarrow$ nil
2. choose an arbitrary vertex r
3. $r.key \leftarrow 0$
4. add every vertex in G to Q

Prim's algorithm

Prim(G)

- for each vertex $v \in G$: $v.key \leftarrow \infty$, $v.\pi \leftarrow$ nil
- choose an arbitrary vertex r
- $r.key \leftarrow 0$
- add every vertex in G to Q
- while Q not empty
  - $u \leftarrow$ ExtractMin(Q)
  - for each vertex v such that $u \leftrightarrow v$
    - if $v \in Q$ and $w(u, v) < v.key$
      - $v.\pi \leftarrow u$
      - DecreaseKey($v, w(u, v)$)

Prim's algorithm

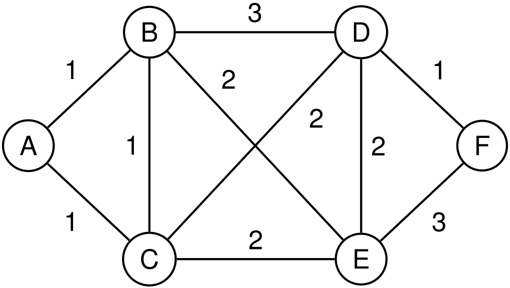
Prim(G)

for each vertex $v \in G$: $v.key \leftarrow \infty$, $v.\pi \leftarrow$ nil
choose an arbitrary vertex r
$r.key \leftarrow 0$
add every vertex in G to Q
while Q not empty
    $u \leftarrow$ ExtractMin(Q)
    for each vertex v such that $u \leftrightarrow v$
        if $v \in Q$ and $w(u, v) < v.key$
            $v.\pi \leftarrow u$
            DecreaseKey($v, w(u, v)$)

The algorithm can be seen as maintaining a growing tree, defined by the predecessor information $v.\pi$, to which each vertex extracted from the queue is added.

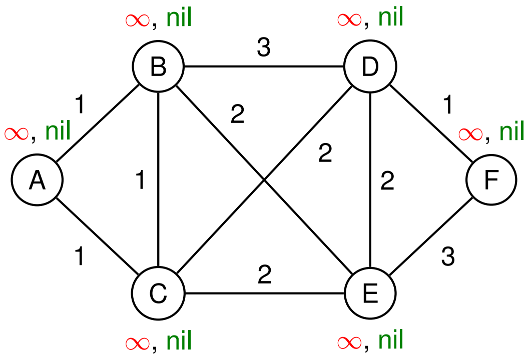
Example

We use Prim's algorithm to find an MST in the following graph.



Example

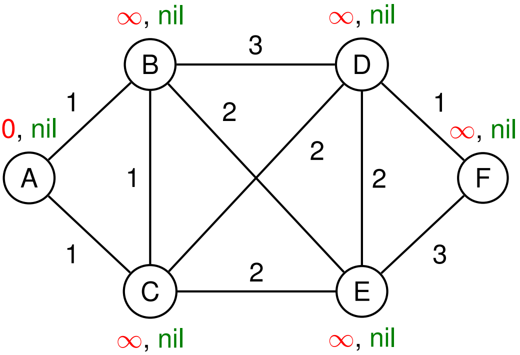
The state at the start of the algorithm:



- In the above diagram, the red text is the key values of the vertices (i.e. $v.key$) and the green text is the predecessor vertex (i.e. $v.\pi$).

Example

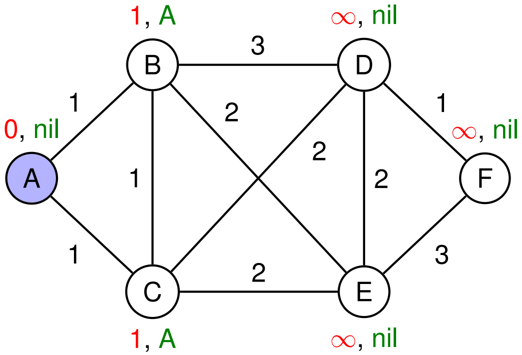
First the algorithm picks an arbitrary starting vertex r and updates its key value to 0.



- Here we arbitrarily choose A as our starting vertex.

Example

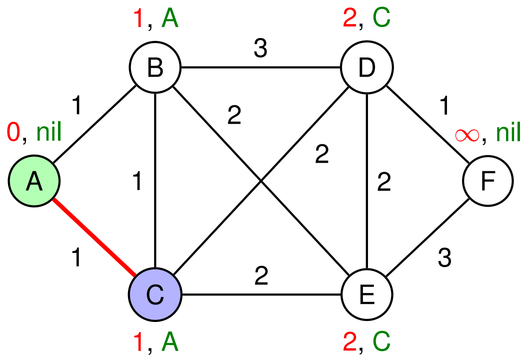
Then A is extracted from the queue, and the keys of its neighbours are updated:



- Vertex colours: Blue: current vertex, green: vertices added to tree.

Example

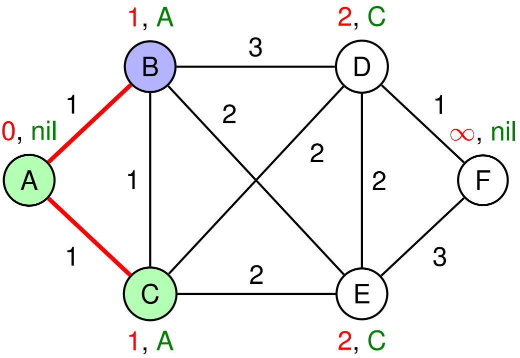
Then either B or C is extracted from the queue (here, we pick C):



- The red line shows the growing tree.

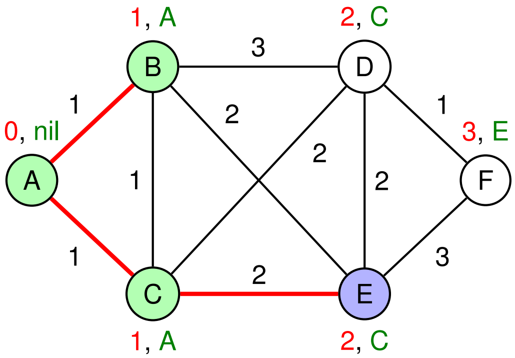
Example

Then B is extracted from the queue:



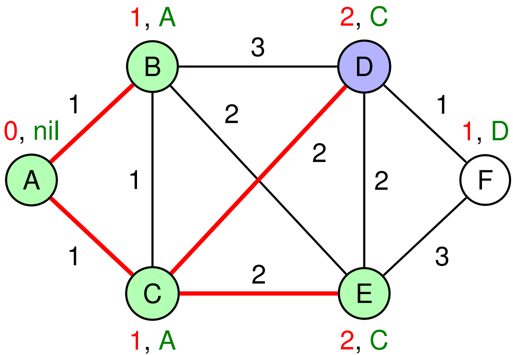
Example

Then either D or E is extracted from the queue (here, we pick E):



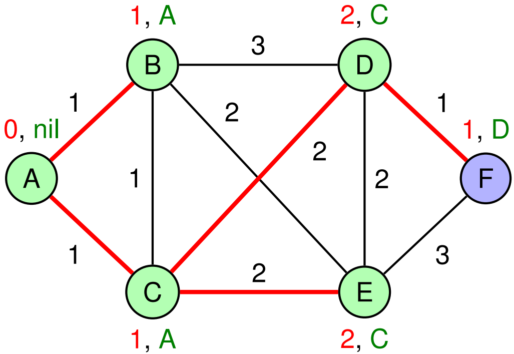
Example

Then D is extracted from the queue:



Example

Finally F is extracted from the queue and the algorithm is complete:



Correctness and complexity

Proof of correctness

- Prim's algorithm maintains a single, growing tree T starting with r, and to which each vertex removed from Q is appended.

Correctness and complexity

Proof of correctness

- Prim's algorithm maintains a single, growing tree T starting with r, and to which each vertex removed from Q is appended.
- Each vertex added to T is a vertex connected to T by a lightest edge.

Correctness and complexity

Proof of correctness

- Prim's algorithm maintains a single, growing tree T starting with r, and to which each vertex removed from Q is appended.
- Each vertex added to T is a vertex connected to T by a lightest edge.
- The cut property is therefore satisfied (taking $S = T$), so when the algorithm completes, T is an MST.

Correctness and complexity

Proof of correctness

- Prim's algorithm maintains a single, growing tree T starting with r, and to which each vertex removed from Q is appended.
- Each vertex added to T is a vertex connected to T by a lightest edge.
- The cut property is therefore satisfied (taking $S = T$), so when the algorithm completes, T is an MST.
- The predecessor information $v.\pi$ can be used to output T. $\square$

Correctness and complexity

Proof of correctness

- Prim's algorithm maintains a single, growing tree T starting with r, and to which each vertex removed from Q is appended.
- Each vertex added to T is a vertex connected to T by a lightest edge.
- The cut property is therefore satisfied (taking $S = T$), so when the algorithm completes, T is an MST.
- The predecessor information $v.\pi$ can be used to output T. $\square$

Complexity analysis:

- The complexity is asymptotically the same as Dijkstra's algorithm.
- If the priority queue is implemented using a binary heap, we get an overall bound of $O(E \log V)$; if it is implemented using a Fibonacci heap, we get $O(E + V \log V)$.

Comparison of MST algorithms

To summarise the two MST algorithms discussed:

Algorithm	Underlying structure	Runtime
Kruskal	Disjoint-set	$O(E \log E)$ (linked lists) $O(E\,\alpha(V))$ (disjoint-set forest, edges already sorted)
Prim	Priority queue	$O(E \log V)$ (binary heap) $O(E + V \log V)$ (Fibonacci heap)

Comparison of MST algorithms

To summarise the two MST algorithms discussed:

Algorithm	Underlying structure	Runtime
Kruskal	Disjoint-set	$O(E \log E)$ (linked lists) $O(E\,\alpha(V))$ (disjoint-set forest, edges already sorted)
Prim	Priority queue	$O(E \log V)$ (binary heap) $O(E + V \log V)$ (Fibonacci heap)

So which algorithm to use?

- If the edges are not already sorted, and cannot be sorted in linear time, the most efficient algorithm in theory is Prim with a Fibonacci heap (but in practice, either Kruskal with a disjoint-set forest or Prim with a binary heap is likely to be quicker).
- If the edges are already sorted, or can be sorted in time $O(E)$, then Kruskal with an optimised disjoint-set forest is quickest.

Summary

- A disjoint-set structure provides an efficient way to store a collection of disjoint subsets of some universe, and can be implemented using an array of linked lists.
- Disjoint-set structures can be used to maintain a set of connected components of a graph, and also to find minimum spanning trees using Kruskal's algorithm.
- An alternative way of finding minimum spanning trees is Prim's algorithm, which is based on the use of a priority queue and is similar to Dijkstra's algorithm.
- Both algorithms are greedy algorithms which rely on the optimal substructure property of minimum spanning trees.

Further Reading

- Introduction to Algorithms
  T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein.
  MIT Press/McGraw-Hill, ISBN: 0-262-03293-7.
  - Chapter 21 – Data Structures for Disjoint Sets
    (NB: presented slightly differently to lecture)
  - Chapter 23 – Minimum Spanning Trees

- Algorithms
  S. Dasgupta, C. H. Papadimitriou and U. V. Vazirani
  `http://www.cse.ucsd.edu/users/dasgupta/mcgrawhill/`
  - Chapter 5 – Greedy algorithms

- Algorithms lecture notes, University of Illinois
  Jeff Erickson
  `http://www.cs.uiuc.edu/~jeffe/teaching/algorithms/`
  - Lecture 18 – Minimum spanning trees

Biographical notes

Joseph B. Kruskal, Jr. (1928–2010)

- Kruskal was an American mathematician and computer scientist who did important work in statistics and combinatorics, as well as computer science.
- His algorithm was discovered in 1956 while at Princeton University; he spent most of his later career at Bell Labs.
- His two brothers William and Martin were also famous mathematicians.

Pic: ams.org

Biographical notes

Robert C. Prim III (1921–)

- Prim is an American mathematician and computer scientist, who developed his algorithm while working at Bell Labs in 1957, where he was later director of mathematics research.
- Prim's algorithm was originally and independently discovered in 1930 by Jarník. It was later rediscovered again by Edsger Dijkstra in 1959.

Pic: ams.org