Last Time

• Graph Data Structures: Implementation
  • Adjacency Array Implementation Details
    • Featuring many Iterators!
Today’s Outline

• Greedy Algorithms for Optimization
• Lab 11: Exam Scheduling
  • Defining the problem
  • Sketching a design
• Adjacency List Implementation Details
• More Fundamental Graph Properties
• An Important Algorithm: Minimum-cost spanning subgraph
Lab 11 Overview: Graph Algorithms using structure5
Greedy Algorithms

- A greedy algorithm attempts to find a globally optimum solution to a problem by making locally optimum (greedy) choices.

Example: Graph Coloring
- A (proper) coloring of a graph $G = (V,E)$ is an assignment of a value (color) to each vertex so that adjacent vertices get different values (colors).
- Typically one strives to minimize the number of colors used.
Greedy Coloring : Math

Here’s a greedy coloring algorithm

Build a collection $C = \{C_1, ..., C_k\}$ of sets of vertices

$i = 0; \; C_i = \{\} \; // \text{empty set}$

while $G$ has more vertices

for each vertex $u$ in $G$

if $u$ is not adjacent to any vertex of $C_i$

remove $u$ from $G$ and add $u$ to $C_i$

add $C_i$ to $C$

$i++;$

Return $C$ as the coloring
Here’s a greedy coloring algorithm

Create a structure C to hold a collection of lists

while G is not empty

    pick a vertex v in G; create an empty list L; add v to L

    for each vertex \( u \neq v \) in G

        if u is not adjacent to any vertex of L
            add u to L

    remove all vertices of L from G

    add L to C

Return C as the coloring
Greedy Coloring
Greedy Coloring

Some observations

• Each list (color class) L is a set of vertices no two of which are adjacent (an independent set)

• Each color class is maximal: cannot be made any larger
  • The hope is that this results in fewer colors being needed
  • But the solution is not always optimum!
  • This is a very hard problem

• The coloring problem is the same as finding a partition of the vertex set into independent sets
  • Partition means union of disjoint sets
Lab 11 : Exam Scheduling

Find a schedule (set of time slots) for exams so that

• No student has two exams in the same slot
• Every course is in a slot
• The number of slots is as small as possible

This is just the graph coloring problem in disguise!

• Each course is a vertex
• Two vertices are adjacent if the courses share students
• A slot must be an independent set of vertices (that is, a color class)
Lab 11 Notes: Using Graphs

• Create a new graph in structure5
  • GraphListDirected, GraphListUndirected,
  • GraphMatrixDirected, GraphMatrixUndirected

• Graph<V,E> conflictGraph = new GraphListUndirected<V,E>();
Lab 11: Useful Graph Methods

- `void add(V label)`
  - add vertex to graph
- `void addEdge(V vtx1, V vtx2, E label)`
  - add edge between vtx1 and vtx2
- `Iterator<V> neighbors(V vtx1)`
  - Get iterator for all neighbors to vtx1
- `boolean isEmpty()`
  - Returns true iff graph is empty
- `Iterator<V> iterator()`
  - Get vertex iterator
- `V remove(V label)`
  - Remove a vertex from the graph
- `E removeEdge(V vLabel1, V vLabel2)`
  - Remove an edge from graph
The vertices are stored in an array $V[]$
$V[]$ contains a linked list of edges having a given source
Adjacency List: Undirected Graph

The vertices are stored in an array $V[]$
$V[]$ contains a linked list of edges incident to a given vertex
GraphList

• Maintain an *adjacency list of edges* at each vertex (no adjacency matrix)
  • Keep only outgoing edges for directed graphs
• Support both directed and undirected graphs
  (GraphListDirected, GraphListUndirected)
Vertex and GraphListVertex

• We use the same Edge class for all graph types
• We extend Vertex to include an Edge list
• GraphListVertex class adds to Vertex class
  • A Structure to store edges adjacent to the vertex
    protected Structure<Edge<V,E>> adjacencies; // adjacent edges
    – adjacencies is created as a SinglyLinkedList of edges
• Several methods

  public void addEdge(Edge<V,E> e)
  public boolean containsEdge(Edge<V,E> e)
  public Edge<V,E> removeEdge(Edge<V,E> e)
  public Edge<V,E> getEdge(Edge<V,E> e)
  public int degree()
  // and methods to produce Iterators...
public GraphListVertex(V key) {
    super(key); // init Vertex fields
    adjacencies = new SinglyLinkedList<Edge<V,E>>();
}

public void addEdge(Edge<V,E> e) {
    if (!containsEdge(e)) adjacencies.add(e);
}

public boolean containsEdge(Edge<V,E> e) {
    return adjacencies.contains(e);
}

public Edge<V,E> removeEdge(Edge<V,E> e) {
    return adjacencies.remove(e);
}
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

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

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

```java
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

GraphListEIterator uses one instance variable

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
GraphList

• 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();
}