Graph Implementations

Instructors: Sam McCauley and Dan Barowy

May 4, 2022

Admin

- Any questions?

Practice Quiz 11 Review

The general contract of `hashCode` is:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must consistently return the same integer, provided no information used in `equals` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
- It is *not* required that if two objects are unequal according to the `equals(java.lang.Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

(Reminder of Java hashcode rules)

Breadth-First Search

BFS

- Breadth-First Search (BFS) is a lot like level order traversal!
- Start with a node. Explore its children in order. Then, explore their children (in the same order)
- Plan: use a queue to store nodes that are waiting
- Use *pseudocode*: a description of an algorithm in code-like notation (without worrying about language-specific details)

Breadth-First Search

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

Analyzing BFS

- Can we prove: BFS(G,v) visits exactly the vertices *u* that are reachable from *v*?
- Why is this true intuitively?
  - If we explore the neighbors of *v*, and their neighbors, etc., we surely need to get to *u* eventually
  - More concretely: if *u* is reachable from *v*, then there must be a path from *v* to *u*. The first node on the path is *v*, which is reached. The second node is a neighbor of *v*, which is reached. The third node…
  - How can we formulate this as a more-formal induction?
- The *distance* between a vertex *v* and a vertex *u* is the minimum length of all paths between *v* and *u* (denoted $d(v, u)$)
- Induction on the distance between *v* and the target vertex

BFS Induction

- Base case ($d = 0$): if $d(v, u) = 0$, then $u = v$. BFS visits v, so BFS also visits u.
- Inductive Hypothesis: For some $d \geq 0$, for all vertices u with $d(v, u) = d$, BFS(G,v) visits u.
- Inductive step: consider some u where $d(v, u) = d + 1$. Then there is a path of length $d + 1$ from v to u:

  $$v = v_0, e_1, v_1, \ldots v_d, e_{d+1}, v_{d+1} = u$$

  Then there is a path of length d from v to v_d. By the *Inductive Hypothesis*, v_d is visited by BFS(G,v) and added to Q. Then v_d is removed from Q, and u is visited.

Induction on Graphs and Trees

- We'll see a few more examples
- Same basic parts!
- Often: remove one node to obtain a smaller instance. Then, use the inductive hypothesis.

Interesting BFS Aside

- We visit vertices in order of distance
- So: with some extra bookkeeping, can use BFS to calculate the shortest path in a graph!
- We'll come back to this next week

Implementing Graphs

First: `Graph` Interface

- Supports storing a value at each vertex and edge
  - Called a *label*
  - Can be any kind of object. (That is to say: we'll use generic types for both.)
- Support methods for:
  - Get the value of a vertex or edge
  - Add/remove vertices and edges
  - Search for vertex/edge labels
  - Query/change "visited" state of vertices and edges
  - Iterators of vertices, neighbors, edges

Graph Interface Methods

- `void add(V vtx), V remove(V vtx)`
  - Add/remove vertex to the graph
- `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 if there is an edge between `vtx1` and `vtx2`
- `Edge<V,E> getEdge(V vtx1, V vtx2)`
  - Returns the edge between `vtx1` and `vtx2`
- `void clear()`
  - Remove all nodes and edges from the graph

Graph Interface Methods (2)

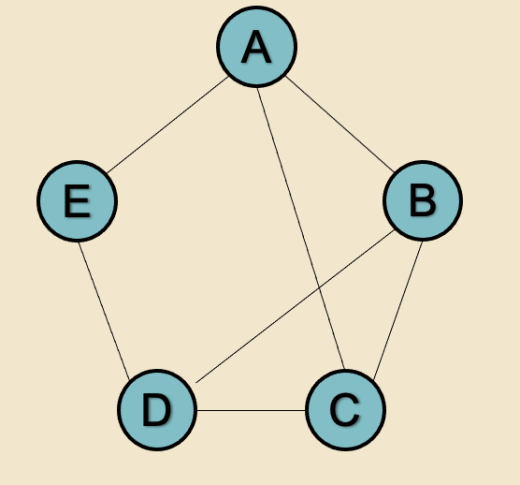
- `boolean visit(V vertexLabel)`
  - Mark vertex as “visited”; returns previous value of visited flag
- `boolean visitEdge(Edge<V,E> e)`
  - Mark edge as “visited”
- `boolean isVisited(V vtx), boolean isVisitedEdge(Edge<V,E> e)`
  - Returns if given vertex/edge has been visited
- `Iterator<V> neighbors(V vtx)`
  - Get iterator for all neighbors of `vtx`. (Out-edges only for directed graphs.)
- `Iterator<V> iterator()`
  - Get vertex iterator
- `void reset()`
  - Reset all visited flags

Pseudocode to Code

```
// pre: all vertices are
    unvisited
BFS(G, v):
   count ← 0
   Create empty queue Q
   Q.enqueue(v)
   mark v as visited
   count++
   while Q isn't empty:
      current ← Q.dequeue()
      for each unvisited
          neighbor u of
          current:
         Q.enqueue(u)
         mark u as visited
         count++
   return count;
```

```
public static <V,E> int BFS(Graph<V,E> g, V
    src) {
  Queue<V> todo = new QueueList<V>();
  todo.enqueue(src);
  g.visit(src);
  int count = 1;
  while (!todo.isEmpty()) {
    V node = todo.dequeue();
    Iterator<V> neighbors =
        g.neighbors(node);
    while (neighbors.hasNext()) {
      V next = neighbors.next();
      if (!g.isVisited(next)) {
        g.visit(next); count++;
        todo.enqueue(next);
      }
    }
```

Creating a Graph



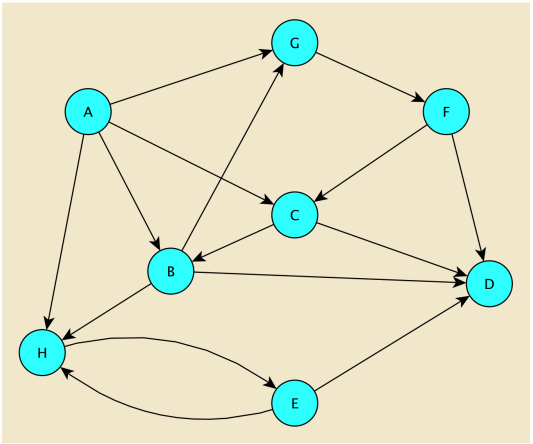
- Let’s look at some code to create this graph
- Then, we’ll run BFS on it

How can we store the edges of a graph?

- Need to be able to quickly determine the neighbors of a vertex, what edges are adjacent to it, etc.
- Two options:
  - Adjacency Matrix (Adjacency Array)
  - Adjacency List

Adjacency Matrix Representation

Adjacency Matrix



	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

If there's an `Edge` between i and j, Entry(i,j) stores it. Else, Entry(i,j) stores `null`.

(We use `1` in the picture, but in reality it will be a reference to some `Edge` object)

Adjacency Matrix

- How can we store a matrix?
- One option: *two-dimensional array*
- Works just like an array, but with two indices to access each element
- So: can access e.g. `data[i][j]`
- Has fixed size (like an array)

How to use the adjacency matrix

- How can we find the neighbors of a vertex *v*?
- Go to corresponding row of matrix
- Scan through the row. Each time we see a non-null `Edge` *e*, look at the two vertices of *e*. The non-*v* vertex is a neighbor!
- Let's look at the code for `Edge`, and the node for `neighbors()` in `GraphMatrix`

Making the adjacency matrix work

- How can I look up a vertex in the matrix?
- We look up by label, but we need a specific *row* in the matrix
- Each `GraphMatrixVertex` object stores its own index for its row (in addition to label, visited, etc.)
- How can we get the `GraphMatrixVertex` object that corresponds to a given label (of type `V`)?
- Answer: a hash table!

Graph Matrix Classes

- `GraphMatrixVertex` and `Vertex`: classes for holding vertices
- `Edge`: class for holding edges
- `GraphMatrix`: abstract class for graphs stored using adjacency matrix
- `GraphMatrixDirected` and `GraphMatrixUndirected`: any remaining methods (that differ between directed and undirected graphs)
- Let's take a look!

Analyzing Adjacency Matrix Representation

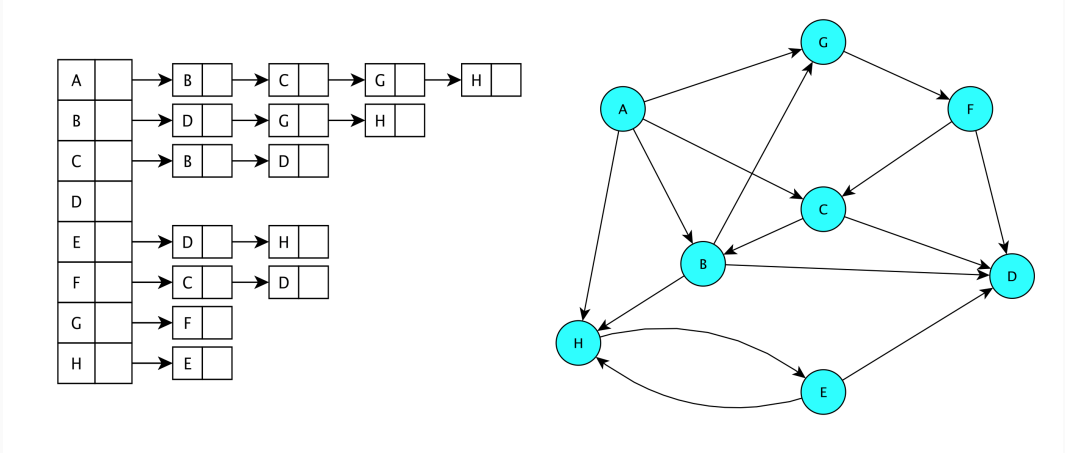
- Let's say we have a graph with n vertices and m edges
- How long does it take to find all neighbors of a vertex?
  - $O(n)$ (need to scan through all columns—coresponding to all vertices)
- How long does it take to find the edge between vertices v_1 and v_2? To add a new edge between two vertices?
  - $O(1)$! Just need to look it up in the matrix
- Space?
  - $O(n^2)$ (Can be very large!)

Adjacency List Representation

Adjacency Lists

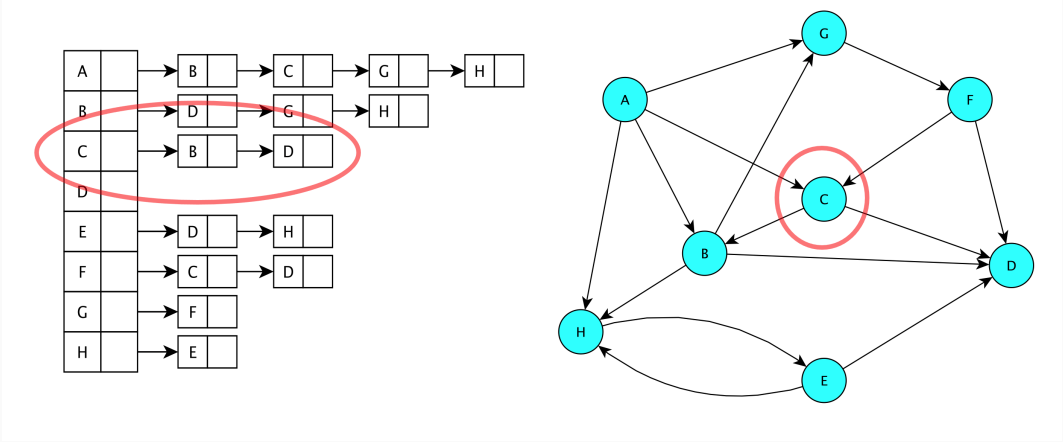
- The adjacency matrix was very wasteful of space, and finding the neighbors of a vertex was very slow
  - But, finding if there was an edge between two vertices was very fast
- Adjacency list representation: maintain a list all edges that are indicent to each vertex
  - Only keep *outgoing* edges for directed graphs
  - Usually going to be a singly linked list
- Abstract class `GraphList`, concrete classes `GraphListDirected` and `GraphListUndirected`; also a new vertex class `GraphListVertex`

Adjacency List Visualization



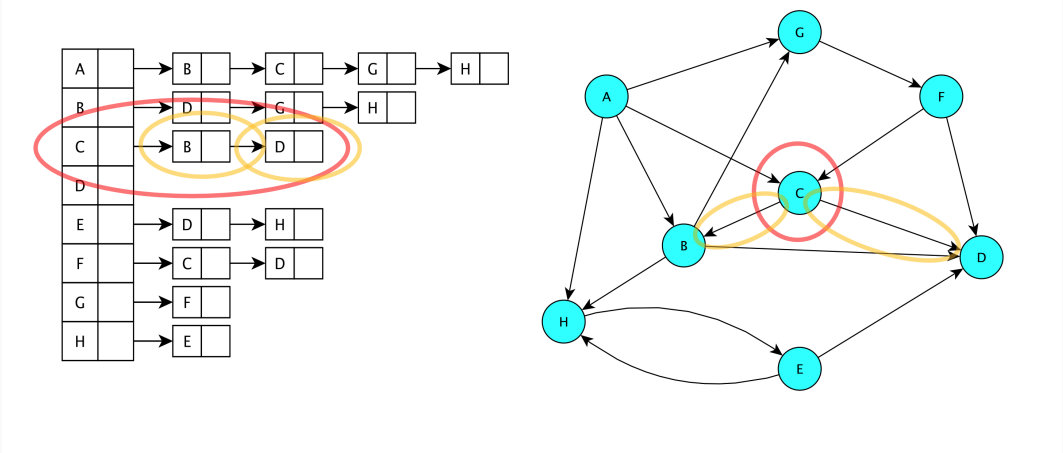
The vertices are stored in a `Map<V, GraphListVertex<V,E>>`. Each `GraphListVertex<V,E>` contains a linked list of all edges with a given *source*

Adjacency List Visualization



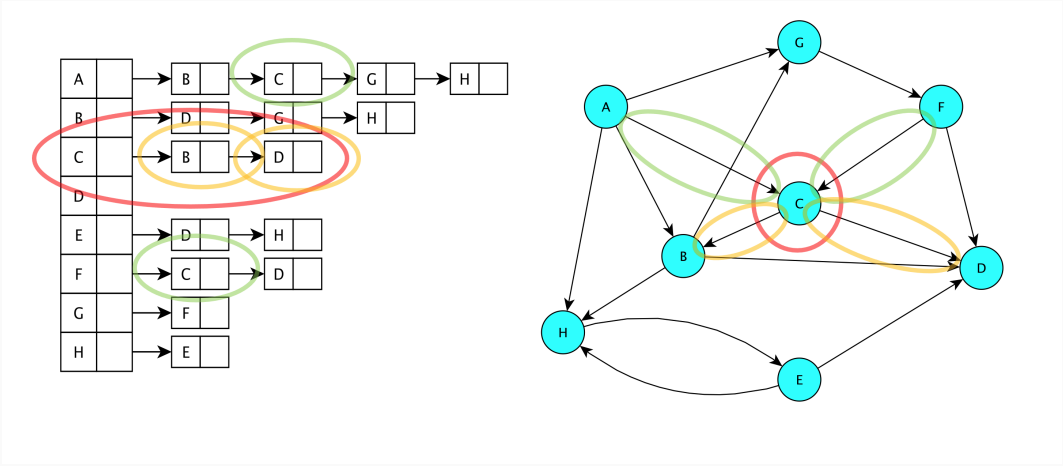
The vertices are stored in a `Map<V, GraphListVertex<V,E>>`. Each `GraphListVertex<V,E>` contains a linked list of all edges with a given *source*

Adjacency List Visualization



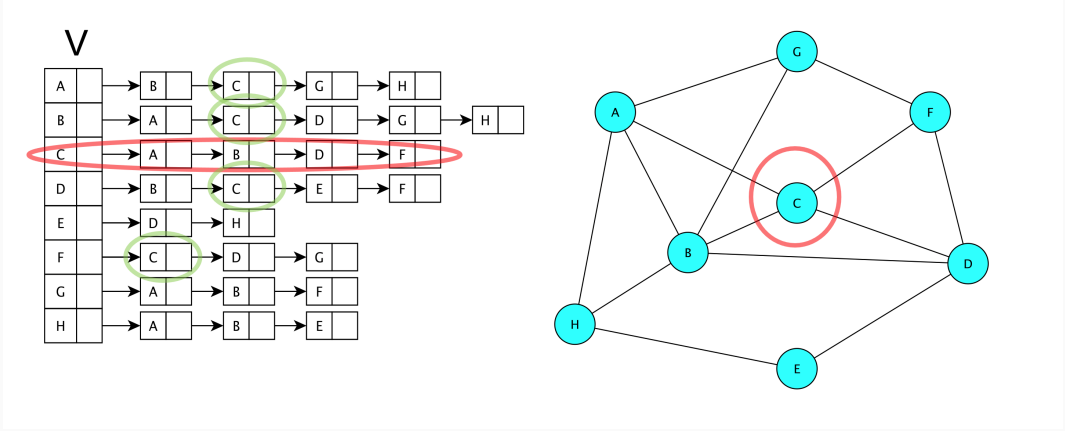
The vertices are stored in a `Map<V, GraphListVertex<V,E>>`. Each `GraphListVertex<V,E>` contains a linked list of all edges with a given *source*

Adjacency List Visualization



The vertices are stored in a `Map<V, GraphListVertex<V,E>>`. Each `GraphListVertex<V,E>` contains a linked list of all edges with a given *source*

Adjacency List Visualization: Undirected



The vertices are stored in a `Map<V, GraphListVertex<V,E>>`. Each `GraphListVertex<V,E>` contains a linked list of all edges *indident* to that vertex.

Creating adjacency list classes

- What does `GraphListVertex` need on top of `Vertex`?
- What is the difference between `GraphList` and `GraphMatrix`?
  - Do not need a free list of remaining vertices
  - Do not need to know number of vertices ahead of time
- Let's take a look

Operations on an adjacency list for a graph?

- Let's say we have a graph with n vertices and m edges
- Getting all neighbors of a vertex?
  - $O(\#\text{ neighbors})$
- Adding or removing a vertex?
  - $O(1)$
- Getting an edge?
  - $O(\#\text{ neighbors of vertex})$. Could be as bad as $O(n)$!
- Space?
  - $O(1)$ space per vertex or edge. Total: $O(n + m)$

Adjacency List vs Adjacency Matrix

- Adjacency List is (often) much faster for listing neighbors of a vertex:
    - Adjacency Matrix gives time proportional to the total number of vertices, Adjacency List gives time proportional to the number of neighbors
- Adjacency Matrix is much faster for looking up if there is an edge bettween two vertices
- Adjacency List is much more space efficient if $m < n^2$