Undirected Graphs: Depth First Search

- Similar to the algorithm for directed graphs
- \((v, w)\) is similar to \((v,w) (w,v)\) in a digraph
- For the depth first spanning forest (dfs), each connected component in the graph will have a tree in the dfsf
  - (if the graph has one component, the dfsf will consist of one tree)
- In the dfsf for digraphs, there were 4 kinds of edges: **tree, forward, back and cross**
- For a graph there are 2: **tree** and **back** edges (forward and back edges are not distinguished and there are no cross edges)
Undirected Graphs: Depth First Search

- **Tree edges:**
  - edges \((v,w)\) such that \(\text{dfs}(v)\) directly calls \(\text{dfs}(w)\) (or vice versa)

- **Back edges:**
  - edges \((v,w)\) such that neither \(\text{dfs}(v)\) nor \(\text{dfs}(w)\) call each other directly (e.g. \(\text{dfs}(w)\) calls \(\text{dfs}(x)\) which calls \(\text{dfs}(v)\) so that \(w\) is an ancestor of \(v\))

- in a dfs, the vertices can be given a dfs number similar to the directed graph case
DFS: Example

start: a
a b d e c f g
DFS: Example

```
\[
\begin{array}{ccccccc}
\text{a(1)} & \text{b} & \text{c} & \text{d} & \text{e} \\
\text{b} & \text{a} & \text{d} & \text{e} \\
\text{c} & \text{a} & \text{f} & \text{g} \\
\text{d} & \text{a} & \text{b} & \text{e} \\
\text{e} & \text{a} & \text{b} & \text{d} \\
\text{f} & \text{c} & \text{g} \\
\text{g} & \text{c} & \text{f}
\end{array}
\]
```
DFS: Example

a(1)
DFS: Example

a(1) → b(2)
DFS: Example

a(1) ➔ b(2)
DFS: Example

a(1) ➔ b(2) ➔ d(3)
DFS: Example

a(1) ➔ b(2) ➔ d(3)
DFS: Example

\[ a(1) \rightarrow b(2) \rightarrow d(3) \rightarrow e(4) \]
DFS: Example

a(1) → b(2) → d(3) → e(4)
DFS: Example

```
a(1) ➔ b(2)
```

```
[Diagram of a graph with vertices a, b, c, d, e, f, g and arrows indicating the order of visiting nodes during a depth-first search.]
```
DFS: Example

```
   a  b  c  d  e
  /  |  |  |  |
 b  a  d  e
 /  |  |  |
c  a  f  g
 /  |  |  
d  a  b  e
 /  |  |  
e  a  b  d
 /  |  |  
f  c  g
 /  |  
g  c  f
```

Graph:

```
   a  b  c  d  e
  /  |  |  |  |
 b  a  d  e
 /  |  |  |
c  a  f  g
 /  |  |  
d  a  b  e
 /  |  |  
e  a  b  d
 /  |  |  
f  c  g
 /  |  
g  c  f
```
DFS: Example

a(1) \rightarrow c(5)
DFS: Example

a(1) ➔ c(5)
DFS: Example

\[ a(1) \rightarrow c(5) \rightarrow f(6) \]
DFS: Example

a(1) ➞ c(5) ➞ f(6)
DFS: Example

a(1) \rightarrow c(5) \rightarrow f(6) \rightarrow g(7)
DFS: Example

a(1) \rightarrow c(5) \rightarrow f(6) \rightarrow g(7)
DFS: Example

a(1) → c(5)
DFS: Example

![Graph Example]

DFS: Example
DFSF: Example (Depth-First Spanning Forest)
Undirected Graphs: Breadth First Search

- for each vertex $v$, visit all the adjacent vertices first
- a breadth-first spanning forest can be constructed
  - consists of
    - tree edges: edges $(v,w)$ such that $v$ is an ancestor of $w$ (or vice versa)
    - cross edges: edges which connect two vertices such that neither is an ancestor of the other
- NB the search only works on one connected component
  - if the graph has several connected components then apply bfs to each component
BFSF: Example

Note that this represents the MST for an unweighted undirected graph.
BFSF: algorithm (Breadth-First Spanning Forest)

bfs ( )
{
    mark v visited; enqueue (v);
    while ( not is_empty (Q) ) {
        x = front (Q); dequeue (Q);
        for each y adjacent to x if y unvisited {
            mark y visited; enqueue (y);
            insert ( (x, y) in T );
        }
    }
}
Articulation Point

- An articulation point of a graph is a vertex \( v \) such that if \( v \) and its incident edges are removed, a connected component of the graph is broken into two or more pieces.
- A connected component with no articulation points is said to be biconnected.
- The DFS can be used to help find the biconnected components of a graph.
- Finding articulation points is one problem concerning the connectivity of graphs.
Connectivity

- finding articulation points is one problem concerning the connectivity of graphs

- a graph has **connectivity k** if the deletion of any \((k-1)\) vertices fails to disconnect the graph (what does this mean?)
  - e.g. a graph has connectivity 2 or more iff it has no articulation points i.e. iff it is biconnected

- the higher the connectivity of a graph, the more likely the graph is to survive failure of some of its vertices
  - e.g. a graph representing sites which must be kept in communication (computers / military / other)
Articulation Points / Connectivity: Example

- Articulation points are a and c
- Removing a gives \{b,d,e\} and \{c,f,g\}
- Removing c gives \{a,b,d,e\} and \{f,g\}
- Removing any other vertex does not split the graph
Articulation Points: Algorithm

- Perform a **dfs of the graph**, computing the df-number for each vertex \( v \)
  (df-numbers order the vertices as in a pre-order traversal of a tree)
- for each vertex \( v \), compute \( \text{low}(v) \) - the smallest df-number of \( v \) or any vertex \( w \) reachable from \( v \) by following down 0 or more tree edges to a descendant \( x \) of \( v \) (\( x \) may be \( v \)) and then following a back edge \( (x, w) \)
- compute \( \text{low}(v) \) for each vertex \( v \) by visiting the vertices in post-order traversal
- when \( v \) is processed, \( \text{low}(y) \) has already been computed for all children \( y \) of \( v \)
Articulation Points: Algorithm

- \( \text{low}(v) \) is taken to be the minimum of
  - df-number(v)
  - df-number(z) for any vertex z where (v,z) is a back edge
  - low(y) for any child y of v

**Example**

- \( e = \min(4, (1,2), -) \)
- \( d = \min(3, 1, 1) \) \( b = \min(2, -, 1) \)
- \( g = \min(7, 5, -) \) \( f = \min(6, -, 5) \)
- \( c = \min(5, -, 5) \)
- \( a = \min(1, -, (1,5)) \)
Articulation Points: Algorithm

- the root is an AP iff it has 2 or more children
  - since it has no cross edges, removal of the root must disconnect the sub-trees rooted at its children
  - removing a => {b, d, e} and {c, f, g}

- a vertex v (other than the root) is an AP iff there is some child w of v such that low(w) >= df-number(v)
  - v disconnects w and its descendants from the rest of the graph
  - if low(w) < df-number(v) there must be a way to get from w down the tree and back to a proper ancestor of v (the vertex whose df-number is low(w)) and therefore deletion of v does not disconnect w or its descendants from the rest of the graph
Articulation Points: Example 1

- root - 2 or more children
- other vertices
  - some child w of v such that low(w) >= df-number(v)
- example
  - a root >= 2 children
  - b low(e) = 1 dfn = 2
  - c low(g) = 5 dfn = 5
  - d low(e) = 1 dfn = 3
  - e N/A
  - f low(g) = 5 dfn = 6
  - g N/A
Articulation Points: Example 2

- root - 2 or more children
- other vertices
  - some child w of v such that low(w) >= df-number(v)
- example
  - a  root >= 2 children
  - b  low(e) = 1  dfn = 2
  - c  low(g) = 7  dfn = 5
  - d  low(e) = 1  dfn = 3
  - e  N/A
  - f  low(g) = 7  dfn = 6
  - g  N/A
A graph $G$ is **bipartite** if $V$ is the disjoint union of $V_1$ and $V_2$ such that no $x_i$ and $x_j$ in $V_1$ are adjacent (similarly $y_i$ and $y_j$ in $V_2$).

**Example**

- Set of courses
- Set of teachers
- Edge $\Rightarrow$ can teach course
- (marriage problem!)
Bipartite Graph: Matching Problem

- A matching in a bipartite graph (BG) is a set of edges whose end points are distinct.
- A matching is **complete** if every member of $V_1$ is the end point of one of the edges in the matching.
- A matching is **perfect** if every member of $V$ is the end point of one of the edges in the matching.
- In a BG where $V = V_1 \text{ disjoint union } V_2$, there is a **complete matching** iff for every subset $C$ of $V_1$ there are at least $|C|$ vertices in $V_2$ adjacent to members of $C$.
- In a BG where $V = V_1 \text{ disjoint union } V_2$, there is a **perfect matching** iff for every subset $C$ of $V_1$ there are at least $|C|$ vertices in $V_2$ adjacent to members of $C$ and $|V_1| = |V_2|$.
BG Matching: Example
Königsberg Bridge Problem (Euler)

- Find a cycle in the graph G that includes all the vertices and all the edges in G – **Euler Cycle**
- if G has an Euler cycle, then G is connected and every vertex has an **even degree**
- \( \text{degree}(v) = \) number of edges incident on v
Hamiltonian Cycle

- Hamiltonian cycle: cycle in a graph $G = (V,E)$ which contains each vertex in $V$ exactly once, except for the starting and ending vertex that appears twice.

- \(\text{degree}(v) = 2\) for all \(v\) in \(V\)
TSP Problem

- **What may we assume?**
- Graph is fully connected
- a-b,5 = 5
- a-c,sqrt(50) = 7+
- a-d,sqrt(274) = 16+
- a-e,sqrt(241) = 15+
- a-f,18 = 18
- b-c,5 = 5
- b-d,sqrt(137) = 11+
- b-e,sqrt(122) = 11+
Start estimating!

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>5</td>
<td>7+</td>
<td>16+</td>
<td>15+</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>5</td>
<td>5</td>
<td>11+</td>
<td>11+</td>
<td>15+</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>7+</td>
<td>5</td>
<td>14</td>
<td>14+</td>
<td>18+</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>16+</td>
<td>11+</td>
<td>14</td>
<td>3</td>
<td>7+</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>15+</td>
<td>11+</td>
<td>14+</td>
<td>3</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>18</td>
<td>15+</td>
<td>18+</td>
<td>7+</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
TSP Problem

Start estimating!

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>5</td>
<td>7+</td>
<td>16+</td>
<td>15+</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>5</td>
<td>5</td>
<td>11+</td>
<td>11+</td>
<td>15+</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>7+</td>
<td>5</td>
<td>14</td>
<td>14+</td>
<td>18+</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>16+</td>
<td>11+</td>
<td>14</td>
<td>3</td>
<td>7+</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>15+</td>
<td>11+</td>
<td>14+</td>
<td>3</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>18</td>
<td>15+</td>
<td>18+</td>
<td>7+</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
TSP Problem

Adapt Kruskal PQ plus degree max 2 (see below)

1. d-3-e
2. a-5-b, b-5-c, e-5-f
3. c-14-d
4. a-18-f

(0,0,0,1,1,0) \rightarrow (1,1,0,1,1,0) \rightarrow (1,2,1,1,1,0) \rightarrow (1,2,2,2,2,1) \rightarrow (2,2,2,2,2,2)
Travelling Salesman Problem (TSP)

- **Euler / Hamilton**
  - E visits each edge once
  - H visits each vertex once
  - to find an Euler cycle - O(n)

- **Hamilton**
  - factorial or exponential

- **Hamilton - applications**
  - TSP
  - knight’s tour of n * n board

- **TSP**
  - Find the minimum-length Hamiltonian cycle for G
  - salesman starts and ends at x

- **TSP Algorithm**
  - variant of Kruskal’s
  - edge acceptance conditions
    - degree(v) should not >= 3
    - no cycles unless # selected edges = |V|
    - greedy / near-optimal
Graphs: Summary 1

- Directed Graphs
  - G = (V, E)
  - create / destroy G
  - add / remove V
    (=> remove E)
  - add / remove E
  - is_path(v, w)
  - path_length(v, w)
  - is_cycle(v)
  - is_connected(G)
  - is_complete(G)

- Undirected Graphs
  - G = (V, E)
  - create / destroy G
  - add / remove V
    (=> remove E)
  - add / remove E
  - is_path(v, w)
  - path_length(v, w)
  - is_cycle(v)
  - is_connected(G)
  - is_complete(G)
Graphs: Summary 2

- Directed Graphs
  - navigation
    - depth-first search (dfs)
    - breadth-first search (bfs)
    - Warshall
  - spanning forests
    - df spanning forest (dfsf)
    - bf spanning forest (bfsf)
  - minimum cost algorithms
    - Dijkstra (single path)
    - Floyd (all paths)

- Undirected Graphs
  - navigation
    - depth-first search (dfs)
    - breadth-first search (bfs)
    - Warshall
  - spanning forests
    - df spanning forest (dfsf)
    - bf spanning forest (bfsf)
  - minimum cost algorithms
    - Prim (spanning tree)
    - Kruskal (spanning tree)
Graphs: Summary 3

- Directed Graphs
  - topological sort (DAG)
  - strong components
  - reduced graph

- Undirected Graphs
  - sub-graph
  - induced sub-graph
  - unconnected graph-free tree
  - articulation points
  - connectivity
  - bipartite graph & matching
  - Königsberg Bridge Problem
  - Hamiltonian cycles
  - Travelling Salesman