CSCI 136
Data Structures &
Advanced Programming

Lecture 30

Spring 2018

Instructors: Bill $\rightarrow$ Jon

Last Time

- Introduction To Graphs
  - Definitions and Properties: Undirected Graphs
  - Small Proofs
  - Reachability

Today's Outline

- Graphs in Structure5
  - Graph Interface
- Using the Graph interface to implement graph algorithms:
  - BFS + DFS
- Lab 10 Preview: Graph Coloring to schedule exams

Graphs in Structure5

- Implementation involves a number of design decisions, depending on intended uses
  - What kinds of graphs will be available?
    - Undirected, directed, mixed
  - What underlying data structures will be used?
  - What functionality will be provided?
  - What aspects will be public/protected/private
- We'll focus on popular implementations for undirected and directed graphs (separately)

Graphs in structure5

- We want to store information at vertices and at edges, but we favor vertices
  - Let V and E represent the types of information held by vertices and edges respectively
  - Interface Graph<V,E> extends Structure<V>
    - Vertices are the building blocks; edges depend on them
- Type V holds a *label* for a (hidden) vertex
- Type E holds a *label* for an (available) edge
  - Label: Application-specific data for a vertex/edge

Graphs in structure5

- So, the methods described in the Structure interface are about vertices (but also impact edges: e.g., `clear()`)
- We'll want to add a number of similar methods to provide information about edges, and the graph itself
  - Ultimately the `Structure` interface is a subset of the total functionality in the graph classes

Recall: Desired Functionality

- What are the basic operations we need in order to describe algorithms on graphs?
  - Given vertices `u` and `v`: are they adjacent?
  - Given vertex `v` and edge `e`, are they incident?
  - Given an edge `e`, get its incident vertices (*ends*)
  - How many vertices are adjacent to `v`? (*`deg(v)`*)
    - The vertices adjacent to `v` are called its *neighbors*
  - Get a list of the neighbors of `v` (or the edges incident with `v`)

Graph Interface Methods

- void add(V vLabel), V remove(V vLabel)
  - Add/remove vertex to graph
- void addEdge(V vLabel1, V vLabel2, E edgeLabel), E removeEdge(V vLabel1, V vLabel2)
  - Add/remove edge between vLabel1 and vLabel2
- boolean containsEdge(V vLabel1, V vLabel2)
  - Returns true iff there is an edge between vLabel1 and vLabel2
- Edge<V,E> getEdge(V vLabel1, V vLabel2)
  - Returns edge between vLabel1 and vLabel2
- void clear()
  - Remove all nodes (and edges) from graph

Graph Interface Methods

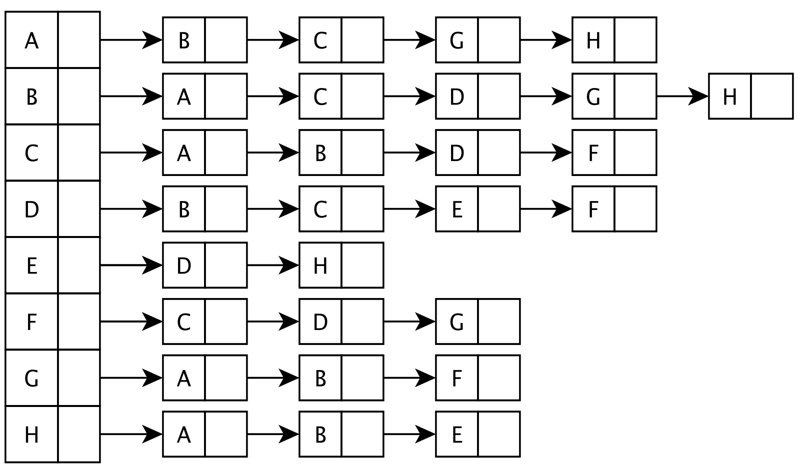
- boolean visit(V vLabel)
  - Mark vertex as "visited" and return *previous* value of visited flag
- boolean visitEdge(Edge<V,E> e)
  - Mark edge as "visited"
- boolean isVisited(V vLabel), boolean isVisitedEdge(Edge<V,E> e)
  - Returns true iff vertex/edge has been visited
- Iterator<V> neighbors(V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

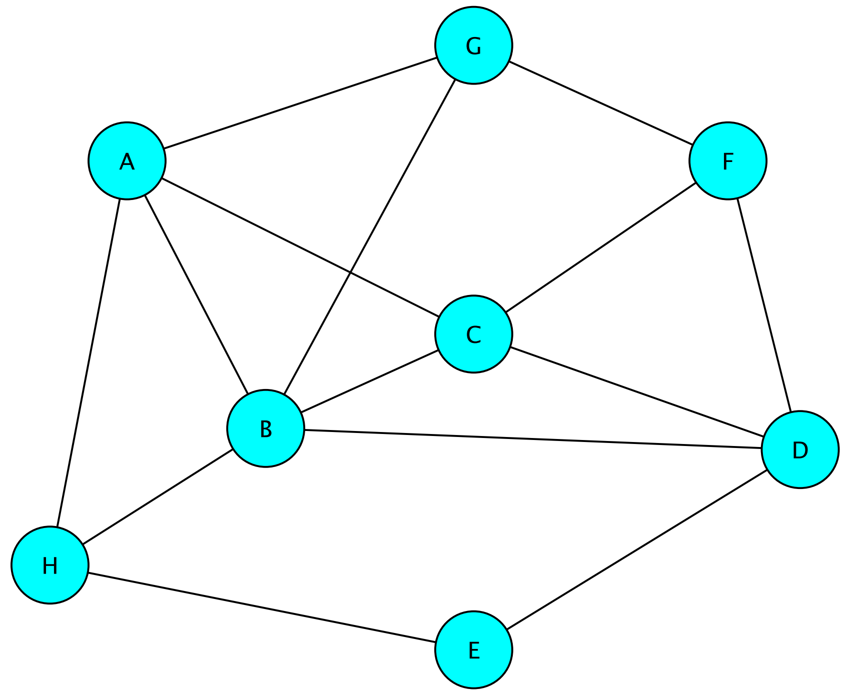
Representing Graphs

- Two standard approaches
  - Option 1: Array-based (directed and undirected)
  - Option 2: List-based (directed and undirected)
- We'll look at both
  - Array-based graphs store the edge information in a 2-dimensional array indexed by the vertices
  - List-based graphs store the edge information in a (1-dimensional) array of lists
    - The array is indexed by the vertices
    - Each array element is a list of edges incident with that vertex

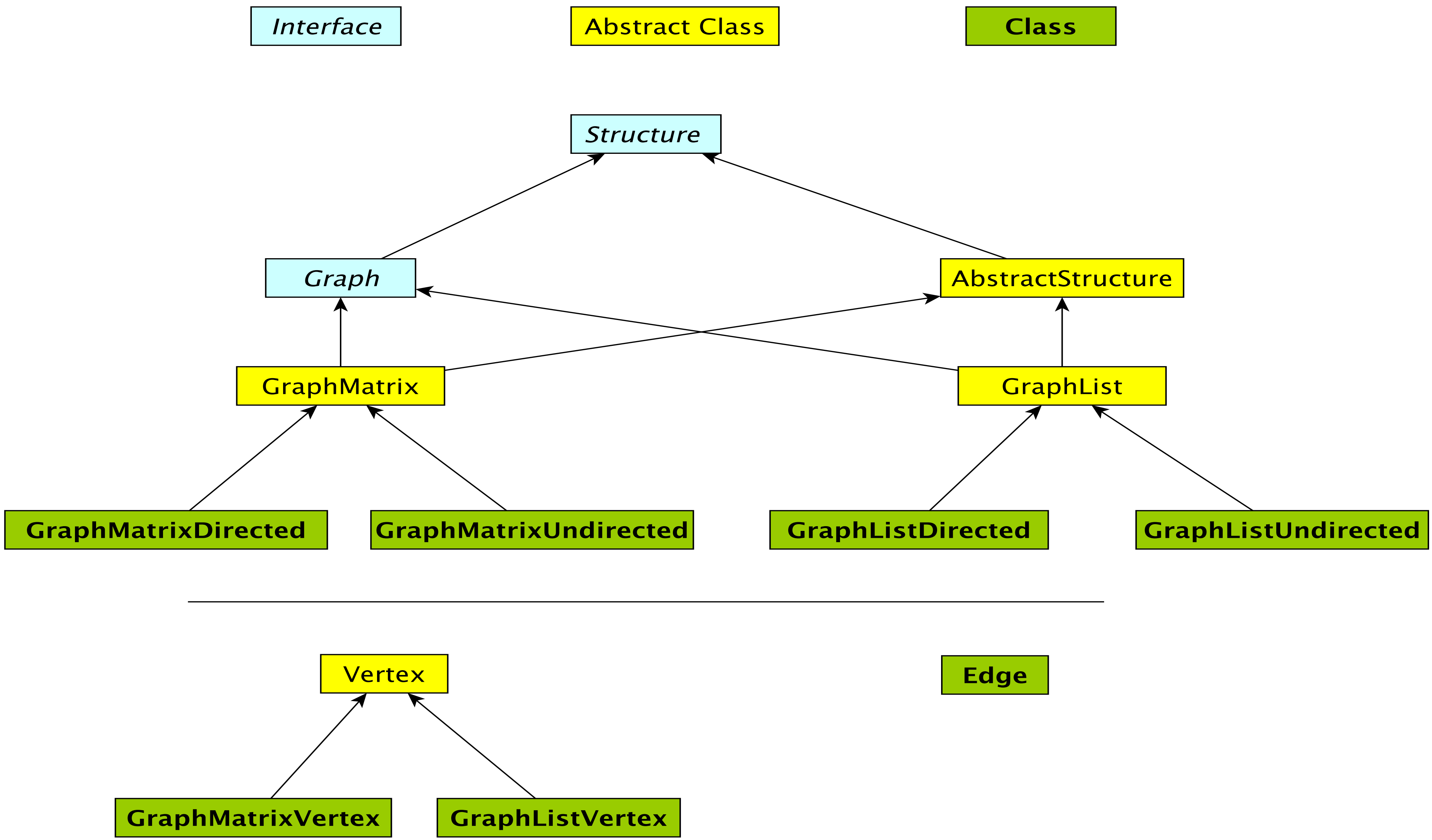
Example Graph Representations: Lists and Matrices

	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





Graph Classes in structure5



Edge Class

- Graph *edges* are defined in their own public class (*vertices* are hidden: referenced only by their label)
  - `Edge<V,E>(V vLabel1, V vLabel2,`
    `E label, boolean directed)`
  - Construct a (possibly directed) edge between two labeled vertices (`vLabel1` → `vLabel2`)
  - vLabel1 : here; vLabel2 : there
- Useful `Edge` methods (getters and setters):

```
label(), here(), there()
setLabel(), isVisited(), isDirected()
```

Reachability: Breadth-First Search

```
BFS(G, v)        // Do a breadth-first search of G starting at v
// pre: all vertices are marked as unvisited
// post: return number of visited vertices
count ←0;
Create empty queue Q;
add v to Q, mark v as visited, add 'v' to count
While Q isn't empty
        current ←Q.dequeue();
        for each unvisited neighbor u  of current :
                add u to Q, mark u as visited, add 'u' to count
return count;
```

How does this translate to code?

Breadth-First Search

```
int BFS(Graph<V,E> g, V src) {
  int count = 0; Queue<V> todo = new QueueList<V>();
  todo.enqueue(src);
  g.visit(src); count++;
  while (!todo.isEmpty()) {
    V vertex = todo.dequeue();
    Iterator<V> neighbors = g.neighbors(vertex);
    while (neighbors.hasNext()) {
       V next = neighbors.next();
       if (!g.isVisited(next)) {
          todo.enqueue(next);
          g.visit(next); count++;
       }
    }
  }
  return count;
}
```

Breadth-First Search of Edges

```
int BFS(Graph<V,E> g, V src) {
  int count = 0; Queue<V> todo = new QueueList<V>();
  todo.enqueue(src);
  g.visit(src); count++;
  while (!todo.isEmpty()) {
    V vertex = todo.dequeue();
    Iterator<V> neighbors = g.neighbors(vertex);
    while (neighbors.hasNext()) {
       V next = neighbors.next();
       if (!g.isVisitedEdge(vertex, next))
              g.visitEdge(vertex, next);
       if (!g.isVisited(next)) {
          todo.enqueue(next);
          g.visit(next); count++;
       }
    }
  }
  return count;
}
```

Recursive Depth-First Search

```
// Before first call to DFS, set all vertices to unvisited
//Then call DFS(G,v)
DFS(G, v)
        Mark v as visited; count=1;
        for each unvisited neighbor u of v:
                count += DFS(G,u);
        return count;
```

How does this translate to code?

Recursive Depth-First Search

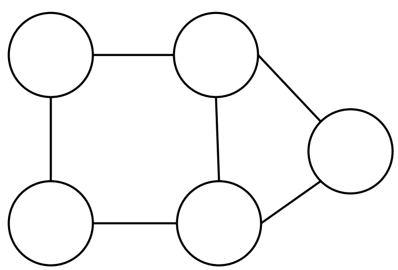
```
int depthFirstSearch(Graph<V,E> g, V src) {
    g.visit(src);
    int count = 1;
    Iterator<V> neighbors = g.neighbors(src);
    while (neighbors.hasNext()) {
       V next = neighbors.next();
       if (!g.isVisited(next))
             count += depthFirstSearch(g, next);
    }
  return count;
}
```

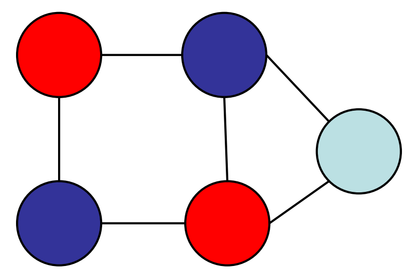
Lab 10 Overview:
Graph Algorithms using structure5

Greedy Algorithms

- A *greedy algorithm* attempts to find a globally optimum solution to a problem by making locally optimum (greedy) choices
- Example: Walking in Manhattan
- 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

Graph Coloring Example





Greedy Coloring : Math

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 $u \neq v$ in *G*

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

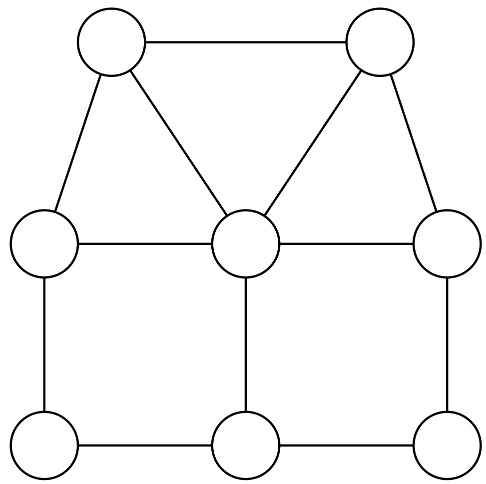
      add *u* to *L*

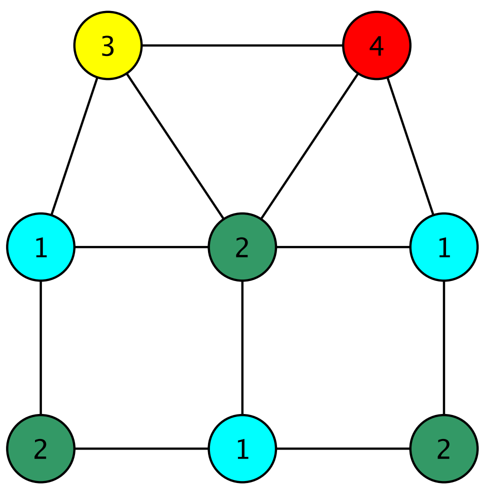
  remove all vertices of *L* from *G*

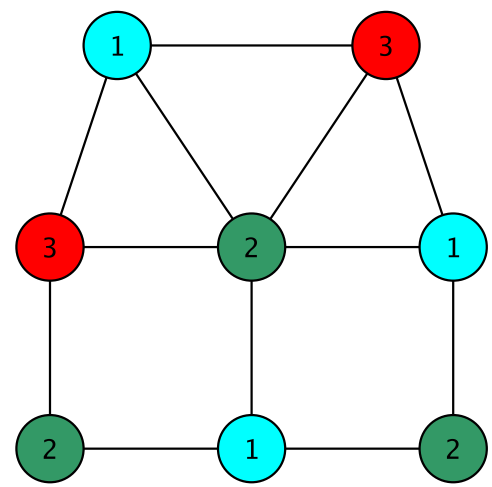
  add *L* to *C*

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 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