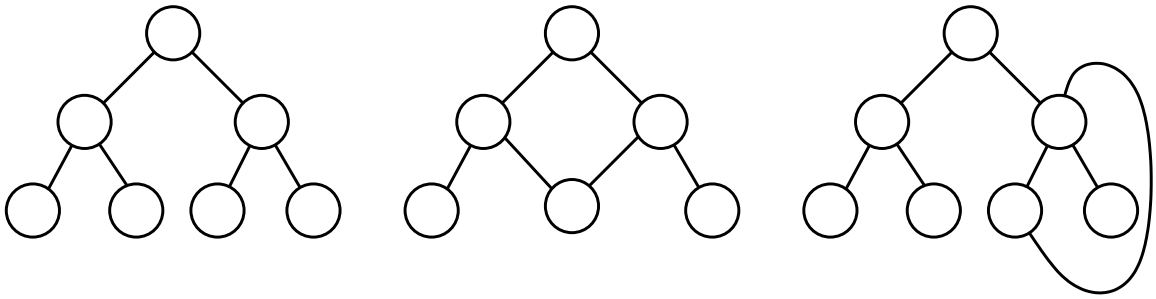
Computer Science 501
Data Structures & Algorithms
The College of Saint Rose
Fall 2014

Topic Notes: Graphs

When does a tree stop being a tree? When it has a cycle!



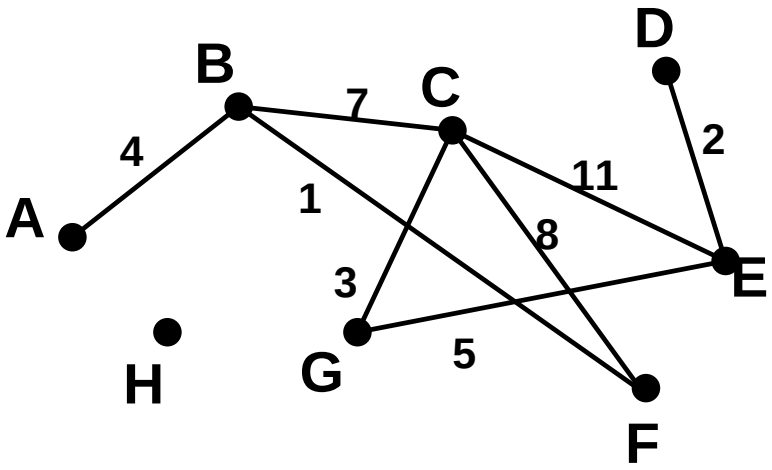
Just like a list is really just a boring case of a tree (everyone has just one child), a tree is really just a boring case of a graph (no cycles).

Definition and Terminology

A *graph* G is a collection of *nodes* or *vertices*, in a set V, joined by *edges* in a set E. Vertices have labels. Edges can also have labels (which often represent *weights*). Such a graph would be called a *weighted graph*.

The graph structure represents relationships (the edges) among the objects stored (the vertices).

For a tree, we might think of the tree nodes as vertices and edges labeled “parent” and “child” to represent nodes that have those relationships.



- Two vertices are *adjacent* if there exists an edge between them.

  e.g., A is adjacent to B, G is adjacent to E, but A is not adjacent to C.

- A *path* is a sequence of adjacent vertices.

  e.g., A-B-C-F-B is a path.

- A *simple path* has no vertices repeated (except that the first and last may be the same).

  e.g., A-B-C-E is a simple path.

- A simple path is a *cycle* if the first and last vertex in the path are same.

  e.g., B-C-F-B is a cycle.

- *Directed graphs* (or *digraphs*) differ from *undirected graphs* in that each edge is given a direction.

- The *degree* of a vertex is the number of edges incident on that vertex.

  e.g., the degree of C is 3, the degree of D is 1, the degree of H is 0.

  For a directed graph, we have more specific *out-degree* and *in-degree*.

- Two vertices u and v are *connected* if a simple path exists between them.

- A *subgraph* S is a *connected component* iff there exists a path between every pair of vertices in S.

  e.g., $\{$A,B,C,D,E,F,G$\}$ and $\{$H$\}$ are the connected components of our example.

- A graph is *acyclic* if it contains no cycles.

- A graph is *complete* if every pair of vertices is connected by an edge.

A Sample Graph Problem

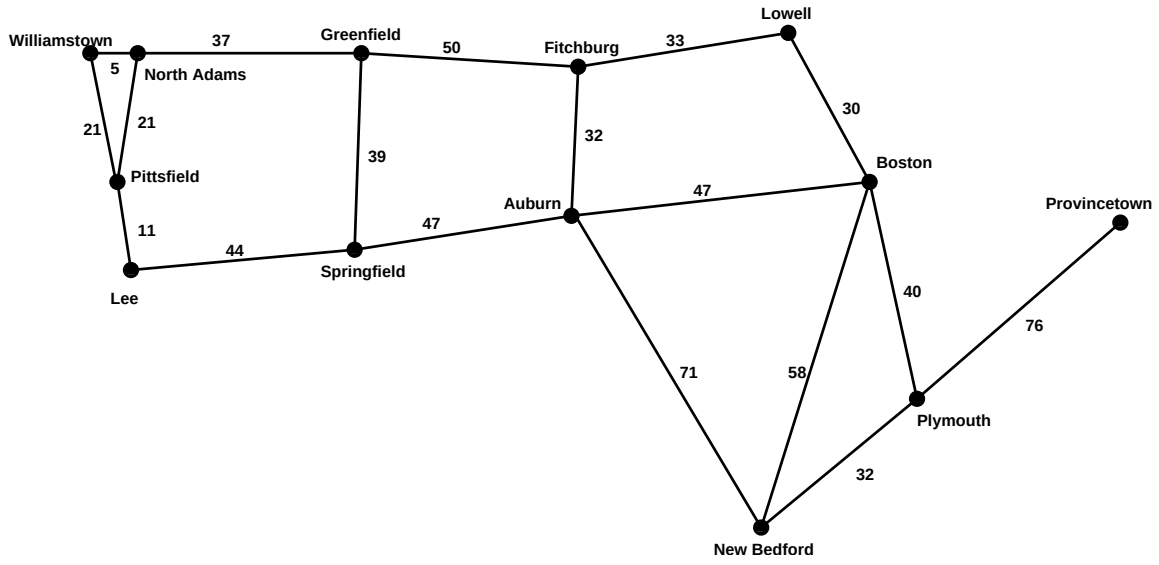
Many problems in computer science can be converted to graph problems.

Consider, for example, an application in which we need to plan a driving route from Williamstown to Boston.

We might represent Williamstown and all other towns in Massachusetts as vertices; we might represent roads as edges between the vertices.

If we labeled the edges with the mileage between the vertex cities, the path planning problem then becomes a problem of finding the shortest (weighted) path in the graph between Williamstown and Boston.

Here's a subset of that data that we'll use later:



The Graph Interface

As with many of our structures this semester, we will have an interface that defines a general behavior of graphs, independent of what structures we actually use to represent them in specific implementations.

See Structure Source:
`/home/cs501/src/structure5/Graph.java`

`Graph` has two type parameters – `V` determines the types of the labels of the graph vertices, `E` determines the types of the labels of the graph edges.

We have the usual methods like `add`, `remove`, `get`, `contains`, but what should these mean? We have both vertices and edges!

Here, we use these to manipulate the vertices in the graph.

Note that vertices are specified by their labels. Often we will use strings, but the labels may be of any type.

There is a corresponding set of methods that deals with edges, but these are named `addEdge`, `removeEdge`, etc.

Note that edges are added by specifying the labels of the vertices to which it is connected, and the label of the edge itself.

The `getEdge` method doesn't return an edge label but rather a new structure we haven't looked at yet called an `Edge`. We will see this one shortly.

And there are also a number of methods that deal with vertices and edges being "visited". Many graph algorithms need to know which vertices or edges they've already considered, so this has been designed right into the graph interface.

Finally, there are a number of methods that give us some information about the graph or about particular vertices, such as `degree`, `neighbors`, and iterators over vertices an edges.

Implementations of Graphs

First, we have classes to represent vertices and edges. These are quite simple:

See Structure Source:
`/home/cs501/src/structure5/Vertex.java`

First, we notice that `Vertex` is not a public class. Code outside of the structure cannot create a `Vertex`.

A `Vertex` is uniquely defined by its label (an object of type `E`).

Important note: the label used for our `Vertex` can be of any type, but is assumed to be *immutable*. If it is an instance of a class that can be modified (e.g., a `Vector`), we cannot modify it after using it as a `Vertex` label.

We also keep the `visited` flag for use later in traversals and other algorithms.

See Structure Source:
`/home/cs501/src/structure5/Edge.java`

Unlike the `Vertex`, `Edge` is a public class. Some `Graph` methods return an `Edge`, so it must be public. An `Edge` is defined by its two `Vertex`s, and also may have a label of its own. It also has the `visited` flag.

`Vertex` and `Edge` classes may need to be extended as we implement specific types of `Graph`s.

A `Graph` is really just a mechanism to manage all of these edges and vertices.

If there are a fixed number of edges from each node then we can have fixed number of edges stored with each node (like a binary tree).

For general graphs, we typically use either

1. an *adjacency matrix*, or
2. *adjacency lists*.

As a running example, we will consider an undirected graph where the vertices represent the states in the northeastern U.S.: NY, VT, NH, ME, MA, CT, and RI. An edge exist between two states if they share a common border, and we assign edge weights to represent the length of their border.

We will represent this graph as both an adjacency matrix and an adjacency list.

Adjacency Matrix Representation

In an adjacency matrix, we have a two-dimensional array, indexed by the graph vertices. Entries in this array give information about the existence or non-existence of edges.

We represent a missing edge with `null` and the existence of an edge with a label (often a positive number) representing the edge label (often representing a weight).

Labels of vertices are stored in a dictionary, so we can look up corresponding index for each vertex label.

Adjacency matrix representation of NE graph

	NY	VT	NH	ME	MA	CT	RI
NY	null	150	null	null	54	70	null
VT	150	null	172	null	36	null	null
NH	null	172	null	160	86	null	null
ME	null	null	160	null	null	null	null
MA	54	36	86	null	null	80	58
CT	70	null	null	null	80	null	42
RI	null	null	null	null	58	42	null

If the graph is undirected, then we could store only the lower (or upper) triangular part, since the matrix is symmetric.

Since there is a lot of the implementation that will be common between the directed and undirected matrix-based graphs, the structure package defines an abstract class `GraphMatrix`.

See Structure Source:
`/home/cs501/src/structure5/GraphMatrix.java`

Two implementations, `GraphMatrixDirected` and `GraphMatrixUndirected`, extend it, adding in the functionality that depends on the directed-ness of the graph.

See Structure Source:
`/home/cs501/src/structure5/GraphMatrixDirected.java`

See Structure Source:
`/home/cs501/src/structure5/GraphMatrixUndirected.java`

In the abstract class, we declare all of the instance variables needed to support both matrix-baseds implementations:

- `data`: a two dimensional array of edges. Note that we need to store them as `Object` for the same reasons we saw in the implementation of `Vector`. In actuality, the items stored in this array will be of type `Edge<V,E>`.
- `freeList`: a list of integers which represent available vertex indices. More on this below.
- `dict`: a mapping from vertex labels to (integer) vertex indices that can be used to index into the `data` array.
- `directed`: a boolean flag to indicate the directed-ness of the graph

We won't worry too much about the `Map` that translates vertex labels to indices yet. It's using a hash table – a topic we'll cover after graphs. For now, just realize it should be (and will be) an efficient tool to look up indicies from vertex labels.

The free list indicates which of our vertex indices are available to be assigned to new vertices being added to the graph. For efficiency of the matrix-based implementation, the maximum number of vertices is specified at construction time. This will be an important restriction to be aware of with the matrix-based representation of graphs. It could be made to expand as needed like a `Vector`, but this implementation does not support that. We would simply run out of space for vertices and throw an exception.

The constructor, as we expect, initializes our instance variables to represent an empty graph. Since the constructor doesn't need to care whether the edges are directed or not, the constructor can be defined in the abstract class.

However, it is declared as `protected` since this will not be called by users, they will need to construct directed or undirected constructors.

Those constructors don't do anything else, but they are necessary because we can't construct an instance of an abstract class. Note that they pass the appropriate boolean value to the abstract class constructor to indicate directed-ness.

Note that by constructing a `GraphMatrix` capable of storing up to `size` vertices, we allocate $O(\texttt{size}^2)$ space, even for an empty graph!

Adding a vertex can be done entirely in the abstract class, as this is the same for both directed and undirected graphs. If the vertex is not already in the graph, we look up a free index and associate it in our map with the label of the vertex.

However, we store more than just the index for the label, we have a `GraphMatrix`-specific extension of the `Vertex` class.

See Structure Source:
`/home/cs501/src/structure5/GraphMatrixVertex.java`

In addition to the label and the visited flag provided by `Vertex`, `GraphMatrixVertex` stored the index to allow quick access from a Vertex to its row/column index in the adjacency matrix.

In the vertex add method, we are just making sure that we're not adding a duplicate vertex, getting an available row, and creating a new `GraphMatrixVertex` and remembering it in our mapping between labels and vertices. The row/col number is remembered as part of the vertex.

The cost of this depends on the cost of the methods associated with the label/vertex mapping. Efficient implementations of such mappings will be the subject of the last major topic in the course. At worst, it should involve linear time searches, and we'll see it can be much better.

Adding an edge, however, requires knowledge of the directed-ness, so this is an abstract method in the abstract class, and is provided by the subclasses. The implementations are similar:

- For the undirected graph, we find the indices of its endpoints and create an edge to be stored in two matrix slots (since we need to represent it in both directions).

- For a directed graph, the method is the same, except we only add the edge in the specified direction, leaving the edge corresponding to the other direction alone.

Removing a vertex can be done in the abstract class. We remove it from the lookup table, clear any edges that might be using that index, and add the now-available position to the free list. Note that this means edges are silently removed if either of their vertices is removed.

Removing an edge needs to be done in the subclasses, again so we can remove the edge from just one matrix slot in the directed case, two matrix slots in the undirected case.

Finding a vertex or an edge or checking containment of vertices or edges are also simple and done in the abstract class.

Mutator and accessor methods to set and retrieve the visited attributes of the vertices and edges are also straightforward.

`visit` and `isVisited` apply to vertices, `visitEdge` and `isVisitedEdge` apply to edges, `reset` clears the visited flags for all vertices and edges.

We can easily get the number of vertices (returned by `size()`) by querying the number of vertices in the mapping.

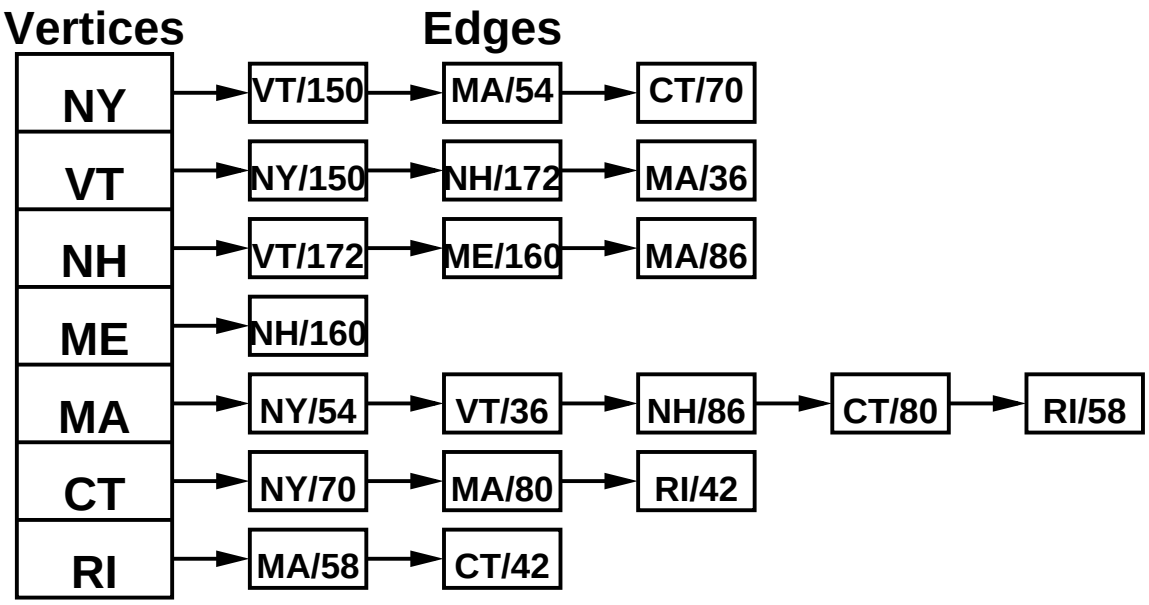
The number of edges can't be determined in the abstract class, so it is an abstract method and is defined appropriately in the subclasses.

We can compute the degree of a vertex by looking across its row and counting up the non-`null`s. If the graph is directed, this will be either in- or out-degree, depending on how we orient the matrix, and if we want the other, it would have to be provided in a separate method.

There is also a method `neighbors` to get an iterator over all vertices adjacent to a given vertex.

Adjacency List Representation

An adjacency list is composed of a list of vertices. Associated with each each vertex is a linked list of the edges adjacent to that vertex.



Once again, the implementation is broken into an abstract class that provides the data and functionality that are common to both the directed and undirected case, and concrete classes that implement the specifics for each directed-ness.

See Structure Source:
`/home/cs501/src/structure5/GraphList.java`

See Structure Source:
`/home/cs501/src/structure5/GraphListDirected.java`

See Structure Source:
`/home/cs501/src/structure5/GraphListUndirected.java`

In `GraphList`, we see that the graph needs to contain a collection of vertices.

This collection could be vector or linked list, but we'll use something more clever. Again, we will consider an efficient way to do this after our discussion of graphs.

Each vertex holds collection of edges that are adjacent to it.

Similarly the list of edges could be implemented in many ways, including all kinds of lists or binary search trees. We'll use singly-linked lists.

See Structure Source:
`/home/cs501/src/structure5/GraphListVertex.java`

There's a lot more going on here than there was in the `GraphMatrixVertex`.

Addition of an edge to a vertex's singly-linked list of edges will always be done at beginning of list (constant time, once we find the vertex).

Edges connected to a given vertex can be held in order by key, but we do not do this.

For directed graphs, we only need to store an edge in one vertex's list. For undirected, each edge

is inserted into two lists.

Back to the `GraphList` abstract class. Again, we implement those things that are independent of directed-ness.

The constructor doesn't need to do as much, and doesn't allocate much space ($O(1)$, though we haven't yet seen the details of the `HashTable` implementation of a `Map`).

Adding vertices is just the addition of a new entry in the mapping.

Remove needs to be done in the subclasses, since we must remove the vertex from all edge lists in which it appears (see below).

Many operations on edges depend on the directed-ness.

Some operations that we could implement in the abstract class for the adjacency matrix representation need to be implemented in the subclasses.

Some others have been moved into the vertex implementation.

First, we'll look more at `GraphListUndirected`.

Adding edges is relatively straightforward: just add it to the adjacency lists of both vertices if it is not already there.

Notice how deleting a vertex is expensive since we must delete all adjacent edges which are in each neighboring vertex. Fortunately, we don't have to check for the edge in all vertex edge lists, only the neighbors of the vertex being removed.

Deleting an edge requires a search of the appropriate vertex edge list(s).

What about space usage? The adjacency matrix representation is more efficient for relatively dense graphs. The adjacency list representation is more efficient (space-wise) for sparse graphs.

Graph Applications

Example: Reachability

As a simple example of something we can do with a graph, we determine the subset of the vertices of a graph $G = (V, E)$ which are *reachable* from a given vertex s by traversing existing edges.

A possible application of this is to answer the question "where can we fly to from ALB?". Given a directed graph where vertices represent airports and edges connect cities which have a regularly-scheduled flight from one to the next, we compute which other airports you can fly to from the starting airport. To make it a little more realistic, perhaps we restrict to flights on a specific airline.

For this, we will make use of the "visited" field that we have included in our implementation of vertices and edges.

We start with all vertices markes as unvisited, and when the procedure completes, all reachable vertices are marked as visited.

See Example:
`/home/cs501/examples/Reachability`

This will visit the vertices starting from s in a *breadth-first order*.

If we replace `toVisit` by a stack, we will visit vertices in a *depth-first order*.

There is a recursive version in the text that performs a depth-first reachability, with the stack implicit in the recursion.

The cost of this procedure will involve at most $\Theta(|V|+|E|)$ operations if all vertices are reachable, which is around $\Theta(|V|^2)$ if the graph is dense.

We can think about how to extend this to find reasonable flight plans, perhaps requiring that all travel takes place in the same day and that there is a minimum of 30 minutes to transfer.

Example: Transitive Closure

Taking the *transitive closure* of a graph involves adding an edge from each vertex to all reachable vertices. We could do this by computing the reachability for each vertex, in turn, with the algorithm above. This would cost a total of $\Theta(|V|^3)$.

A more direct approach is due to Warshall.

We modify the graph so that when we're done, for every pair of vertices u and v such that v is reachable from u, there is a direct edge from u to v.

Note that this is a destructive process! We modify our starting graph.

The idea is that we build the transitive closure iteratively. When we start, we know that edges exist between any vertices that are connected by a path of length 1.

We can find all pairs of vertices which are connected by a path of length 2 (2 edges) by looking at each pair of vertices u and v and checking, for each other vertex, whether there is another vertex w such that u is connected to w and w is connected to v. If so, we add a direct edge u to v.

If we repeat this, we will then find pairs of vertices that were connected by paths of length 3 in the original graph. If we do this $|V|$ times, we will have all possible paths added.

The text has an example Java method (in `bookExamples.java`) that will compute this using this basic idea, though it reorders the loops to gain some efficiency.

Note: I believe that the inner iterators used by the text's method need to be recreated or reset after each iteration of the outer loops.

The outermost loop is over the "intermediate" vertices (the w's), and inner loops are over u and v.

This is still a $\Theta(|V|^3)$ algorithm, though efficiency improvements are possible.

Here is psuedocode for Warshall's algorithm as an operation directly on an adjacency matrix representation of a graph, where each entry is a boolean value indicating whether an edge exists.

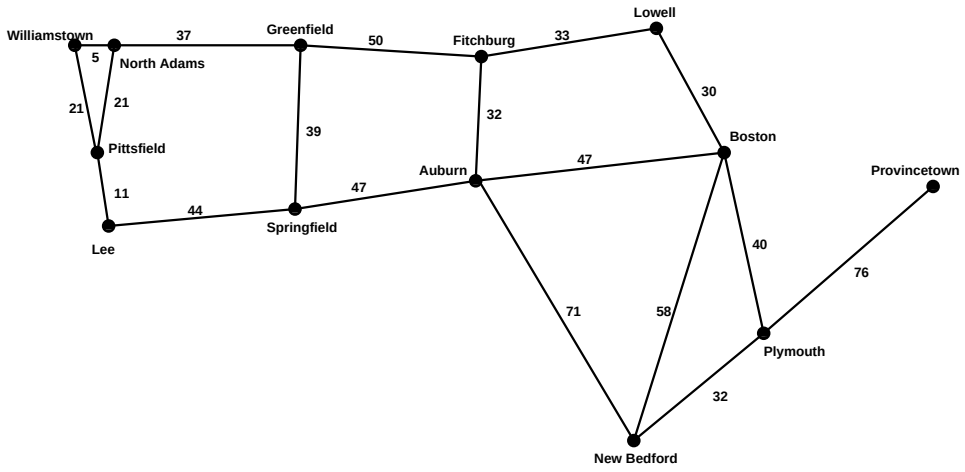
```
warshall(A[1..n][1..n])
```

```
// Each R{i}[1..n][1..n] is an iteration toward the closure
R{0} = A
for k=1 to n
  for i=1 to n
    for j=1 to n
      R{k}[i][j] = R{k-1}[i][j] OR
                   (R{k-1}[i][k] AND R{k-1}[k][j])
return R{n}
```

Example: All Pairs Minimum Distance

We can expand this idea to get *Floyd's Algorithm* for computing minimum distances between all pairs of (reachable) vertices.

For the example graph:



We can use the same procedure (three nested loops over vertices) as we did for Warshall's Algorithm, but instead of just adding edges where they may not have existed, we will add or modify edges to have the minimum cost path (we know of) between each pair.

The pseudocode of this algorithm below again works directly on the adjacency matrix, now with weights representing the edges in the matrix.

```
floyd(W[1..n][1..n])
  D=W  // matrix copy
  for k=1 to n
    for i=1 to n
      for j=1 to n
        D[i][j] = min{D[i][j],D[i][k]+D[k][j]}
  return D
```

Like Warshall's Algorithm, this algorithm's efficiency class is $\Theta(|V|^3)$.

Notice that at each iteration, we are overwriting the adjacency matrix.

And we can see Floyd's Algorithm in action using the above simple graph.

See Example:
`/home/cs501/examples/MassFloyd`

Example: Dijkstra's Algorithm

Dijkstra's Algorithm is a procedure to find shortest paths from a given vertex s in a graph G to all other vertices. The algorithm incrementally builds a sub-graph of G which is a tree containing shortest paths from s to every other vertex in the tree. A step of the algorithm consists of determining which vertex to add to the tree next.

This is a variant of the approach in the text and the approach you will use in the last programming project.

Basic structures needed:

1. The graph $G = (V, E)$ to be analyzed.
2. The tree, actually stored as a map, T. Each time a shortest path to a new vertex is found, an entry is added to T associating that vertex name with a pair indicating the total minimum distance to that vertex and the last edge traversed to get there.
3. A priority queue in which each element is an edge (u, v) to be considered as a path from a located vertex u and a vertex v which we have not yet located. The priority is the total distance from the starting vertex s to v using the known shortest path from s to u plus the length of (u, v).

The algorithm proceeds as follows:

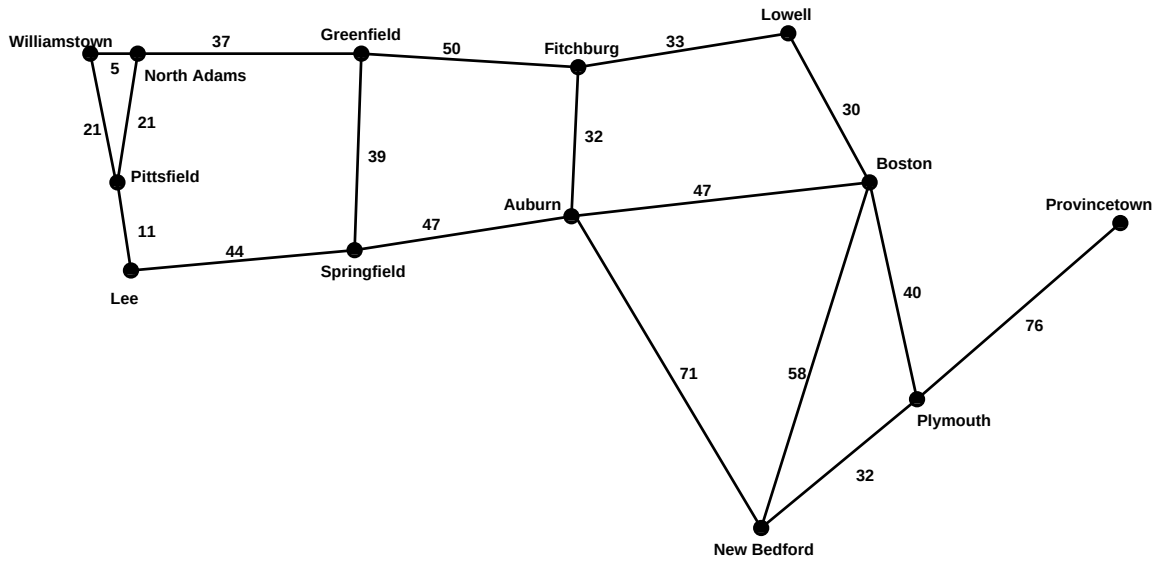
```
T is an empty map;
PQ is an empty priority queue;
All vertices in V are marked unvisited;
Add s to T with a total distance of 0 and a null previous edge;
mark s as visited in G;
Add each edge (s,v) of G to PQ with appropriate value
while (T.size() < G.size() and PQ not empty)
       do
           nextEdge = PQ.remove();
       until(one vertex of nextEdge is visited and the other is unvisited)
          or until there are no more edges in PQ

       // assume nextEdge = (v,u) where v is visited (in T) and u is
          unvisited (not in T)
```

```
Add u to T; mark u as visited in G;
Add (u,v) to T;
for each unvisited neighbor w of u
   add (u,w) to PQ with appropriate weight
```

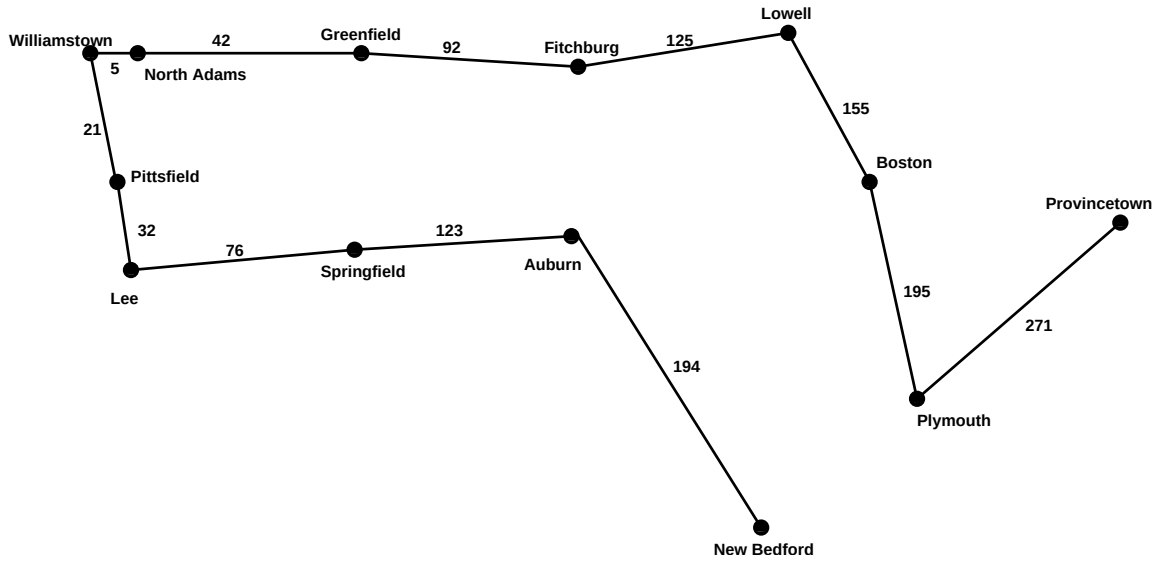
When the procedure finishes, T should contain all vertices reachable from s, along with the last edge traverest along the shortest path from s to each such vertex.

Disclaimer: Many details still need to be considered, but this is the essential information needed to implement the algorithm.

Consider the following graph:



From that graph, the algorithm would construct the following tree for a start node of Williamstown. Costs on edges indicate total cost from the root.


Williamstown
42
Greenfield
92
Fitchburg
125
Lowell
5
North Adams
155
21
Boston
Pittsfield
Provincetown
32
76
123
Springfield
Auburn
195
271
Lee
194
Plymouth
New Bedford

We obtain this by filling in the following table, a map which has place names as keys and pairs indicating the distance from Williamstown and the last edge traversed on that shortest route as values.
It is easiest to specify edges by the labels of their endpoints rather than the edge label itself.

Place	(distance,last-edge)
W'town	(0, `null`)
North Adams	(5, W'town-North Adams)
Pittsfield	(21, Williamstown-Pittsfield)
Lee	(32, Pittsfield-Lee)
Greenfield	(42, North Adams-Greenfield)
Springfield	(76, Lee-Springfield)
Fitchburg	(92, Greenfield-Fitchburg)
Auburn	(123, Springfield-Auburn)
Lowell	(125, Fitchburg-Lowell)
Boston	(155, Lowell-Boston)
New Bedford	(194, Auburn-New Bedford)
Plymouth	(195, Boston-Plymouth)
Provincetown	(271, Plymouth-Provincetown)

The table below shows the evolution of the priority queue. To make it easier to see how we arrived at the solution, entries are not erased when removed from the queue, just marked with a number in the "Seq" column of the table entry to indicate the sequence in which the values were removed from the queue. Those which indicate the first (and thereby, shortest) paths to a city are shown in bold.

(distance,last-edge)	Seq
(5, Williamstown-North Adams)	1
(21, Williamstown-Pittsfield)	2
(26, North Adams-Pittsfield)	3
(42, North Adams-Greenfield)	5
(32, Pittsfield-Lee)	4
(76, Lee-Springfield)	6
(81, Greenfield-Springfield)	7
(92, Greenfield-Fitchburg)	8
(123, Springfield-Auburn)	9
(124, Fitchburg-Auburn)	10
(125, Fitchburg-Lowell)	11
(194, Auburn-New Bedford)	14
(170, Auburn-Boston)	13
(155, Lowell-Boston)	12
(213, Boston-New Bedford)	16
(195, Boston-Plymouth)	15
(226, New Bedford-Plymouth)	17
(271, Plymouth-Provincetown)	18

From the table, we can find the shortest path by tracing back from the desired destination until we work our way back to the source.