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 $S = \{S_1, \ldots, S_k\}$ 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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Returns</th>
<th>$S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(start)</td>
<td>(empty)</td>
<td></td>
</tr>
<tr>
<td>MakeSet(a)</td>
<td>{a}</td>
<td></td>
</tr>
<tr>
<td>MakeSet(b)</td>
<td>{a}, {b}</td>
<td></td>
</tr>
<tr>
<td>FindSet(b)</td>
<td>$b$</td>
<td></td>
</tr>
<tr>
<td>Union(a, b)</td>
<td>{a, b}</td>
<td></td>
</tr>
<tr>
<td>FindSet(b)</td>
<td>$a$</td>
<td></td>
</tr>
<tr>
<td>FindSet(a)</td>
<td>{a, b}</td>
<td></td>
</tr>
<tr>
<td>MakeSet(c)</td>
<td>{a, b}, {c}</td>
<td></td>
</tr>
</tbody>
</table>
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 \texttt{elem} in the
list stores a pointer \texttt{elem.next} to the next element in the list, and the
set element itself, \texttt{elem.data}.

We also have an array \texttt{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:

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

**Example**

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

Then the following sequence of updates occurs:

\begin{itemize}
  
  \item \texttt{MakeSet(a)}
  
\end{itemize}
Example

MakeSet(c)

Example

Union(a, c)

Example

MakeSet(d)

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 $n$.
- 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 $O(m + n \log n)$.

**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.
- 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

\[
\text{ConnectedComponents}(G)\\
1. \text{for each vertex } v \in G: \text{MakeSet}(v)\\
2. \text{for each edge } u \leftrightarrow v \text{ in arbitrary order}\\
3. \text{if } \text{FindSet}(u) \neq \text{FindSet}(v)\\
4. \text{Union}(u, v)\\
\]

▶ 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, \text{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 \(T\).

\(T\) 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
▶ …

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:

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($G$)

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

First an arbitrary edge with weight 1 is picked:

![Graph](image1)

Example

Then any other edge with weight 1:

![Graph](image2)

Example

Then any other edge with weight 1:

![Graph](image3)

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:

![Graph](image4)
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.
- 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.

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 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 \( v \in G \): \( v.\text{key} \leftarrow \infty \), \( v.\pi \leftarrow \text{nil} \)
2. choose an arbitrary vertex \( r \)
3. \( r.\text{key} \leftarrow 0 \)
4. add every vertex in \( G \) to \( Q \)
5. while \( Q \) not empty
6. \( u \leftarrow \text{ExtractMin}(Q) \)
7. for each vertex \( v \) such that \( u \leftrightarrow v \)
8. if \( v \in Q \) and \( w(u, v) < v.\text{key} \)
9. \( v.\pi \leftarrow u \)
10. \( \text{DecreaseKey}(v, w(u, v)) \)

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.

![Graph](image)

**Example**

The state at the start of the algorithm:

![Graph](image)

▶ In the above diagram, the red text is the key values of the vertices (i.e. \( v.\text{key} \)) 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.

![Graph](image)

▶ 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:

```
A: 0, nil
B: 1, A
C: 1, A
D: 1, nil
E: 1, nil
F: ∞, nil
```

- 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):

```
A: 0, nil
B: 1, A
C: 1, A
D: 2, C
E: 2, C
F: 1, nil
```

- The red line shows the growing tree.

Example
Then B is extracted from the queue:

```
A: 0, nil
B: 1, A
C: 1, A
D: 2, C
E: 2, C
F: 2, nil
```

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

```
A: 0, nil
B: 1, A
C: 1, A
D: 2, C
E: 3, E
F: 1, nil
```

- The red line shows the growing tree.
Example

Then D is extracted from the queue:

```
A 0, nil
B 1, A
C 1, A
D 2, C
E 2, C
F 1, D
```

Example

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

```
A 0, nil
B 1, A
C 1, A
D 2, C
E 2, C
F 1, D
```

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 $T$.

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:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Underlying structure</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kruskal</td>
<td>Disjoint-set</td>
<td>$O(E \log E)$ (linked lists)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$O(E \alpha(V))$ (disjoint-set forest, edges already sorted)</td>
</tr>
<tr>
<td>Prim</td>
<td>Priority queue</td>
<td>$O(E \log V)$ (binary heap)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$O(E + V \log V)$ (Fibonacci heap)</td>
</tr>
</tbody>
</table>

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
  - 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.

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.