Last Time

- Greedy Algorithms for Optimization
- Lab 10: Exam Scheduling
- Adjacency List Implementation Details
Today’s Outline

• GraphList Wrap-up
• An Important Algorithm: Minimum-cost spanning subgraph
GraphListVertex Iterators

// Iterator for incident edges
public Iterator<Edge<V,E>> adjacentEdges() {
    return adjacencies.iterator();
}

// Iterator for adjacent vertices
public Iterator<V> adjacentVertices() {
    return new GraphListAIterator<V,E>(adjacentEdges(), label());
}

GraphListAIterator creates an Iterator over vertices based on the Iterator over edges produced by adjacentEdges()
GraphListAIterator

GraphListAIterator uses two instance variables

protected AbstractIterator<Edge<V,E>> edges;
protected V vertex;

public GraphListAIterator(Iterator<Edge<V,E>> i, V v) {
    edges = (AbstractIterator<Edge<V,E>>)i;
    vertex = v;
}

public V next() {
    Edge<V,E> e = edges.next();
    if (vertex.equals(e.here()))
        return e.there();
    else { // could be an undirected edge!
        return e.here();
    }
}
GraphListEIterator uses one instance variable

```java
protected AbstractIterator<Edge<V,E>> edges;
```

GraphListEIterator

• Takes the Map storing the vertices
• Uses it to build a linked list of all edges
• Gets an iterator for this linked list and stores it, using it in its own methods
To implement GraphList, we use the GraphListVertex (GLV) class

GraphListVertex class
- Maintain linked list of edges at each vertex
- Instance vars: label, visited flag, linked list of edges

GraphList abstract class
- Instance vars:
  - Map<V,GraphListVertex<V,E>> dict; // label -> vertex
  - boolean directed; // is graph directed?

How do we implement key GL methods?
- GraphList(), add(), getEdge(), …
protected GraphList(boolean dir) {
    dict = new Hashtable<V, GraphListVertex<V, E>>();
    directed = dir;
}

public void add(V label) {
    if (dict.containsKey(label)) return;
    GraphListVertex<V, E> v = new GraphListVertex<V, E>(label);
    dict.put(label, v);
}

public Edge<V, E> getEdge(V label1, V label2) {
    Edge<V, E> e = new Edge<V, E>(get(label1),
                                   get(label2), null, directed);
    return dict.get(label1).getEdge(e);
}
GraphListDirected

- GraphListDirected (GraphListUndirected) implements the methods requiring different treatment due to (un)directedness of edges
  - addEdge, remove, removeEdge, …
// addEdge in GraphListDirected.java
// first vertex is source, second is destination
public void addEdge(V vLabel1, V vLabel2, E label) {
    // first get the vertices
    GraphListVertex<V,E> v1 = dict.get(vLabel1);
    GraphListVertex<V,E> v2 = dict.get(vLabel2);
    // create the new edge
    Edge<V,E> e = new Edge<V,E>(v1.label(), v2.label(), label, true);
    // add edge only to source vertex linked list (aka adjacency list)
    v1.addEdge(e);
}
public V remove(V label) {
    //Get vertex out of map/dictionary
    GraphListVertex<V,E> v = dict.get(label);

    //Iterate over all vertex labels (called the map “keyset”)
    Iterator<V> vi = iterator();
    while (vi.hasNext()) {
        //Get next vertex label in iterator
        V v2 = vi.next();

        //Skip over the vertex label we're removing
        //(Nodes don't have edges to themselves...)
        if (!label.equals(v2)) {
            //Remove all edges to "label"
            //If edge does not exist, removeEdge returns null
            removeEdge(v2,label);
        }
    }

    //Remove vertex from map
    dict.remove(label);
    return v.label();
}
public E removeEdge(V vLabel1, V vLabel2) {
    //Get vertices out of map
    GraphListVertex<V,E> v1 = dict.get(vLabel1);
    GraphListVertex<V,E> v2 = dict.get(vLabel2);

    //Create a “temporary” edge connecting two vertices
    Edge<V,E> e = new Edge<V,E>(v1.label(), v2.label(), null, true);

    //Remove edge from source vertex linked list
    e = v1.removeEdge(e);
    if (e == null) return null;
    else return e.label();
}
Efficiency Revisited

• Assume Map operations are $O(1)$ (for now)
  • $|E| =$ number of edges
  • $|V| =$ number of vertices
• Runtime of add, addEdge, getEdge, removeEdge, remove?
• Space usage?
• Conclusions
  • Matrix is better for dense graphs
  • List is better for sparse graphs
  • For graphs “in the middle” there is no clear winner
## Efficiency: Assuming Fast Map

<table>
<thead>
<tr>
<th>Operation</th>
<th>Matrix</th>
<th>GraphList</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>addEdge</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>getEdge</td>
<td>$O(1)$</td>
<td>$O(</td>
</tr>
<tr>
<td>removeEdge</td>
<td>$O(1)$</td>
<td>$O(</td>
</tr>
<tr>
<td>remove</td>
<td>$O(</td>
<td>V</td>
</tr>
<tr>
<td>space</td>
<td>$O(</td>
<td>V</td>
</tr>
</tbody>
</table>
Applications
Minimum-Cost Spanning Trees
Minimum-Cost Spanning Trees
Basic Graph Properties

• A subgraph of a graph $G=(V, E)$ is a graph $G'=(V', E')$ where
  • $V' \subseteq V$
  • $E' \subseteq E$, and
  • If $e \in E'$ where $e = \{u, v\}$, then $u, v \in V'$

• Special Subgraphs
  • If $E'$ contains every edge of $E$ having both ends in $V'$, then $G'$ is called the subgraph of $G$ induced by $V'$
  • If $V' = V$, then $G'$ is called a spanning subgraph of $G$
Basic Graph Properties

• Recall: An undirected graph $G=(V,E)$ is connected if for every pair $u,v$ in $V$, there is a path from $u$ to $v$ (and so from $v$ to $u$)
• The maximal sized connected subgraphs of $G$ are called its connected components
  • Note: They are induced subgraphs of $G$
• An undirected graph without cycles is a forest
• A connected forest is called a tree.
  • Not to be confused with the data structure!
Facts About Graphs

Thm: If $G=(V,E)$ is a forest with $|E| > 0$, then $G$ has at least one vertex $v$ of degree 1 (a leaf)

- Let’s prove this: Consider a longest simple path in $G$...

Thm: If $G=(V,E)$ is a tree then $|E| = |V| - 1$.

- Hint: Induction on $v$: delete a leaf

Thm: Every connected graph $G=(V,E)$ contains a spanning subgraph $G’=(V,E’)$ that is a tree

- That is, a spanning tree

Proof idea:

- If $G$ is not a tree, then it contains a cycle $C$
- Removing an edge from $C$ leaves $G$ connected (why)
- Repeat until no more cycles remain
Edge-Weighted Graphs

- An *edge-weighting* of a graph $G=(V,E)$ is an assignment of a number (weight) to each edge of $G$
  - We write the weight of $e$ as $w(e)$ or $w_e$

- The weight $w(G')$ of any subgraph $G'$ of $G$ is the sum of the weights of the edges in $G'$

- We will focus on edge-weights that are non-negative, so if $G'$ is a subgraph of $G$, then $w(G') \leq w(G)$
A Famous Problem

• Given a connected, undirected graph $G = (V, E)$ with non-negative edge weights, find a minimum-weight, connected, spanning subgraph of $G$.

• Note: Such a subgraph must be a spanning tree!

• Frequently, we refer to the edge weights as costs and so this problem becomes:

• Given an undirected graph $G$ with edge costs, compute a minimum-cost spanning tree of $G$. 
Minimum-Cost Spanning Trees

Diagram of a minimum-cost spanning tree.
Minimum-Cost Spanning Trees
Finding a MCST

Suppose we just wanted to find a PCST (pretty cheap spanning tree), here’s one idea:

Grow It Greedily!

- Pick a vertex and find its cheapest incident edge. Now we have a (small) tree
- Repeatedly add the cheapest edge to the tree that keeps it a tree (connected, no cycles)
- This method is called Prim’s Algorithm
- How close might this get us to the MCST?
An Amazing Fact

Thm: (Prim 1957) The greedy tree-growing algorithm always finds a minimum-cost spanning tree for any connected graph.

Contrast this with the greedy exam scheduling algorithm, which does not always find a minimum schedule (coloring).

Why does this work?
Def: Sets $V_1$ and $V_2$ form a *partition* of a set $V$ if $V_1 \cup V_2 = V$ and $V_1 \cap V_2 = \emptyset$

Lemma: Let $G=(V,E)$ be a connected graph and let $V_1$ and $V_2$ be a partition of $V$. Every MCST of $G$ contains a cheapest edge between $V_1$ and $V_2$

• Let $e$ be a cheapest edge between $V_1$ and $V_2$

• Let $T$ be a MCST of $G$. If $e \notin T$, then $T \cup \{e\}$ contains a cycle $C$ and $e$ is an edge of $C$

• Some other edge $e'$ of $C$ must also be between $V_1$ and $V_2$; $e$ is a cheapest edge, so $w(e') = w(e)$ [Why?]