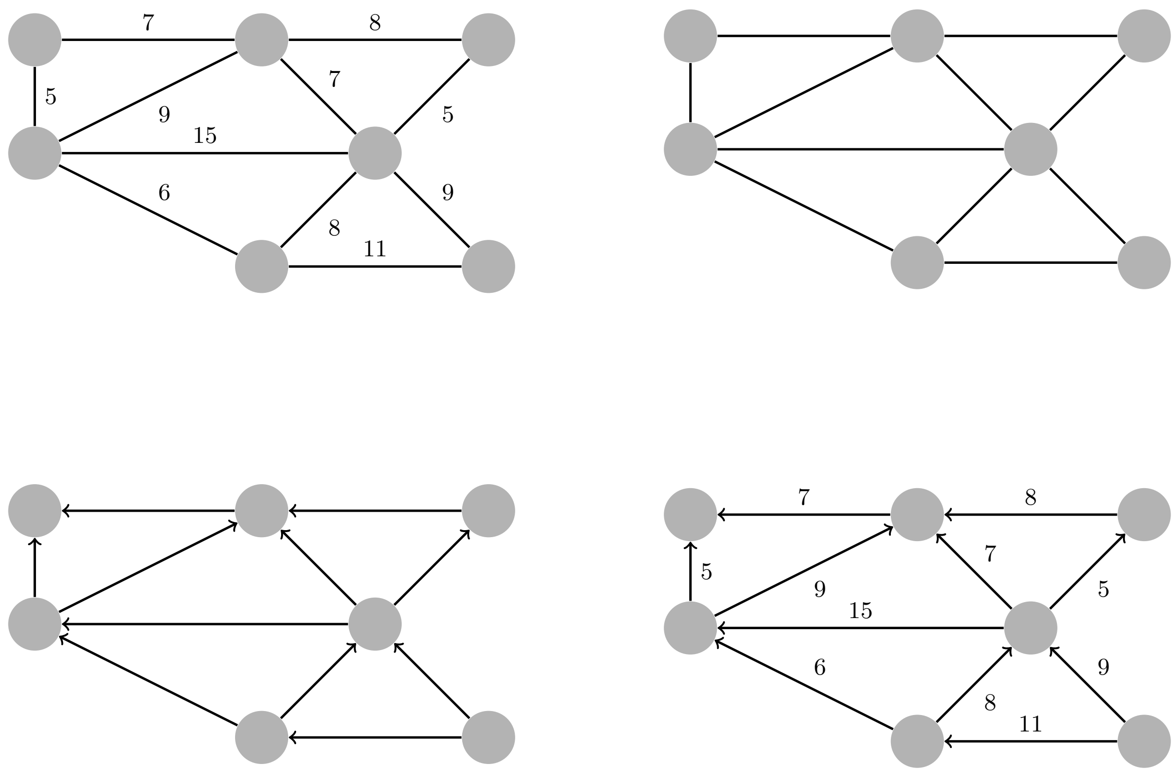
University of Illinois @ Urbana-Champaign

CI 487: Data Structures for CS Teachers

Implementation #6: Implementing a Generic Graph



1 Objectives and Overview

The objective of this lab will be the following:

1. Construct a generic graph with an adjacency list.
2. The graph constructor and implementation should allow for weighted/unweighted and directed/undirect construction and operations.
3. The graph should support Breadth-First Search (BFS) and Depth-First Search(DFS) from a given starting vertex.

To construct an generic graph that supports operations for a generic graph represented using an *adjacency list*.

2 Checkpoint 1

2.1 Adjacency List Design Details

Lets begin this lab with a general description of the algorithms and general design of the graph we will be building.

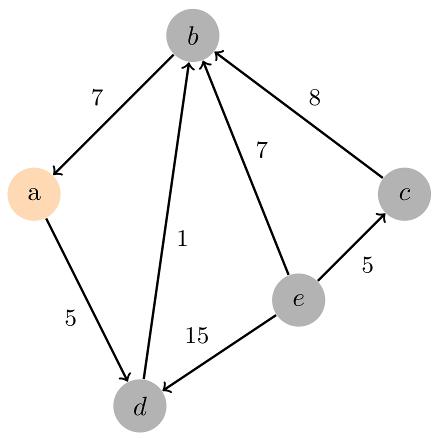
2.1.1 Adjacency List

value=12
weight=1

Figure 1: A visual representation of an edge

Our implementation of an adjacency list will be composed of a HashMap with vertices as keys and edge lists as values. As you will see in Section 2.2, an edge is composed of two attributes, a weight and the vertex containing a destination (Figure 1). The following is an example of how a graph is represented by an adjacency matrix.

Key	Value
a	dest=D, weight=5 → ⊠
b	dest=A, weight=7 → ⊠
c	dest=B, weight=8 → ⊠
d	dest=B, weight=1 → ⊠
e	dest=D, weight=15 → dest=B, weight=7 → dest=C, weight=5 → ⊠



2.2 Edge

Just as with the simpler graphs we covered this semester (e.g., Trees, LinkedLists) our graph must have some set of basic units. For graphs these units are the vertecies and the edges. Since we are implementing our graph with an adjacency list we will need a method of indicating to which vertices a vertex is connected and what the weight of that edge is, if applicable. As such, we will be building an inner class `Edge<E>` that stores this information on the destination and the weight. The adjacency list that results from this will be a lists of edges that exist with respect to a given vertex. As you will see in the starter files, this class is inner class within the encompassing `ListGraph<E>` class since it has little utility outside the context of constructing a list graph. In this case `E` will be the type of our vertecies since we are represnting the graph with just an edgelist.

2.2.1 Attributes

The following attributes will be used for representing the edges of our class:

1. `final int weight`
2. `final E dest`

Both of these attributes should be declared as `final` since, once set, we will not be changing the values associated with an instance of an `Edge`.

2.2.2 Constructors

`public Edge(E dest)...` : This constructor will be used for the creation of edges for unweighted graphs; hence, the reason for the absence of an integer parameter indicating the weight of the edge. It should therefore set the `weight` attribute to zero when a new edge is instantiated using this constructor. The `dest` parameter is generic and will contain a reference to the destination object.

`public Edge(E dest, int weight)...` This constructor is similar to the former one but adds a primitive integer parameter `weight`. As such, this constructor will be used for the construction of weighted edges and should set the `dest` and `weight` attributes of the object equal to the parameters passed through the function.

2.3 ListGraph

2.3.1 Attributes

1. `private final boolean directed`
2. `private final boolean weighted`
3. `private Map<E, List<Edge<E>>> map`

The purpose of the `directed` and `weighted` is to direct the control flow of methods relating to the addition and removal of edges. The map is the data-structure we will be using to represent our adacency list. Each vertex `E` will be associated with a `List` of `Edge` instances.

2.3.2 Constructor

`public ListGraph(boolean directed, boolean weighted){ ... }` : This is the only constructor we will have for this class since control flow of variations on the graph will be handled via the two parameters. It should set the `directed` and `weighted` attributes equal to the values passed in as parameter s to the constructor. It should also create a new, empty instance of `Map` that associates vertecies of `E` with and edge list of `List<Edge<E>>`.

2.3.3 Vertex Methods

`public void addVertex(E vertex)...` : The add vertex method should create a new entry in the `map` and associate it with a new instance of an empty `List` of `Edges`.

`public void removeVertex(E vertex)...` : The remove vertex function should do two things: (1) remove the given vertex and it's associated list of edges and (2) search through all other edge lists in `map` and all those edges with the given vertex as the `dest`.

`public Set<E> getVertecies()...` : This should return a list of the vertecies currently in `map`.

Hint: This method should be one line of code that returns the keyset from the map. Refer to the `HashMap` documentation for which method can accomplish this.

2.3.4 Edge Methods

The following two methods will handle the addition of edges to the graph.

1. `public void addEdgeUnweighted(E source, E dest){ ... }`
2. `public void addEdgeWeighted(E source, E dest, int weight){ ... }`

Each of these methods should first check if both the given `source` and the `dest` should be in the `map`. If they are, it should instantiate a new edge, *taking care to call the appropriate constructor*, and add the edge to `source`'s associate edge list in `map`. If the graph is undirected, it should follow a similar process to create an edge from `dest` to `source`.

`public void removeEdge(E source, E dest){ ... }` : This method should remove the edge from `source` to `dest` and, if the graph graph is undirected, from `dest` to `source`.

`public List<Edge<E>> getEdges(E vertex){ ... }` : This is a getter method that retrieves the edgelist associated with a given vertex.

Hint: This method should be one line of code that returns the list associated with a vertex. Refer to the `HashMap` documentation for which method can accomplish this.

3 Checkpoint 2

For the second checkpoint you will be implementing both a breadth first and depth first search. In donig so, you will complete the following two methods:

`public List<E> bfs(T source){ ... }` : This function should perform the BFS function described in Section 3.0.1 begining at the `source` node passed in as a parameter, collect a list of references to each of the objects along the way, and return that list once the algorithm has completed.

`public List<E> dfs(T vertex){ ... }` : This method should perform the DFS function described in Section 3.0.2 begining at the `source` node passed in as a parameter, collect a list of references to each of the objects along the way, and return that list once the algorithm has completed.

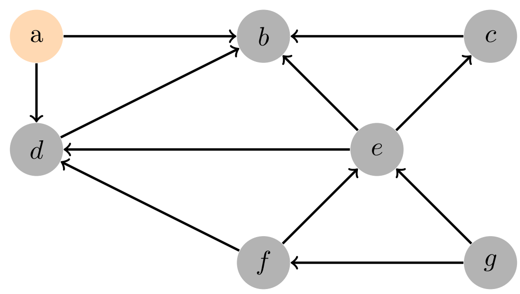
3.0.1 Breadth-First Search

Algorithm 1: Breadth First Search

```
Function BFS(G, Source)
    for u ∈ G.Vertex do
        u.visited ← null
    end
    Q ← ∅
    Q.enqueue(Source)
    while Q ≠ ∅ do
        u ← Q.dequeue()
        if u is visited then
            continue
        end
        u.visited ← true
        for edge ∈ G.Adj[u] do
            if edge.dest is not visited then
                Q.enqueue(edge.dest)
            end
        end
    end
```

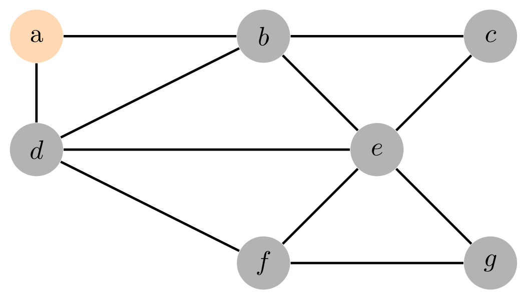
Note: the traversals shown on the right will vary depending on the order in which vertecies are pushed to the queue.

Results for Directed Graph:



Traversal: a b d

Results for Undirected Graph:



Traversal: a b d e c f g

Algorithm 1 presents the iterative approach you will take for implementing BFS from a given source node. The purpose of the variables used in that psudeocode are as follows:

- `G` is the graph on which the alorithm will operate.
- `Source` is our starting vertex
- `Q` is a *queue* that contains the ondes under consideration for traversal. We initiallize it to empty, add the starting vertex, and then begin the traversal.

As with the other psudeocode we've used in this class, it is important to note that the psudeocode is simply a general set of instructions on how the solution should be structured. Your implementation should follow it in spirit but *will not be a direct translation of the psudeocode to Java.* Noteably, your vertecies will not have a `visited` attribute. You will instead need to use some other inbuilt data structure (e.g., `Map`) to keep track of whether or not a node has been visited.

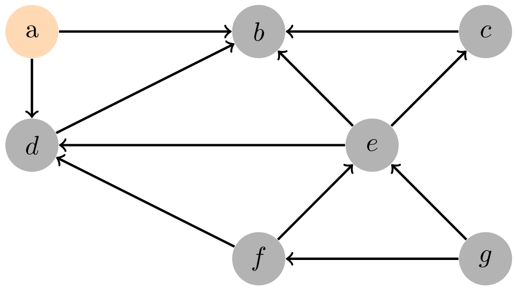
3.0.2 Depth-First Search

Algorithm 2: Depth First Search

```
Function DFS(G, Source)
    for u ∈ G.Vertex do
        u.visited ← false
    end
    S ← ∅
    S.push(Source)
    while S ≠ ∅ do
        u ← S.pop()
        if u is visited then
            continue
        end
        u.visited ← true
        for edge ∈ G.Adj[u] do
            if edge.dest is not visited then
                S.add(edge.dest)
            end
        end
    end
return
```

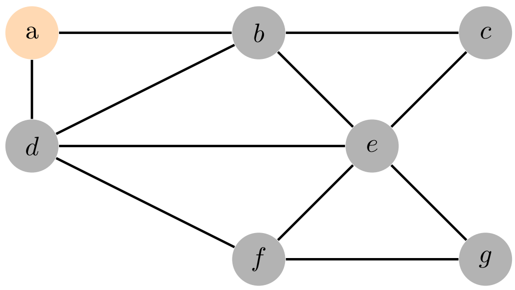
Note: the traversals shown on the right will vary depending on the order in which vertecies are pushed to the stack.

Results for Directed Graph:



Traversal: a b d

Results for Undirected Graph:



Traversal: a d f g e b c

Algorithm 2 presents the iterative approach you will take for implementing DFS from a given source node. This is very similar to BFS however there are a few new variables in the psudeocode

- `G` is the graph on which the alorithm will operate.
- `Source` is our starting vertex
- `S` is a *stack* that contains the node still under consideration for traversal.

Again, your implementation should follow it in spirit but *will not be a direct translation of the psudeocode to Java*. As with the BFS, your vertecies will not have a `visited` attribute. You will instead need to use some other inbuilt data structure (e.g., `Map`) to keep track of whether or note a node has been visited.

4 Checklist

- Edge
  - ☐ The class has a `weight` and `attribute`
  - ☐ The class has two constructors, one for weighted edges and the other for unweighted.
- ListGraph
  - ☐ The following methods have been implemented:
    - ☐ `addVertex`
    - ☐ `removeVertex`
    - ☐ `getVertecies`
    - ☐ `addEdgeUnweighted`
    - ☐ `addEdgeWeighted`
    - ☐ `removeEdge`
    - ☐ `getEdges`
    - ☐ `bfs`
    - ☐ `dfs`