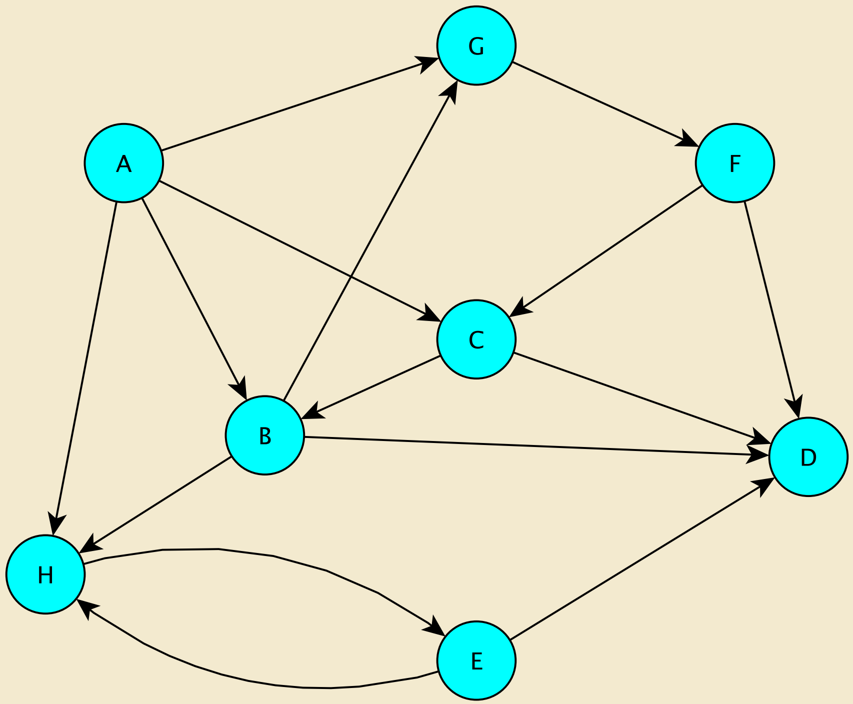
CSCI 136
Data Structures &
Advanced Programming

Graph Implementations I

Outline

- Recall : Adjacency Matrix of a Graph
- Recall : Graph Interface
- GraphMatrix implementation

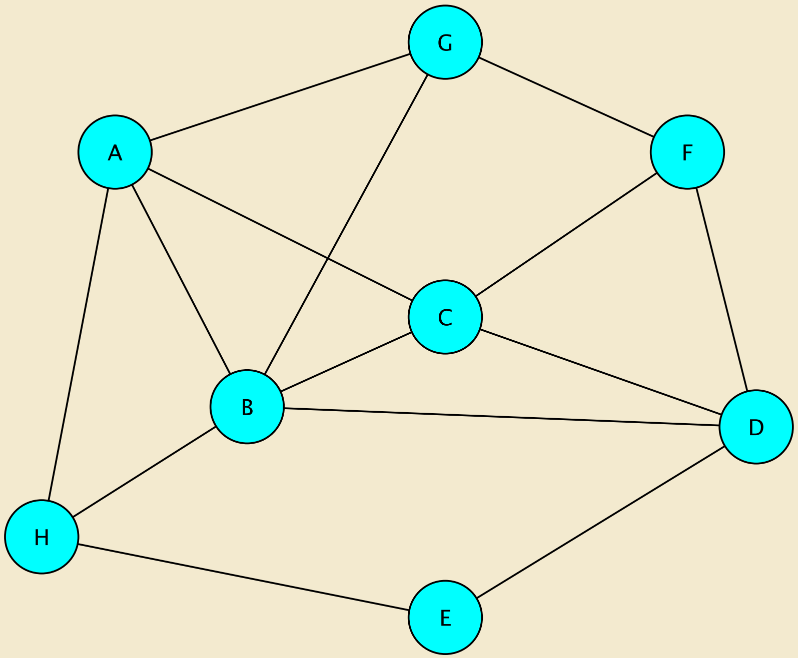
Adjacency Array: Directed Graph



	A	B	C	D	E	F	G	H
A	0	1	1	0	0	0	1	1
B	0	0	0	1	0	0	1	1
C	0	1	0	1	0	0	0	0
D	0	0	0	0	0	0	0	0
E	0	0	0	1	0	0	0	1
F	0	0	1	1	0	0	0	0
G	0	0	0	0	0	1	0	0
H	0	0	0	0	1	0	0	0

Entry (i,j) stores 1 if there is an edge from i to j; 0 otherwise
E.G.: edges(C,B) = 1 but edges(B,C) = 0

Adjacency Array: Undirected Graph



	A	B	C	D	E	F	G	H
A	0	1	1	0	0	0	1	1
B	1	0	1	1	0	0	1	1
C	1	1	0	1	0	1	0	0
D	0	1	1	0	1	1	0	0
E	0	0	0	1	0	0	0	1
F	0	0	1	1	0	0	1	0
G	1	1	0	0	0	1	0	0
H	1	1	0	0	1	0	0	0

Entry (i,j) store 1 if there is an edge between i and j; else 0
E.G.: edges(B,C) = 1 = edges(C,B)

Aside : Adjacency Array Optimization

Halving the Space (***not in structure5***)

	0	1	2	3	4	5	6
0	0	1	1	0	0	0	1
1	1	0	1	1	0	0	1
2	1	1	0	1	0	1	0
3	0	1	1	0	1	1	0
4	0	0	0	1	0	0	0
5	0	0	1	1	0	0	1
6	1	1	0	0	0	1	0

	0	1	2	3	4	5	6
0	0	1	1	0	0	0	1
1		0	1	1	0	0	1
2			0	1	0	1	0
3				0	1	1	0
4					0	0	0
5						0	1
6							0

0	1	2	3	4	5	6	7	8	9	…																	
0	1	1	0	0	0	1	0	1	1	0	0	1	0	1	0	1	0	0	1	1	0	0	0	0	0	1	0

(r,c) maps to $n \cdot r + c - \frac{r(r+1)}{2}$ (here n = 7)

Recall : Graph Interface

- Supports storing a value at each vertex and edge
  - Called a *label*
  - V : vertex label type
  - E : edge label type
- Supports methods for
  - get vertex/edge value
  - adding/removing vertices/edges
  - searching for vertex/edge labels
  - changing/querying 'visited' state of vertices/edges
  - producing iterators to vertices, neighbors, edges

Graph Interface Methods

- void add(V vtx), V remove(V vtx)
  - Add/remove vertex to/from graph
    - remove(vtx) also removes *all* edges containing vtx
- void addEdge(V vtx1, V vtx2, E edgeLabel),
  E removeEdge(V vtx1, V vtx2)
  - Add/remove edge between vtx1 and vtx2
- boolean containsEdge(V vtx1, V vtx2)
  - Returns true iff there is an edge between vtx1 and vtx2
- Edge<V,E> getEdge(V vtx1, V vtx2)
  - Returns edge between vtx1 and vtx2 (or null if no edge)
- void clear()
  - Remove all nodes and edges from graph

Graph Interface Methods

- boolean visit(V vertexLabel)
  - Mark vertex as "visited" and return *previous* value of visited flag
- boolean visitEdge(Edge<V,E> e)
  - Mark edge as "visited"
- boolean isVisited(V vtx), boolean isVisitedEdge(Edge<V,E> e)
  - Returns true iff vertex/edge has been visited
- Iterator<V> neighbors(V vtx1)
  - Get iterator for all neighbors of vtx1
  - For directed graphs, out-edges only
- Iterator<V> iterator()
  - Get vertex iterator
- void reset()
  - Remove visited flags for all nodes/edges

Implementing the Matrix Model

What we'll want

- Edge objects : store edge label, 2 vertex labels, …
- A 2-D array (adjacency matrix) of edge objects
- A way to store vertex labels
- A way to convert from vertex labels to matrix row/column indices
- A way to keep track of unused rows/columns so that new vertices can be added
    - Note: Max. number of vertices is specified at creation

Edge Class : Description

- Graph *edges* are defined in their own public class

  `Edge<V,E>(V vtx1, V vtx2,  E label, boolean directed)`

  - Construct a (possibly directed) edge between the two vertices having labels vtx1 and vtx2

- Useful methods:

  `E label()` : returns edge label

  `V here(), there()` : returns vertex label

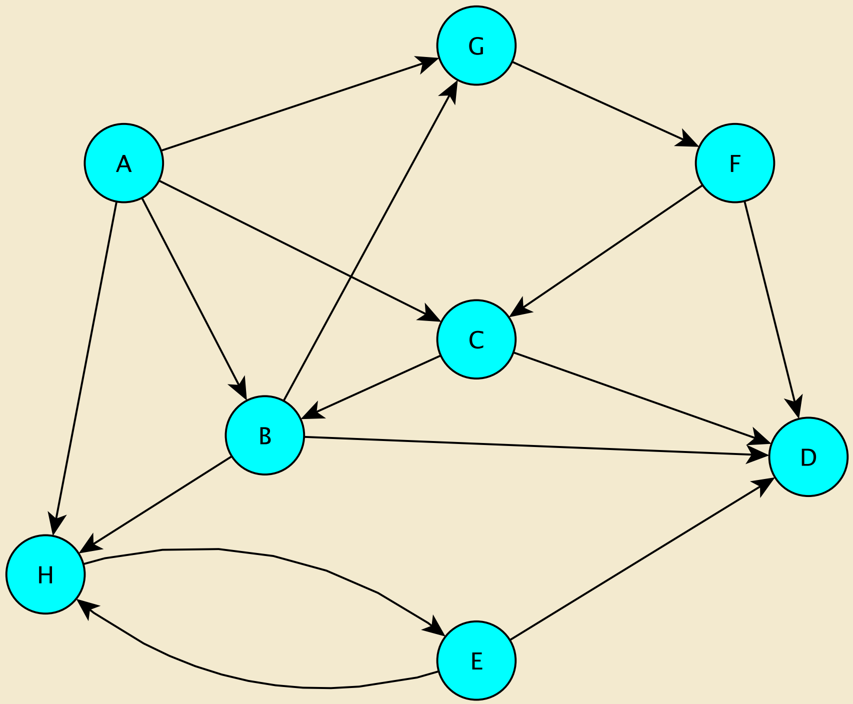
  `void setLabel()` : updates edge label

  `boolean visit(), isVisited(), isDirected()`

  - visit returns old value of visited flag

  `reset()` : resets visited flag to false

Adjacency Array: Directed Graph



	0	1	2	3	4	5	6	7
0	0	1	1	0	0	0	1	1
1	0	0	0	1	0	0	1	1
2	0	1	0	1	0	0	0	0
3	0	0	0	0	0	0	0	0
4	0	0	0	1	0	0	0	1
5	0	0	1	1	0	0	0	0
6	0	0	0	0	0	1	0	0
7	0	0	0	0	1	0	0	0

A	B	C	D	E	F	G	H
0	1	2	3	4	5	6	7

Vertex look-up dictionary

Implementing the Matrix Model

What we'll want

- Edge objects : store edge label, 2 vertex labels, …
- A 2-D array (adjacency matrix) of edge objects
- *A way to store vertex labels*
- *A way to convert from vertex labels to matrix row/column indices*
- A way to keep track of unused rows/columns so that new vertices can be added
    - Note: Max. number of vertices is specified at creation

Vertex and GraphMatrixVertex

- Vertex class stores vertex info (label, …)
  - Unlike the Edge class, Vertex class **is not public**
  - Useful Vertex methods:
    - `V label()` : returns vertex label
    - `boolean visit()` : returns previous visited value
    - `boolean isVisited()`
    - `void reset()` : resets visited flag to false
- GraphMatrixVertex class extends Vertex class
  - Adds one more useful attribute
    - Index of node (int) in adjacency matrix

      `int index()`
  - Note : We only need one int to represent index
- In these slides, we write GMV for GraphMatrixVertex

Choosing a Dictionary Structure

- We want to be able to retrieve the info stored in a vertex given the vertex label
  - Allows retrieval of row/column index given vertex label
- Many choices
  - Vector of associations:
    - Vector<Association<V, GraphMatrixVertex<V>>>
  - Ordered Vector of Associations
  - BinarySearchTree of Associations
- Constraint: Vertices should have unique labels
  - Otherwise, distinct edges may have same vertex labels!
- We'll use the Map Interface
  - Maps require a unique key for each entry

Implementing the Matrix Model

What we'll want

- Edge objects : store edge label, 2 vertex labels, …
- A 2-D array (adjacency matrix) of edge objects
- A way to store vertex labels
- A way to convert from vertex labels to matrix row/column indices
- *A way to keep track of unused rows/columns so that new vertices can be added*
    - Note: Max. number of vertices is specified at creation

Managing Unused Rows/Columns

- We want to be able to retrieve an available row/column when adding a new vertex
- Only operations required are add/remove
- Use SinglyLinkedList (or StackList)
  - Both operations are O(1)

Implementing the Matrix Model

```
public abstract class GraphMatrix<V,E>
     implements Graph<V,E>
```

- Abstract –implements majority of Graph and contains
  - `Map<V, GraphMatrixVertex<V>> dict`
    - Allows for retrieving vertex index and full vertex label from (partial) vertex label
  - `SinglyLinkedList<Integer> freeList`
    - Stores unused row/column indices from adjacency array
  - `Object[][] data`
    - Stores Edge<V,E> objects
  - `protected int size;  //max size of matrix`
  - `protected boolean directed;`
- Handles code that doesn't depend on directedness of edges

GraphMatrix Constructor
(Yes, abstract classes can have constructors!)

```
protected GraphMatrix(int size, boolean dir) {
    this.size = size; // set maximum size
    directed = dir; // fix direction of edges

    // the following constructs a size x size matrix
    // (the “Objects” will be “Edges”)
    // (can’t use generics with arrays!)
    data = new Object[size][size];

    // label→vertex info translation table
    dict = new Hashtable<V,GraphMatrixVertex<V>>(size);

    // put all indices in the free list
    freeList = new SinglyLinkedList<Integer>();
    for (int row = size-1; row >= 0; row--)
        freeList.add(new Integer(row));
}
```

GraphMatrix add()

```
 public void add(V label) {
      // if there already, do nothing
      if (dict.containsKey(label)) return;

      Assert.pre(!freeList.isEmpty(), "Matrix not full");
      // allocate a free row and column
      int row = freeList.removeFirst().intValue();
       // Note: intValue() was required when class was written
      // add vertex to dictionary
      dict.put(label, new GraphMatrixVertex<V>(label, row));
}
```

GraphMatrix remove()

```
public V remove(V label) {
        // find and extract vertex
        GraphMatrixVertex<V> vert = dict.remove(label);
        // If vert is null, no such vertex in graph
        if (vert == null) return null;
        // remove vertex from matrix
        int index = vert.index();
        // clear row and column entries
        for (int row=0; row<size; row++) {
            data[row][index] = null;
            data[index][row] = null;
        }
        // add node index to free list
        freeList.add(new Integer(index));
        return vert.label();
}
```

Neighbors Iterator : GraphMatrix

Neighbors Iterator

```
public Iterator<V> neighbors(V label) {
      GraphMatrixVertex<V> vert = dict.get(label);
      List<V> list = new SinglyLinkedList<V>();
      for (int row=size-1; row>=0; row--) {
            Edge<V,E> e = (Edge<V,E>)data[vert.index()][row];
            if (e != null)
                  if (e.here().equals(vert.label()))
                        list.add(e.there());
                        else list.add(e.here());
      }
      return list.iterator();
   }
```

GraphMatrixDirected

GraphMatrix does not implement methods that depend on edge directions.

This is done by the GraphMatrixDirected class

- Completes the implementation of GraphMatrix to ensure graph is directed
- GraphMatrixUndirected is very similar…
- How do we implement GraphMatrixDirected?
  - We'll discuss some methods
  - Refer to source code or Ch 16 for further 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;
}
```

Why do we get the vertex labels if we already know them?!

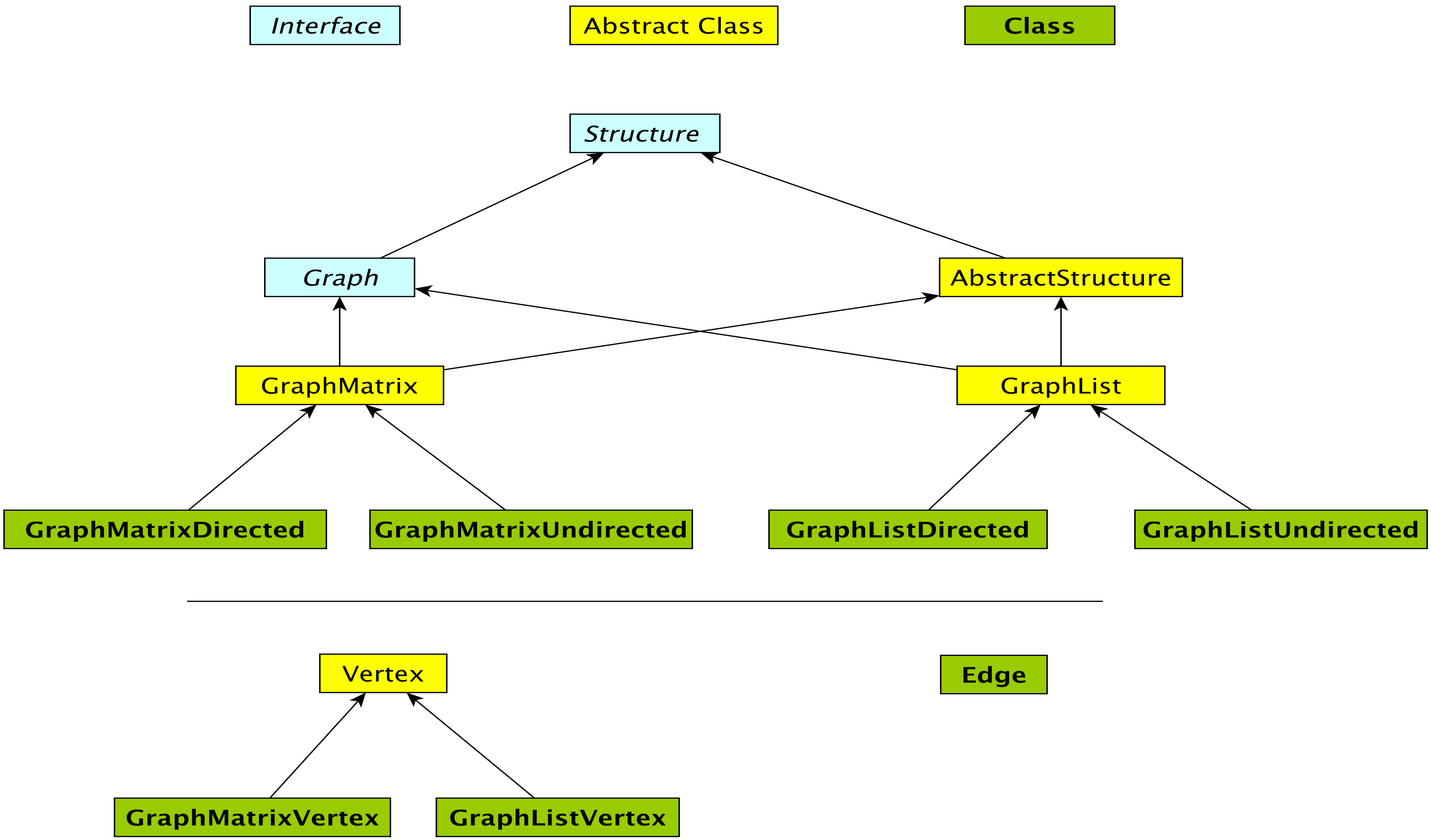
- vLabel1/vLabel2 may only contain *partial* label information
  - A vertex label may be an association that uses the key value for equality testing

GraphMatrixDirected

- removeEdge

```
// pre: vLabel1 and vLabel2 are labels of existing vertices
public E removeEdge(V vLabel1, 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
}
```

Graph Classes in structure5



GraphMatrix Efficiency
(Assuming O(1) time Map Ops)

For a directed graph G = (V,E)

- |E| = number of edges (often folks write m = |E|)
- |V| = number of vertices (often folks write n = |V|)

	GraphMatrix
add	
addEdge	
getEdge	
removeEdge	

GraphMatrix Efficiency (Assuming O(1) time Map Ops)

	GraphMatrix
degree	
remove	
iterator	
neighbors	
edges	
space	

Summary & Observations

- Assuming Map operations are O(1)
  - Adding a vertex or edge, and removing an edge take O(1) time
  - Operations that depend on traversing portions of the 2-D array of edges take longer
    - Finding vertex degree, neighbors, removing a vertex, …
- Conclusions
  - Matrix is good for dense graphs
  - But: Need to commit to maximum # of vertices in advance

- Up Next : Linked List Implementations