Computer Science 431
Algorithms
The College of Saint Rose
Spring 2015

Topic Notes: Graph Algorithms

Our next few topics involve algorithms that operate on graph structures.

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 is included in the "structure" implementations of graph 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/cs431/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*, but the result will be the same in the end.

There is a recursive version in the Bailey 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.

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 Bailey 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.

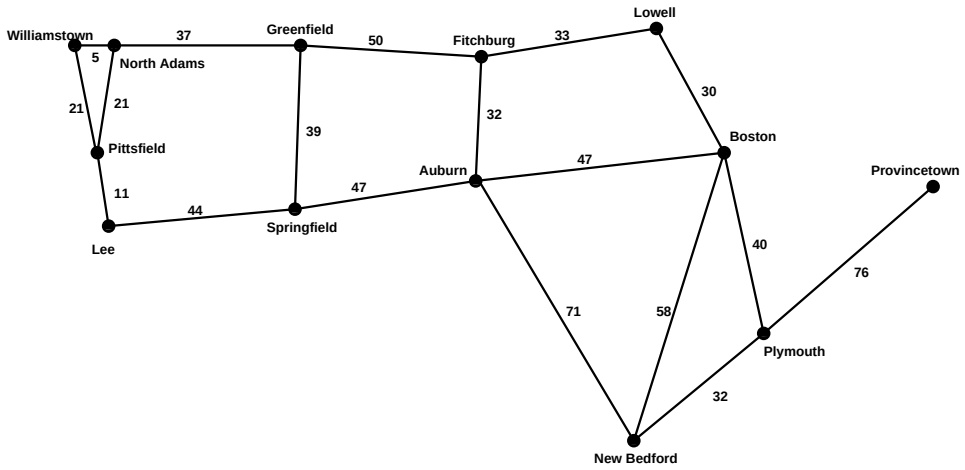
Here is psuedocode for Warshall's algorithm as an operation directly on an adjacency matrix representation of a graph, where each entry is a boolean value indicating whether an edge exists.

```
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}
```

All Pairs Minimum Distance

We can expand just a bit on the idea of Warshall's Algorithm to get *Floyd's Algorithm* for computing minimum distances between all pairs of (reachable) vertices – the *all sources shortest path problem*.

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, so again we have a destructive algorithm.

And we can see Floyd's Algorithm in action using the above simple graph with this program:

See Example:
`/home/cs431/examples/MassFloyd`

Minimum Spanning Tree (MST) Algorithms

Before we look at more graph algorithms, we think about a general class of algorithms to which many of them belong: *greedy algorithms*.

The idea of a greedy technique is one which constructs a solution to an optimization problem one piece at a time by a sequence of choices which must be:

- feasible
- locally optimal
- irrevocable

In some cases, a greedy approach can be used to find an optimal solution, but in many cases it does not. Even in those cases, it often leads to a reasonably good solution in much less time than would be required to find the optimal.

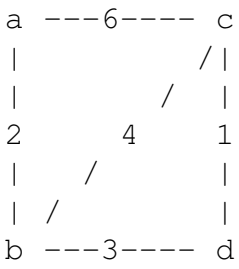
In our first example, a greedy approach does find an optimal solution.

The problem is to find the *minimum spanning tree* of a weighted graph.

The *spanning tree* of a connected graph G is a connected acyclic subgraph of G that includes all of G's vertices.

The *minimum spanning tree* of a weighted, connected graph G is the spanning tree of G (it may have many) of minimum total weight.

We can construct examples of spanning trees and find the minimum using the graph:



The following greedy approach, known as *Prim's algorithm*, will compute the optimal answer as follows:

- Start with a tree T_1, consisting of any one vertex.
- We "grow" the tree one vertex at a time to produce the MST through a series of expanding subtrees T_1, T_2, ..., T_n.
  - On each iteration, we construct T_{i+1} from T_i by adding the vertex not in T_i that is "closest" to those already in T_i (this is a "greedy" step).
  - The algorithm will use a priority queue to help find the nearest neighbor vertices to be added at each step.
- Stop when all vertices are included in the tree.

The text proves by induction that this construction always yields the MST.

Efficiency:

- $\Theta(|V|^2)$ for the adjacency matrix representation of graph and an array implementation of the priority queue.
- $\Theta(|E| \log |V|)$ for an adjacency list representation and a min-heap implementation of the priority queue.

A second optimal, greedy algorithm for finding MST's, *Kruskal's Algorithm*, is in the text and you will consider it as part of the next problem set.

Single-Source Shortest Path

Dijkstra's Algorithm is a procedure to find shortest paths from a given vertex s in a graph G to all other vertices – the *single-source shortest path problem*.

The algorithm incrementally builds a sub-graph of G which is in fact 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 Bailey text and the approach you will use when you code this.

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:

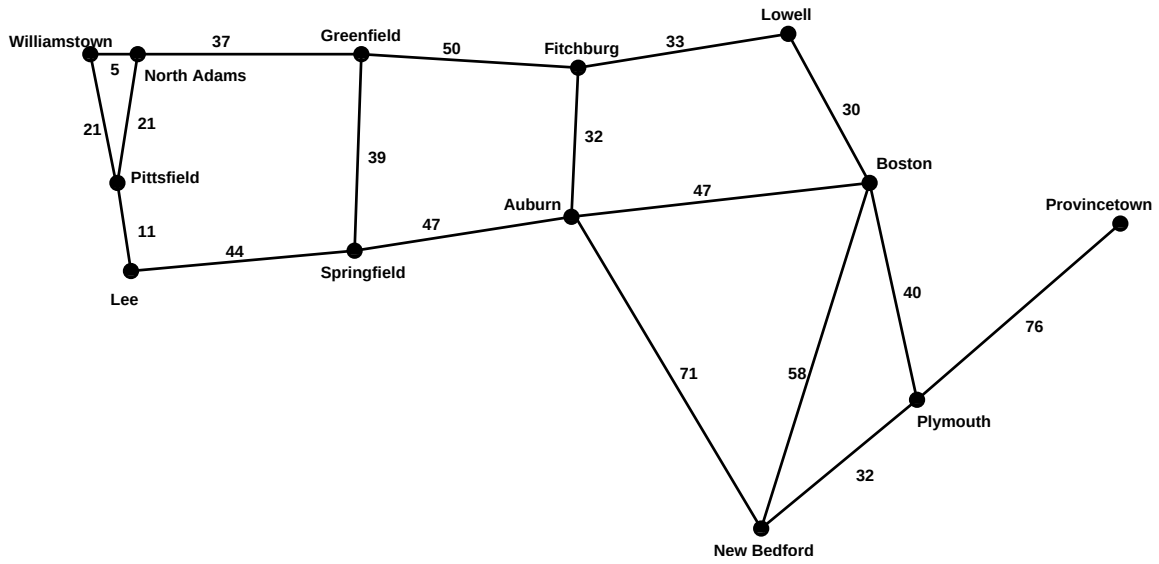
```
T is an empty map;
PQ is an empty priority queue;
All vertices in V are marked unvisited;
Add s to T with a total distance of 0 and a null previous edge;
mark s as visited in G;
Add each edge (s,v) of G to PQ with appropriate value
while (T.size() < G.size() and PQ not empty)
       do
           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
```

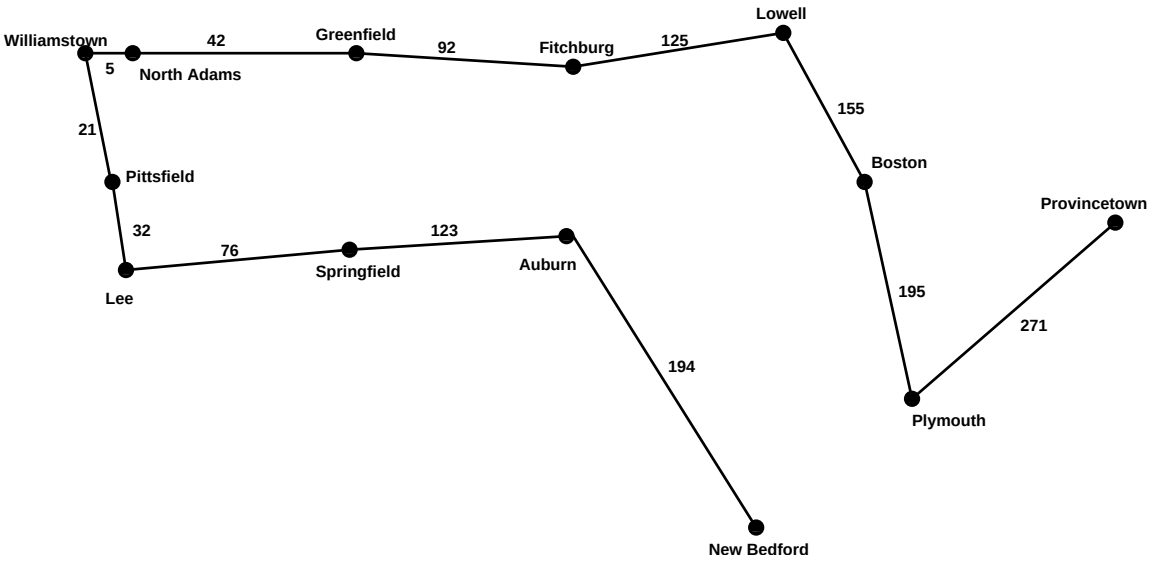
When the procedure finishes, T should contain all vertices reachable from s, along with the last edge traversed along the shortest path from s to each such vertex.

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
123
76
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.