Undirected Graphs

- An undirected graph \( G = (V, E) \)
  - \( V \) a set of vertices
  - \( E \) a set of unordered edges \((v,w)\) where \( v, w \) in \( V \)

- USE: to model **symmetric** relationships between entities
- vertices \( v \) and \( w \) are **adjacent** if there is an edge \((v,w)\) [or \((w,v)\)]
- the edge \((v,w)\) is **incident** upon vertices \( v \) and \( w \)
- an edge may be \((v,w,c)\) where \( c \) is a **cost component** (e.g. distance)
Examples
**Terminology**

**PATH:** a sequence of vertices $v_1, v_2, \ldots v_n$ such that $(v_1, v_2), (v_2, v_3), \ldots (v_{n-1}, v_n)$ are edges

**LENGTH:** number of edges in a path

$v$ denotes a path length 0 from $v$ to $v$

**SIMPLE PATH:** all vertices are distinct

(except possibly the first and the last)

**SIMPLE CYCLE:** a simple path of length 3 or more that connects a vertex to itself

(undirected graph)
Sub-graph

- $G = (V, E)$
- A **sub-graph** of $G$ is a graph $G' = (V', E')$ where
  - $V'$ is a subset of $V$
  - $E'$ consists of edges $(v, w)$ such that both $v$ and $w$ are in $V'$
- If $E'$ consists of all edges $(v, w)$ in $E$ such that both $v$, $w$ in $V'$ then $G'$ is an **INDUCED SUB-GRAH** of $G$
- A **connected component** of a graph $G$ is a maximal connected induced sub-graph that is not itself a proper sub-graph of any other connected sub-graph of $G$
Sub-graph: example

graph $G$  sub-graph $G'$  (an) induced sub-graph

One connected component - namely $G$ itself
An Unconnected Graph

- **two connected components** (each a free tree)
- connected acyclic graph is a FREE TREE
  - every free tree with \( n \geq 1 \) vertices contains exactly \( (n-1) \) edges
  - any edge added to a free tree gives a cycle
Graph Representation

- **Adjacency Matrix**
- **Adjacency List**

### Adjacency Matrix

```
   a b c d
a 0 1 0 1
b 1 0 0 1
c 0 0 0 1
d 1 1 1 0
```

### Adjacency List

```
- a -> b -> d
- b -> a -> d
- c -> d
- d -> a -> b -> c
```
Operations

- insert
- remove
- find
- vertex
- edge
- navigate
- is_path
- is_cycle
- shortest path
- spanning forest
- list operations

Graph:
- a -> b -> d
- b -> a -> d
- c -> d
- d -> a -> b -> c
Minimum-cost Spanning Trees

- G = (V,E) where each edge (v,w) has an associated cost
- A **SPANNING TREE** for G is a **free tree** that connects all the vertices in G *(n nodes and (n-1) edges; no cycles)*
- The **cost of the spanning tree** is the sum of the costs of the edges in the tree

**Application areas**: communication networks *(transport/computer)*
MST Property

- \( G = (V, E) \)
  - a **connected graph** with a cost function on the edges
  - let \( U \) be a proper subset of \( V \)
  - if \( (u,v) \) is an edge of lowest cost such that
    - \( u \) in \( U \) and \( v \) in \( V-U \) then there is a MST that includes
      \( (u,v) \) as an edge
Building an MST: creative guess §1

Step 1

Step 2

Step 3

Step 4

Step 5
Kruskal’s principles

- Build a **priority queue** (PQ) with the edges, shortest edges first

- Each node in the graph becomes a **component**

- Choose an edge from the PQ such that the edge **connects 2 distinct components** until there is only one component – this is the MST
Kruskal’s principles - example

PQ: (a c 1), (d f 2), (b e 3), (c f 4), (a d 5), (b c 5), (c d 5), (a b 6), (c e 6), (e f 6)

Components: [a], [b], [c], [d], [e], [f] - 6 components

- (a c 1) → [a-c], [b], [d], [e], [f] - 5 components
- (d f 2) → [a-c], [b], [d-f], [e] - 4 components
- (b e 3) → [a-c], [b-e], [d, f] - 3 components
- (c f 4) → [a-c, c-f, f-d], [b-e] - 2 components
- (a d 5) → not chosen - a & d in same component
- (b c 5) → [a-c, c-b, b-e, c-f, f-d] - 1 component (MST)
### MST – explanation (Kruskal)

<table>
<thead>
<tr>
<th>Priority Queue</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>a c 1</td>
<td>The edges are stored in a PQ <em>(lowest values first)</em></td>
</tr>
<tr>
<td>d f 2</td>
<td>Each node becomes a component</td>
</tr>
<tr>
<td>b e 3</td>
<td>Each edge should connect 2 components</td>
</tr>
<tr>
<td>c f 4</td>
<td>NB: <em>a d 5</em> does not connect 2 components</td>
</tr>
<tr>
<td>a d 5</td>
<td>o a and d are in the same component (step 5 above)</td>
</tr>
<tr>
<td></td>
<td>o adding <em>a d 5</em> would also create a cycle</td>
</tr>
<tr>
<td></td>
<td>o An MST is a free tree and therefore has no cycles</td>
</tr>
<tr>
<td>b c 5</td>
<td><em>b c 5</em> completes the MST</td>
</tr>
<tr>
<td>c d 5</td>
<td>An MST with <em>n</em> nodes has <em>(n-1)</em> edges</td>
</tr>
<tr>
<td>a b 6</td>
<td>An MST is a <strong>Free Tree</strong> (no cycles)</td>
</tr>
<tr>
<td>c e 6</td>
<td></td>
</tr>
<tr>
<td>e f 6</td>
<td></td>
</tr>
</tbody>
</table>

**NB:** The edges are stored in a PQ (lowest values first). Each node becomes a component, and each edge should connect 2 components. Not all edges will be included in the MST. The MST is a free tree and therefore has no cycles.
Kruskal’s Algorithm  (creative guess §1)

- One method of constructing an MST is Kruskal’s
- start with a graph $T = (V, \pi)$ i.e. only the vertices of $G = (V, E)$
- each vertex is a connected component (in the graph $T$)
- to construct the MST, $T$ examine the edges in $E$ in order of increasing cost (implementation - priority queue)
- if the edge connects two vertices in two connected components then add the edge to $T$ (otherwise discard the edge)
- when all the edges are in one component, $T$ is a MST for $G$
Kruskal’s Algorithm

- \( S = \) set of connected components \( (V \) from \( G=(V,E) \))
- \( \text{merge}(A, B, S) \) -- merge components \( A \) & \( B \) in \( S \) - rename \( A \)
- \( \text{find}(v, S) \) -- return name of component \( X \) in \( S : v \) in \( X \)
- \( \text{initial}('A', v, S) \) -- make \( A \) the name of component in \( S \) containing only vertex \( v \) initially
- \( \text{insert}(e, S) \) -- add a given edge to \( S \)
- \( \text{remove}_\text{pq}() \) -- remove an edge from the PQ
- \( (x, y, c) \) -- edge \( (x, y) \) in PQ with cost \( c \)
Kruskal’s Algorithm

for each v in S initial ( next(name), v, S) -- initialise
while (size(S) > 1 { -- size = number of components
    get_PQ ( ); -- get (x, y, c) from PQ
    if ( find(x, S) != find(y, S) ) { -- x, y in different components
        merge ( find (x, S), find (y, S), S );
        insert (get_PQ ( ), S);
    }
    remove_pq( );
}
Kruskal: example
Kruskal: example

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>c</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>d</td>
<td>f</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>b</td>
<td>e</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>c</td>
<td>f</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>a</td>
<td>d</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>b</td>
<td>c</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>c</td>
<td>d</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>c</td>
<td>e</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>e</td>
<td>f</td>
<td></td>
<td>6</td>
</tr>
</tbody>
</table>

PQ
Kruskal: example
Kruskal: example
Kruskal: example

<table>
<thead>
<tr>
<th>a</th>
<th>c</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>f</td>
<td>2</td>
</tr>
<tr>
<td>b</td>
<td>e</td>
<td>3</td>
</tr>
<tr>
<td>c</td>
<td>f</td>
<td>4</td>
</tr>
<tr>
<td>a</td>
<td>d</td>
<td>5</td>
</tr>
<tr>
<td>b</td>
<td>c</td>
<td>5</td>
</tr>
<tr>
<td>c</td>
<td>d</td>
<td>5</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>6</td>
</tr>
<tr>
<td>c</td>
<td>e</td>
<td>6</td>
</tr>
<tr>
<td>e</td>
<td>f</td>
<td>6</td>
</tr>
</tbody>
</table>

PQ
Kruskal: example

<table>
<thead>
<tr>
<th>a</th>
<th>c</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>f</td>
<td>2</td>
</tr>
<tr>
<td>b</td>
<td>e</td>
<td>3</td>
</tr>
<tr>
<td>c</td>
<td>f</td>
<td>4</td>
</tr>
<tr>
<td>a</td>
<td>d</td>
<td>5</td>
</tr>
<tr>
<td>b</td>
<td>c</td>
<td>5</td>
</tr>
<tr>
<td>c</td>
<td>d</td>
<td>5</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>6</td>
</tr>
<tr>
<td>c</td>
<td>e</td>
<td>6</td>
</tr>
<tr>
<td>e</td>
<td>f</td>
<td>6</td>
</tr>
</tbody>
</table>

PQ

\[ \text{Red edges represent the spanning tree.} \]
Kruskal: example
Kruskal: Comment

- Using the PQ, the algorithm is reasonably easy to understand in principle (the pictorial representation is easy to follow)

- In general it is worth looking at the problem and its solution before going through any algorithm in detail

- Look at each line of the pseudo code and be sure that you can relate the code to the action required i.e. that you can interpret the code
Building an MST: creative guess §2

step 1

step 2

step 3

step 4

step 5
Prim’s principles

- given start node x mark as visited;
- note the edge values from x to the remaining nodes; this uses 2 arrays L for the edge lengths and C for the node name;
- find the shortest edge from x to y; mark y as visited;
- build a COMPONENT (x y) i.e. y is then added to the component (i.e. the visited nodes);
- now examine the edge costs from y to the remaining nodes; if this edge is cheaper, replace the current edge with this edge. The new node is added to the component.

- Repeat for the unvisited nodes. The component grows node by node and cheaper edges replace those edges previously found as cheaper.
Prim’s principles example

- \((a \ b \ 6), (a \ c \ 1), (a \ d \ 5), (b \ c \ 5), (b \ e \ 3), (c \ d \ 5), (c \ e \ 6), (c \ f \ 4), (d \ f \ 2), (e \ f \ 6)\)
- Start node a – visited \{a\} – unvisited \{b, c, d, e, f\}
- \(L = [6, 1, 5, \$, \$] C = [a, a, a, a, a]\)
- Shortest edge \((a \ c \ 1)\) – visited \{a, c\} – unvisited \{b, d, e, f\}
- \((c \ b \ 5)\) is cheaper \(\Rightarrow\)
- \((c \ d \ 5)\) not cheaper \(\Rightarrow\) no change
- \((c \ e \ 6)\) is cheaper \(\Rightarrow\)
- \((c \ f \ 4)\) is cheaper \(\Rightarrow\)

\[
L = [5, 1, 5, 6, \$, \$] C = [c, a, a, c, a]
\]

\[
L = [5, 1, 5, 6, 4] C = [c, a, a, c, c]
\]
Prim’s principles example

- (a b 6), (a c 1), (a d 5), (b c 5), (b e 3), (c d 5), (c e 6), (c f 4), (d f 2), (e f 6)
- \( L = \{5, 1, 5, 6, 4\} \quad C = \{c, a, a, c, c\} \)
- Shortest edge (c f 4) – visited \{a, c, f\} – unvisited \{b, d, e\}
- (f b §) not cheaper \(\Rightarrow\) no change
- (f d 2) is cheaper \(\Rightarrow\)
- \( L = \{5, 1, 2, 6, 4\} \quad C = \{c, a, f, c, c\} \)
- (f e 6) not cheaper \(\Rightarrow\) no change
- Shortest edge (f d 2) – visited \{a, c, d, f\} – unvisited \{b, e\}
- (d b §) not cheaper \(\Rightarrow\) no change
- (d e §) not cheaper \(\Rightarrow\) no change
- \( L = \{5, 1, 2, 6, 4\} \quad C = \{c, a, f, c, c\} \)
Prim’s principles example

- \((a\ b\ 6), (a\ c\ 1), (a\ d\ 5), (b\ c\ 5), (b\ e\ 3), (c\ d\ 5), (c\ e\ 6), (c\ f\ 4), (d\ f\ 2), (e\ f\ 6)\)
  - \(L = [5, 1, 2, 6, 4]\ C = [c, a, f, c, c]\)
- Shortest edge \((c\ b\ 5)\) – visited \{a, b, c, d, f\} – unvisited \{e\}
  - \((b\ e\ 3)\) is cheaper
    - \(L = [5, 1, 2, 3, 4]\ C = [c, a, f, b, c]\)
- Shortest edge \((b\ e\ 3)\) – visited \{a, b, c, d, e, f\} – unvisited \{\}\ empty – STOP
- Result
  - \(L = [5, 1, 2, 3, 4]\ C = [c, a, f, b, c]\)
Prim’s principles - pictures

L = [6, 1, 5, §, §]
C = [a, a, a, a, a]

L = [5, 1, 5, §, §]
C = [c, a, a, a, a]

L = [5, 1, 5, §, §]
C = [c, a, a, a, a]
Prim’s principles - pictures

L = [5, 1, 5, 6, §]
C = [c, a, a, c, a]

L = [5, 1, 5, 6, 4]
C = [c, a, a, c, c]
Prim’s principles - pictures

L = [5, 1, 2, 6, 4]
C = [c, a, f, c, c]

L = [5, 1, 2, 6, 4]
C = [c, a, f, c, c]
Prim’s principles - pictures

L = [5, 1, 2, 3, 4]
C = [c, a, f, b, c]
Prim’s algorithm is a **greedy** algorithm
- greedy = takes the locally best solution
- The MST “grows” the MST as one component (similar to Dijkstra)

**Process**
- Choose the **cheapest edge** from the component to an **unvisited node**, add edge to the MST and **mark the node as visited** (U)
- **Start at node a** – choose the cheapest edge
  - a c 1 mark c
  - Now choose the cheapest edge U = {a,c} c f 4 mark f
  - Now choose the cheapest edge U = {a,c,f} f d 2 mark d
  - Now choose the cheapest edge U = {a,c,f,d} c b 5 mark b
  - Now choose the cheapest edge U = {a,c,f,d,b} b e 5 mark e
- All nodes have now been visited U = {a,c,f,d,b,e} stop.
Prim’s Algorithm (creative guess §2)

- \( V = \{a,b,c,d,\ldots\} \)
- initialise \( U \) to \( \{a\} \)
- the spanning tree grows one edge at a time
- each step:
  - find the shortest edge \((u,v)\) that connects \( U \) and \( V-U \)
  - add \( v \) to \( U \)
  - until \( U = V \) i.e. \( V-U = \emptyset \)
- cost matrix \( C \) gives the costs of each edge
Prim’s Algorithm

Prim ( node v)  -- v is the start node

{ U = {v}; for i in (V-U) { low-cost[i] = C[v,i]; closest[i] = v; } }

while (!is_empty (V-U) ) {  -- find the closest vertex in V-U
  i = first(V-U); min = low-cost[i]; k = i; -- minimum cost edge
  for j in (V-U-k) if (low-cost[j] < min) {min = low-cost[j]; k = j; }
  display(k, closest[k]); -- display edge
  U = U + k; -- k added to U
  for j in (V-U) if ( C[k,j] < low-cost[j] ) ) -- readjust costs
    {low-cost[j] = C[k,j]; closest[j] = k; }
}

See http://www.cs.kau.se/cs/education/courses/dvgb03/revision/index.php?PrimEx=1
Prim: example

Graph:
Prim: example

Init: U  V-U
   {a}  {b,c,d,e,f}  low-cost  closest  k / min
display ((a,c))
   {a,c}  {b,d,e,f}  (-,6, 1, 5, §, §)  (-,a,a,a,a)  c / 1
   display ((c,f))
   {a,c,f}  {b,d,e}  (-,5, 1, 5, 6, 4)  (-,c,a,a,c,c)  f / 4
   display ((f,d))
   {a,c,f,d}  {b,e}  (-,5, 1, 2, 6, 4)  (-,c,a,f,c,c)  d / 2
   display ((c,b))
   {a,c,f,d,b}  {e}  (-, 5, 1, 2, 3, 4)  (-,c,a,f,b,c)  e / 3
   display ((b,e))
   {a,c,f,d,b,e}  {}  (-, 5, 1, 2, 3, 4)  (-,c,a,f,b,c)  

See http://www.cs.kau.se/cs/education/courses/dvgb03/revision/index.php?PrimEx=1
Since the MST is a free tree, there are n-1 edges, hence n-1 iterations.

The following code finds the least cost edge between U and V-U: (min, k)

\[
\begin{align*}
\text{min} &= \text{low-cost}[i]; \quad k = i; \\
\text{for } j \text{ in } (V-U-k) \text{ if } (\text{low-cost}[j] < \text{min}) \{ \text{min} = \text{low-cost}[j]; \quad k = j; \} 
\end{align*}
\]

The following code may be replaced by for example: Add_MST(u,v)

\[
\begin{align*}
\text{display}(k, \text{closest}[k]); \\
\text{-- display edge}
\end{align*}
\]

The re-adjustment of the costs is perhaps the trickiest to understand but is in effect similar to Dijkstra - finding the cheapest (i,j) or (i,k) (k,j)

\[
\begin{align*}
\text{for } j \text{ in } (V-U) \text{ if } (\text{C}[k,j] < \text{low-cost}[j]) \quad \\
\{ \text{low-cost}[j] = \text{C}[k,j]; \quad \text{closest}[j] = k; \} 
\end{align*}
\]