Graphs

An Introduction

Outline

- What are Graphs?
- Applications
  - Terminology and Problems
  - Representation (Adj. Mat and Linked Lists)
- Searching
  - Depth First Search (DFS)
  - Breadth First Search (BFS)

Graphs

- A graph \( G = (V,E) \) is composed of:
  - \( V \): set of vertices
  - \( E \subseteq V \times V \): set of edges connecting the vertices
- An edge \( e = (u,v) \) is a ___ pair of vertices
  - Directed graphs (ordered pairs)
  - Undirected graphs (unordered pairs)
Directed graph

Directed Graph

Undirected GRAPH
Undirected Graph

Applications

- Air Flights, Road Maps, Transportation.
- Graphics / Compilers
- Electrical Circuits
- Networks
- Modeling any kind of relationships (between people/web pages/cities/…)

Some More Graph Applications

<table>
<thead>
<tr>
<th>Graph</th>
<th>Node</th>
<th>Edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transportation</td>
<td>street intersections</td>
<td>highway</td>
</tr>
<tr>
<td>Communication</td>
<td>computers</td>
<td>fiber optic cable</td>
</tr>
<tr>
<td>World Wide Web</td>
<td>web page</td>
<td>hyperlink</td>
</tr>
<tr>
<td>Social</td>
<td>people</td>
<td>relationship</td>
</tr>
<tr>
<td>Food Web</td>
<td>species</td>
<td>predator/prey</td>
</tr>
<tr>
<td>Software Systems</td>
<td>function</td>
<td>function calls</td>
</tr>
<tr>
<td>Scheduling</td>
<td>task</td>
<td>precedence constraints</td>
</tr>
<tr>
<td>Circuits</td>
<td>gate</td>
<td>wire</td>
</tr>
</tbody>
</table>

-
World Wide Web

- Web graph.
  - Node: web page.
  - Edge: hyperlink from one page to another.

9-11 Terrorist Network

- Social network graph.
  - Node: people.
  - Edge: relationship between two people.

Ecological Food Web

- Food web graph.
  - Node: species.
  - Edge: from prey to predator.
Terminology

- **a** is adjacent to **b** iff \((a, b) \in E\).
- **degree**\((a) = \) number of adjacent vertices (Self loop counted twice)
- **Self Loop**\( (a,a) \)
- **Parallel edges**: \( E = \{ ...(a,b), (a,b)... \} \)

Terminology

- **A Simple Graph** is a graph with no self loops or parallel edges.
- **Incidence**: \(v\) is incident to \(e\) if \(v\) is an end vertex of \(e\).

More…
Question

• Max Degree node? Min Degree Node?
• Isolated Nodes? Total sum of degrees over all vertices? Number of edges?

• Max Degree = 4. Isolated vertices = 1.
• |V| = 8, |E| = 8
• Sum of degrees = 16 = ?

– (Formula in terms of |V|, |E| ?)

Handshaking Theorem. Why?
QUESTION

• How many edges are there in a graph with 100 vertices each of degree 4?

– Total degree sum = 400 = 2 |E|
– 200 edges by the handshaking theorem.

Handshaking: Corollary

The number of vertices with odd degree is always even.

Proof: Let \( V_1 \) and \( V_2 \) be the set of vertices of even and odd degrees, respectively (Hence \( V_1 \cap V_2 = \emptyset \) and \( V_1 \cup V_2 = V \)).

• Now we know that

\[
2|E| = \sum_{v \in V} \text{degree}(v) = \sum_{v \in V_1} \text{degree}(v) + \sum_{v \in V_2} \text{degree}(v)
\]

• Since \( \text{degree}(v) \) is odd for all \( v \in V_2 \), \( |V_2| \) must be even.
Terminology

Path and Cycle

- An alternating sequence of vertices and edges beginning and ending with vertices
  - each edge is incident with the vertices preceding and following it.
  - No edge / vertex appears more than once.
  - A path is simple if all nodes are distinct.
- Cycle
  - A path is a cycle if and only if $v_0 = v_k$
    - The beginning and end are the same vertex.

Path example
Connected graph
- Undirected Graphs: If there is at least one path between every pair of vertices. (otherwise disconnected)
- Directed Graphs:
  - Strongly connected
  - Weakly connected

hamiltonian cycle
- Closed cycle that transverses every vertex exactly once.

In general, the problem of finding a Hamiltonian circuit is NP-Complete.

complete graph
- Every pair of graph vertices is connected by an edge.
Directed Acyclic Graphs

A DAG is a directed graph with no cycles.

Often used to indicate precedences among events, i.e., event $a$ must happen before $b$.

- Where have we seen these graphs before?

Tree

A connected graph with $n$ nodes and $n-1$ edges.

A Forest is a collection of trees.

Trees

- An undirected graph is a tree if it is connected and does not contain a cycle.

Theorem. Let $G$ be an undirected graph on $n$ nodes. Any two of the following statements imply the third.
- $G$ is connected.
- $G$ does not contain a cycle.
- $G$ has $n-1$ edges.
Rooted Trees

- Rooted tree. Given a tree $T$, choose a root node $r$ and orient each edge away from $r$.

- Importance. Models hierarchical structure.

Phylogeny Trees

- Phylogeny trees. Describe evolutionary history of species.

GUI Containment Hierarchy

- GUI containment hierarchy. Describe organization of GUI widgets.

References: [http://java.sun.com/docs/books/tutorial/uiswing/overview/anatomy.html](http://java.sun.com/docs/books/tutorial/uiswing/overview/anatomy.html)
Spanning tree

Connected subset of a graph G with n-1 edges which contains all of V

independent set

• An independent set of G is a subset of the vertices such that no two vertices in the subset are adjacent.

cliques

• a.k.a. complete subgraphs.
tough Problem

• Find the maximum cardinality independent set of a graph G.
  – NP-Complete

tough problem

• Given a weighted graph G, the nodes of which represent cities and weights on the edges, distances; find the shortest tour that takes you from your home city to all cities in the graph and back.
  – Can be solved in $O(n!)$ by enumerating all cycles of length $n$.
  – Dynamic programming can be used to reduce it in $O(n^22^n)$.

representation

• Two ways
  – Adjacency List
    • (as a linked list for each node in the graph to represent the edges)
  – Adjacency Matrix
    • (as a boolean matrix)
Representing Graphs

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Adjacent Vertices</th>
<th>Initial Vertex</th>
<th>Terminal Vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2, 3, 4</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1, 4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1, 4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1, 2, 3</td>
<td>4</td>
<td>1, 2, 3</td>
</tr>
</tbody>
</table>

adjacency list

adjacency matrix
AL Vs AM

- AL: Takes $O(|V| + |E|)$ space
- AM: Takes $O(|V|^2)$ space

Question: How much time does it take to find out if $(v_i,v_j)$ belongs to $E$?
- AM?
- AL?
AL Vs AM

- AL: Total space = $4|V| + 8|E|$ bytes (For undirected graphs its $4|V| + 16|E|$ bytes)
- AM: $|V| \times |V| / 8$

- Question: What is better for very sparse graphs? (Few number of edges)

Graph Traversal

- s-t connectivity problem. Given two node s and t, is there a path between s and t?
- s-t shortest path problem. Given two node s and t, what is the length of the shortest path between s and t?

Connectivity

- Applications:
  - Maze traversal
  - Kevin Bacon number / Erdos number
  - Fewest number of hops in a communication network
  - Friendster
BFS/DFS

• Breadth-first search (BFS) and depth-first search (DFS) are two distinct orders in which to visit the vertices and edges of a graph.

BFS: radiates out from a root to visit vertices in order of their distance from the root. Thus closer nodes get visited first.

Breadth first search

• Question: Given G in AM form, how do we say if there is a path between nodes a and b?

Note: Using AM or AL its easy to answer if there is an edge (a,b) in the graph, but not path questions. This is one of the reasons to learn BFS/DFS.
BFS

- A Breadth-First Search (BFS) traverses a connected component of a graph, and in doing so defines a spanning tree.

Source: Lecture notes by Sheung-Hung POON

Algorithm BFS(v)
Input: \( v \) is the source vertex
Output: Mark all vertices that can be visited from \( v \).
1. for each vertex \( u \)
2. do \( \text{visited}[u] := \text{false} \);
3. \( Q = \text{empty queue} \);
4. \( \text{visited}[v] := \text{true} \);
5. enqueue \( Q \), \( v \);
6. while \( Q \) is not empty
7. do \( v := \text{dequeue}(Q) \);
8. for each \( w \) adjacent to \( v \)
9. do if \( \text{visited}[w] = \text{false} \)
10. then \( \text{visited}[w] := \text{true} \);
11. enqueue \( Q \), \( w \)

Example

Adjacency List

Visited Table (T/F)

Q := \{ \}

Initialize Q to be empty
Example

Adjacency List

Visited Table (T/F)

Flag that 2 has been visited.

Q = \{ 2 \}
Place source 2 on the queue.

Example

Adjacency List

Visited Table (T/F)

Mark neighbors as visited.

Q = \{ 2 \} \rightarrow \{ 8, 1, 4 \}
Dequeue 2.
Place all unvisited neighbors of 2 on the queue

Example

Adjacency List

Visited Table (T/F)

Mark new visited Neighbors.

Q = \{ 8, 1, 4 \} \rightarrow \{ 1, 4, 0, 9 \}
Dequeue 8.
Place all unvisited neighbors of 8 on the queue.
Notice that 2 is not placed on the queue again, it has been visited!
Example

Adjacency List

Mark new visited Neighbors.

Q = \{ 1, 4, 0, 9 \} \rightarrow \{ 4, 0, 9, 3, 7 \}

Mark new visited Neighbors.

Q = \{ 4, 0, 9, 3, 7 \} \rightarrow \{ 0, 9, 3, 7 \}

Mark new visited Neighbors.

Q = \{ 0, 9, 3, 7 \} \rightarrow \{ 9, 3, 7 \}

Mark new visited Neighbors.

Q = \{ 9, 3, 7 \} \rightarrow \{ 3, 7 \}

Mark new visited Neighbors.

Q = \{ 3, 7 \} \rightarrow \{ 7 \}

Mark new visited Neighbors.

Q = \{ 7 \} \rightarrow \{ 7 \}

Mark new visited Neighbors.

Q = \{ 7 \} \rightarrow \{ 7 \}
Example

Adjacency List

*source* 0 1 2 3 4 5 6 7 8 9

Visited Table (T/F)

T T T T T F F T T T

$Q = \{9, 3, 7\} \rightarrow \{3, 7\}$

Dequeue 9.

-- 9 has no unvisited neighbors!

-- place neighbor 5 on the queue.

Mark new visited Vertex 5.

Example

Adjacency List

*source* 0 1 2 3 4 5 6 7 8 9

Visited Table (T/F)

T T T T T F F T T T

$Q = \{3, 7\} \rightarrow \{7, 5\}$

Dequeue 7.

-- place neighbor 6 on the queue.

Mark new visited Vertex 6.
Example

Adjacency List | Visited Table (T/F)
--- | ---
0 1 2 3 4 5 6 7 8 9
T T T T T T T T T

Q = {5, 6} → {6}
Dequeue 5.
→ no unvisited neighbors of 5.

Neighbors

Example

Adjacency List | Visited Table (T/F)
--- | ---
0 1 2 3 4 5 6 7 8 9
T T T T T T T T T

Q = {6} → {}  
Dequeue 6.
→ no unvisited neighbors of 6.

Neighbors

Example

Adjacency List | Visited Table (T/F)
--- | ---
0 1 2 3 4 5 6 7 8 9
T T T T T T T T T

Q = {} STOP!!! Q is empty!!!

What did we discover?
Look at "visited" tables.
There exist a path from source vertex 2 to all vertices in the graph!
Time Complexity of BFS
(Using adjacency list)

Assume adjacency list
- \( n = \) number of vertices \( m = \) number of edges

Algorithm: BFS(u)

Input: \( u \) is the source vertex
Output: Mark all vertices that can be reached from \( u \)
1. for each vertex \( v \)
2. \( d[u][v] = 0 \)
3. \( Q = \) empty queue
4. \( f[u] = \) true
5. \( \text{enqueue}(Q, u) \)
6. while \( Q \) is not empty:
7. \( v = \text{dequeue}(Q) \)
8. for each \( w \) adjacent to \( v \)
9. \( d[v][w] = f[v] \)
\( f[w] = \) true
\( \text{enqueue}(Q, w) \)

\( O(n + m) \)

How many adjacent nodes will we ever visit? This is related to the number of edges. How many edges are there?

\( \sum_{v \in \text{vertices}} \text{deg}(v) = 2m \)

*Note: this is not per iteration of the while loop. This is the sum over all the while loops!

Time Complexity of BFS
(Using adjacency matrix)

Assume adjacency matrix
- \( n = \) number of vertices \( m = \) number of edges

Algorithm: BFS(u)

Input: \( u \) is the source vertex
Output: Mark all vertices that can be reached from \( u \)
1. for each vertex \( v \)
2. \( d[u][v] = 0 \)
3. \( f[u] = \) true
4. \( \text{queue}(Q, u) \)
5. \( w = \) dequeue(Q)
6. for each \( w \) adjacent to \( v \)
7. \( d[v][w] = f[v] \)
\( f[w] = \) true
\( \text{enqueue}(Q, w) \)

\( O(n^2) \)

So, adjacency matrix is not good for BFS!!

Path Recording
- BFS only tells us if a path exists from source \( s \) to other vertices \( v \).
  - It doesn’t tell us the path!
  - We need to modify the algorithm to record the path.
- Not difficult
  - Use an additional predecessor array \( \text{pred}[0..n-1] \)
  - \( \text{Pred}[w] = v \)
    - Means that vertex \( w \) was visited by \( v \)
**BFS + Path Finding**

**Algorithm BFS(v)**
1. for each vertex v
2. do if vis[v] := false;
3. \[ \text{pred}[v] := -1; \] 
   Set \( \text{pred}[v] \) to -1 (let -1 means no path to any vertex)
4. \[ Q := \text{empty queue}; \] 
5. \[ \text{flag}[s] := \text{true}; \]
6. \[ \text{enqueue}(Q, s); \]
7. while \( Q \) is not empty
8. do \( x := \text{dequeue}(Q); \)
9. for each \( w \) adjacent to \( x \)
10. do if \( \text{flag}[w] = \text{false} \)
11. then \[ \text{flag}[w] := \text{true}; \]
12. \[ \text{pred}[w] := x; \] 
   Record who visited \( w \)
13. \[ \text{enqueue}(Q, w); \]

**Example**

\[ Q = \{ \} \]
Initialize \( Q \) to be empty

Flag that 2 has been visited.
Place source 2 on the queue.
Example

\[ Q = \{2\} \rightarrow \{8, 1, 4\} \]

- Mark neighbors as visited.
- Record in Pred who was visited by 2.

 dequeue 2.
 Place all unvisited neighbors of 2 on the queue.

\[ Q = \{8, 1, 4\} \rightarrow \{1, 4, 0, 9\} \]

- Mark new visited neighbors.
- Record in Pred who was visited by 8.

 dequeue 8.
 Place all unvisited neighbors of 8 on the queue.
 Notice that 2 is not placed on the queue again, it has been visited!

\[ Q = \{1, 4, 0, 9\} \rightarrow \{4, 0, 9, 3, 7\} \]

- Place all unvisited neighbors of 1 on the queue.
- Only nodes 3 and 7 haven’t been visited yet.

 dequeue 1.
 Place all unvisited neighbors of 1 on the queue.

Example

\[ Q = \{4, 0, 9, 3, 7\} \rightarrow \{0, 9, 3, 7\} \]
Dequeue 4.
4 has no unvisited neighbors!

\[ Q = \{0, 9, 3, 7\} \rightarrow \{9, 3, 7\} \]
Dequeue 9.
9 has no unvisited neighbors!

\[ Q = \{9, 3, 7\} \rightarrow \{3, 7\} \]
Dequeue 3.
3 has no unvisited neighbors!
Example

$Q = \{3, 7\} \rightarrow \{7, 5\}$

Queue 3.

Mark new visited Vertex 5.

Place neighbor 5 on the queue.

Neighbors

$Q = \{7, 5\} \rightarrow \{5, 6\}$

Dequeue 7.

Mark new visited Vertex 6.

Place neighbor 6 on the queue.

Neighbors

$Q = \{5, 6\} \rightarrow \{6\}$

Dequeue 5.

-- no unvisited neighbors of 5.

Neighbors
Example

Q = \{6\} \rightarrow \{\}


Example

Q = \{\} \rightarrow \{\}

Pred now stores the path!

Pred array represents paths

Algorithm: 
1. if \text{pred}[\text{v}] \neq -1
2. then
3. \text{Path} (\text{pred}[\text{v}])
4. output =

Try some examples.
Path(0) \rightarrow
Path(6) \rightarrow
Path(1) \rightarrow
BFS tree

- We often draw the BFS paths as a m-ary tree, where s is the root.

Question: What would a “level” order traversal tell you?

Connected Component

- Connected component. Find all nodes reachable from s.

Flood Fill

- Flood fill. Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.
  - Node: pixel
  - Edge: two neighboring lime pixels
  - Blob: connected component of lime pixels

recolor lime green blob to blue
Flood Fill

- Flood fill. Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.
  - Node: pixel.
  - Edge: two neighboring lime pixels.
  - Blob: connected component of lime pixels.

**Flood Fill Diagram**

Connected Component

- Connected component. Find all nodes reachable from s.

More on Paths and trees in graphs
BFS

- Another way to think of the BFS tree is the physical analogy of the BFS Tree.
  Sphere-String Analogy: Think of the nodes as spheres and edges as unit length strings. Lift the sphere for vertex \( s \).

Sphere-String Analogy

```
  S
  a    b
  /    /
 s--
  d
```

bfs: Properties

- At some point in the running of BFS, \( Q \) only contains vertices/nodes at layer \( d \).
- If \( u \) is removed before \( v \) in BFS then
  \[ \text{dist}(u) \leq \text{dist}(v) \]
- At the end of BFS, for each vertex \( v \) reachable from \( s \), the \( \text{dist}(v) \) equals the shortest path length from \( s \) to \( v \).
BFS

Process nodes layer by layer

BFS: advancing wavefront

old wine in new bottle

forall \( v \in V \):
- \( \text{dist}(v) = \infty \); \( \text{prev}(v) = \text{null} \);
- \( \text{dist}(s) = 0 \)
Queue \( q \); \( q\).push(\( s \));
while (!\( Q\).empty())
  \( v = Q\).dequeue();
  for all \( e=(v,w) \) in \( E \)
    if \( \text{dist}(w) = \infty \):
      \( \text{dist}(w) = \text{dist}(v) + 1 \)
      \( Q\).enqueue(\( w \))
      \( \text{prev}(w) = v \)
dijkstra’s SSSP Alg
BFS With positive int weights

• for every edge e=(a,b) ∈ E, let \( w_e \) be the weight associated with it. Insert \( w_e - 1 \) dummy nodes between a and b. Call this new graph \( G' \).
  Run BFS on \( G' \). dist(\( u \)) is the shortest path length from s to node \( u \).
  • Why is this algorithm bad?

how do we speed it up?

• If we could run BFS without actually creating \( G' \), by somehow simulating BFS of \( G' \) on \( G \) directly.
  Solution: Put a system of alarms on all the nodes. When the BFS on \( G' \) reaches a node of \( G \), an alarm is sounded. Nothing interesting can happen before an alarm goes off.

an example
Another Example

alarm clock alg

alarm(s) = 0
until no more alarms
- wait for an alarm to sound. Let next alarm
  that goes off is at node v at time t.
  • dist(s,v) = 1
  • for each neighbor w of v in G:
    – If there is no alarm for w, alarm(w) = 1+weight(v,w)
    – If w’s alarm is set further in time than 1+weight(v,w),
      reset it to 1+weight(v,w).

recall bfs

forall v ∈ V:
  • dist(v) = ∞; prev(v) = null;
  • dist(s) = 0
Queue q; q.push(s);
while (!Q.empty())
  • v = Q.dequeue();
  • for all e=(v,w) in E
    • if dist(w) = ∞:
      • dist(w) = dist(v)+1
      • Q.enqueue(w)
      • prev(w)= v
dijkstra’s SSSP
for all v ∈ V:
   dist(v) = ∞; prev(v) = null;
   dist(s) = 0
Magic_DS Q; Q.insert(s, 0);
while (!Q.empty())
   v = Q.delete_min();
   for all e=(v,w) in E
      if dist(w) > dist(v)+weight(v,w) :
         – dist(w) = dist(v)+weight(v,w)
         – Q.insert(w, dist(w))
         – prev(w)=v

the magic ds: PQ
• What functions do we need?
  – insert() : Insert an element and its key. If
    the element is already there, change its
    key (only if the key decreases).
  – delete_min() : Return the element with the
    smallest key and remove it from the set.

Example

```
0 1 2 3 4 5 6 7 8
s    u    v    x    y
```

```
another view
region growth

1. Start from s
2. Grow a region R around s such that the SPT from s is known inside the region.
3. Add \( v \) to R such that \( v \) is the closest node to s outside R.
4. Keep building this region till R = V.
how do we find v?

Pick \( v \neq R \) s.t.

\[
\min_{x \in R} \text{dist}(s, x) + \omega(x, v)
\]

Let \((x^*, v^*)\) be the opt.

Example

\[x^* = x_L, \quad v^* = v_L\]

S, v*

Is this the shortest path to \( v^* \)?

Why?
old wine in new bottle

forall v ∈ V:
dist(v) = ∞; prev(v) = null;
dist(s) = 0
R = ∅;
while R ⊊ V
    Pick v not in R with smallest distance to s
    for all edges (v,z) ∈ E
        if(dist(z) > dist(v) + weight(v,z))
            dist(z) = dist(v) + weight(v,z)
            prev(z) = v;
    Add v to R

updates

Update rule:
(Best way to reach z?)

Running time?

delete_min = ?
insert = ?
Running time?

• If we used a linked list as our magic data structure?

\[
\begin{align*}
\text{delete}\_\text{min}() & \rightarrow \mathcal{O}(|V|) \\
\text{insert}() & \rightarrow \mathcal{O}(1)\mathcal{O}(|E|) \\
\text{Total} = |V| \text{ delete}\_\text{min}() \\
& + |E| \text{ insert}() = \mathcal{O}(\frac{|V|^2}{|E|}) \\
\end{align*}
\]

Binary Heap?

\[
\begin{align*}
\text{delete}\_\text{min}() & \rightarrow \mathcal{O}(|V|) \\
\text{insert}() & \rightarrow \mathcal{O}(|V|) \\
\text{Total} & \rightarrow \mathcal{O}(|E| \log |V|) \\
\end{align*}
\]
**d-ary heap**

- $\text{delete-min}() \rightarrow O\left(d \log dvight)$
- $\text{insert}() \rightarrow O\left(\log dv\right)$
- Total $\rightarrow O\left((dv + E) \log dv\right)$

---

**Fibonacci Heap**

- $\text{delete-min}() \rightarrow O\left(1\right)$ (Amortized)
- $\text{insert}() \rightarrow O\left(\log dv\right)$
- Total $\rightarrow O\left(dv \log dv + E\right)$

---

**a Spanning tree**

- Recall?
- Is it unique?
- Is shortest path tree a spanning tree?
- Is there an easy way to build a spanning tree for a given graph $G$?
- Is it defined for disconnected graphs?
Spanning tree

Connected subset of a graph G with n-1 edges which contains all of V.

spanning tree

A connected, undirected graph

Some spanning trees of the graph

easy algorithm

To build a spanning tree:

Step 1: T = one node in V, as root.
Step 2: At each step, add to tree one edge from a node in tree to a node that is not yet in the tree.
Spanning tree property

Adding an edge $e=(a,b)$ not in the tree creates a cycle containing only edge $e$ and edges in spanning tree.

Why?

Spanning tree property

• Let $c$ be the first node common to the path from $a$ and $b$ to the root of the spanning tree.
  The concatenation of $(a,b)$ $(b,c)$ $(c,a)$ gives us the desired cycle.

lemma 1

• In any tree, $T=(V,E)$,
  $|E| = |V| - 1$

Why?
Lemma 1

- In any tree, $T = (V,E)$,
  $|E| = |V| - 1$

Why?
- Tree $T$ with 1 node has zero edges.
- For all $n > 0$, $P(n)$ holds, where
  - $P(n)$: A tree with $n$ nodes has $n-1$ edges.
- Apply MI. How do we prove that given $P(m)$ true for all $1..m$, $P(m+1)$ is true?

Undirected Graphs n Trees

- An undirected graph $G = (V,E)$ is a tree iff
  1. It is connected
  2. $|E| = |V| - 1$

Lemma 2

Let $C$ be the cycle created in a spanning tree $T$ by adding the edge $e = (a,b)$ not in the tree. Then removing any edge from $C$ yields another spanning tree.

Why? How many edges and vertices does the new graph have? Can $(x,y)$ in $G$ get disconnected in this new tree?
**LEMMA 2**

- Let $T'$ be the new graph
- $T'$ has $n$ nodes and $n-1$ edges, so it must be a tree if it is connected.
- Let $(x,y)$ be not connected in $T'$. The only problem in the connection can be the removed edge $(a,b)$. But if $(a,b)$ was contained in the path from $x$ to $y$, we can use the cycle $C$ to reach $y$ (even if $(a,b)$ was deleted from the graph).

**weighted spanning trees**

Let $w_e$ be the weight of an edge $e$ in $G=(V,E)$.

Weight of spanning tree = Sum of edge weights.

Question: How do we find the spanning tree with minimum weight. This spanning tree is also called the Minimum Spanning Tree.

Is the MST unique?

**minimum spanning trees**

- Applications
  - networks
  - cluster analysis
    - used in graphics/pattern recognition
  - approximation algorithms (TSP)
  - bioinformatics/CFD
cut property

• Let $X$ be a subset of $V$. Among edges crossing between $X$ and $V \setminus X$, let $e$ be the edge of minimum weight. Then $e$ belongs to the MST.

• Proof?

cycle property

• For any cycle $C$ in a graph, the heaviest edge in $C$ does not appear in the MST.

Proof?

double chocolate question

• Is the SSSP Tree and the Minimum spanning tree the same?
• Is one the subset of the other always?
double chocolate question

• Is the SSSP Tree and the Minimum spanning tree the same?
• Is one the subset of the other always?

old wine in new bottle

forall v \in V:
\begin{align*}
\text{dist}(v) &= \infty; \text{prev}(v) = \text{null}; \\
\text{dist}(s) &= 0 \\
\text{Heap Q; } &\text{Q.insert}(s,0); \\
\text{while } (\text{Q.empty}()) &\\
\quad v &= \text{Q.delete_min}(); \\
\quad \text{for all } e=(v,w) \in E &\\
\quad &\text{if } \text{dist}(w) > \text{dist}(v)+\text{weight}(v,w) : \\
\quad &\quad \text{dist}(w) = \text{dist}(v)+\text{weight}(v,w) \\
\quad &\quad \text{Q.insert}(w, \text{dist}(w)) \\
\quad &\quad \text{prev}(w) = v
\end{align*}

a slight modification

jarnik’s or prim’s alg.

forall v \in V:
\begin{align*}
\text{dist}(v) &= \infty; \text{prev}(v) = \text{null}; \\
\text{dist}(s) &= 0 \\
\text{Heap Q; } &\text{Q.insert}(s,0); \\
\text{while } (\text{Q.empty}()) &\\
\quad v &= \text{Q.delete_min}(); \\
\quad \text{for all } e=(v,w) \in E &\\
\quad &\text{if } \text{dist}(w) > \text{dist}(v)+\text{weight}(v,w) : \\
\quad &\quad \text{dist}(w) = \text{dist}(v)+\text{weight}(v,w) \\
\quad &\quad \text{Q.insert}(w, \text{dist}(w)) \\
\quad &\quad \text{prev}(w) = v
\end{align*}
our first MST alg.
forall v ε V:
    dist(v) = ∞; prev(v) = null;
    dist(s) = 0
Magic_DS Q; Q.insert(s,0);
while (!Q.empty())
    v = Q.delete_min();
    for all e=(v,w) in E
        if dist(w) > weight(v,w) :
            – dist(w) = weight(v,w)
            – Q.insert(w, dist(w))
            – prev(w)= v

how does the running time depend on the magic_Ds?
• heap?
• insert()?
• delete_min()?
• Total time?
• What if we change the Magic_DS to fibonacci heap?

prim’s/jarnik’s algorithm
• best running time using fibonacci heaps
  – O(E + VlogV)
Why does it compute the MST?
another alg: KRushkal’s
• sort the edges of G in increasing order of weights
  Let S = {}
  for each edge e in G in sorted order
    – if the endpoints of e are disconnected in S
    – Add e to S

have u seen this before?
• Sort edges of G in increasing order of weight
• T = {} // Collection of trees
  For all e ∈ E
    – If T ∪ {e} has no cycles in T, then T = T ∪ {e}
  return T

Naïve running time O((|V|+|E|)|V|) = O(|E||V|)

how to speed it up?
• To O(E + VlogV)
  – Note that this is achieved by fibonacci heaps.
  – Surprisingly the idea is very simple.
3.4 Testing Bipartiteness

Bipartite Graphs

Def. An undirected graph $G = (V, E)$ is bipartite if the nodes can be colored red or blue such that every edge has one red and one blue end.

- Applications:
  - Stable marriage: men = red, women = blue.
  - Scheduling: machines = red, jobs = blue.
Testing Bipartiteness

Testing bipartiteness: Given a graph G, is it bipartite?
- Many graph problems become:
  - easier if the underlying graph is bipartite (matching)
  - tractable if the underlying graph is bipartite (independent set)
- Before attempting to design an algorithm, we need to understand structure of bipartite graphs.

An Obstruction to Bipartiteness

- Lemma. If a graph G is bipartite, it cannot contain an odd length cycle.
- Pf. Not possible to 2-color the odd cycle, let alone G.

Bipartite Graphs

- Lemma. Let G be a connected graph, and let $L_0, \ldots, L_n$ be the layers produced by BFS starting at node s. Exactly one of the following holds.
  1. No edge of G joins two nodes of the same layer, and G is bipartite.
  2. An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).
Bipartite Graphs

- Lemma. Let $G$ be a connected graph, and let $L_0, ..., L_k$ be the layers produced by BFS starting at node $s$. Exactly one of the following holds.
  1. No edge of $G$ joins two nodes of the same layer, and $G$ is bipartite.
  2. An edge of $G$ joins two nodes of the same layer, and $G$ contains an odd-length cycle (and hence is not bipartite).

- Proof:
  - Suppose no edge joins two nodes in the same layer.
  - By previous lemma, this implies all edges join nodes on same level.
  - Bipartition: red = nodes on odd levels, blue = nodes on even levels.

Obstruction to Bipartiteness

- Corollary. A graph $G$ is bipartite iff it contain no odd length cycle.
3.5 Connectivity in Directed Graphs

Directed Graphs

- Directed graph. $G = (V, E)$
  - Edge $(u, v)$ goes from node $u$ to node $v$.

- Ex. Web graph - hyperlink points from one web page to another.
  - Directedness of graph is crucial.
  - Modern web search engines exploit hyperlink structure to rank web pages by importance.

Graph Search

- Directed reachability. Given a node $s$, find all nodes reachable from $s$.
- Directed $s$-$t$ shortest path problem. Given two nodes $s$ and $t$, what is the length of the shortest path between $s$ and $t$?
- Web crawler. Start from web page $s$. Find all web pages linked from $s$, either directly or indirectly.
Strong Connectivity

- Def. Node u and v are mutually reachable if there is a path from u to v and also a path from v to u.
- Def. A graph is strongly connected if every pair of nodes is mutually reachable.
- Lemma. Let s be any node. G is strongly connected iff every node is reachable from s, and s is reachable from every node.
- Pf. ⇒ Follows from definition.
- Pf. ⇐ Path from u to v: concatenate u-s path with s-v path.
  Path from v to u: concatenate v-s path with s-u path.
  ok if paths overlap.

Strong Connectivity: Algorithm

- Theorem. Can determine if G is strongly connected in O(m + n) time.
- Pf.
  - Pick any node s.
  - Run BFS from s in G.
  - Run BFS from s in G^rev.
  - Return true iff all nodes reached in both BFS executions.
  - Correctness follows immediately from previous lemma.

3.6 DAGs and Topological Ordering

To be continued.