CSCI 136
Data Structures &
Advanced Programming

Lecture 30
Fall 2018
Instructors: Bills are Back
Last Time

- Graph Data Structures: Implementation
  - Adjacency Array Implementation Details
    - GraphMatrix Abstract Base Class
Today’s Outline

• GraphMatrixDirected Implementation
• Greedy Algorithms for Optimization
• Lab 10: Exam Scheduling
  • Defining the problem
  • Sketching a design
• Adjacency List Implementation Details
• More Fundamental Graph Properties
• An Important Algorithm: Minimum-cost spanning subgraph
GraphMatrixDirected

• Completes the implementation of GraphMatrix to ensure graph is directed
• GraphMatrixUndirected is very similar…
• How do we implement GraphMatrixDirected?
  • We’ll discuss some methods
  • Read Ch 16 for complete details…
GraphMatrixDirected

• Constructor

public GraphMatrixDirected(int size) {
    // pre: size > 0
    // post: constructs an empty graph that may be
    // expanded to at most size vertices. Graph
    // is directed if dir true and undirected
    // otherwise

    // call GraphMatrix constructor
    super(size,true);
}


GraphMatrixDirected

- addEdge
  
  // pre: vLabel1 and vLabel2 are labels of existing vertices
  public void addEdge(V vLabel1, V vLabel2, E label) {
      GraphMatrixVertex<V> vtx1, vtx2;
      vtx1 = dict.get(vLabel1);
      vtx2 = dict.get(vLabel2);
      Edge<V,E> e = new Edge<V,E>(vtx1.label(), vtx2.label(),
                                  label, true);
      data[vtx1.index()][vtx2.index()] = e;
  }
GraphMatrixDirected

- removeEdge
  
  // pre: vLabel1 and vLabel2 are labels of existing vertices
  public E removeEdge(V vLabel1, V vLabel2) {
    // get indices
    int row = dict.get(vLabel1).index();
    int col = dict.get(vLabel2).index();
    // cache old value
    Edge<V,E> e = (Edge<V,E>)data[row][col];
    // update matrix
    data[row][col] = null;
    if (e == null) return null;
    else return e.label(); // return old value
  }
GraphMatrix Efficiency

- 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 good for dense graphs
  - Have to commit to maximum # of vertices in advance
## Efficiency: Assuming Fast Map

<table>
<thead>
<tr>
<th>Operation</th>
<th>GraphMatrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>addEdge</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>getEdge</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>removeEdge</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>remove</td>
<td>$O(</td>
</tr>
<tr>
<td>space</td>
<td>$O(</td>
</tr>
</tbody>
</table>
Lab 10 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
Here’s a greedy coloring algorithm

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

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

while G is 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
Greedy Coloring : CS

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 ≠ 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 10 : 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 10 Notes: Using Graphs

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

- Graph<V,E> conflictGraph = new GraphListUndirected<V,E>();
Lab 10: 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
Adjacency List: Directed 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 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();
    }
}
GraphListElterator

GraphListElterator uses one instance variable

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

GraphListElterator
• 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