SIENAcollege
Computer Science

Computer Science 385
Design and Analysis of Algorithms
Siena College
Spring 2019

Topic Notes: Greedy Algorithms

Consider the problem of making change for a certain total amount, given a set of coin (and/or bill) denominations. If we use just the common U.S. coins: quarters, dimes, nickels, and pennies, it's an easy problem to solve:

- Use as many quarters as you can.
- If you haven't reached the total, use as many dimes as you can.
- If you haven't reached the total, use as many nickels as you can.
- If you haven't reached the total, fill in with as many pennies as you need.

For this version of the problem, the technique described above is guaranteed to get the solution that requires the smallest total number of coins.

Now, suppose we added in a 9-cent coin and want to make 18 cents total. By extending the previous, you would take 0 quarters, 1 dime, 0 9-cent pieces, 1 nickel, and 3 pennies, for a total of 5 coins. But you have surely observed that it would be better to use 2 9-cent coins. So our original approach in this case does not lead to the fewest number of coins, but at least we found a solution.

The situation could be worse if we have other sets of denominations. Suppose we replace the penny with a 2-cent piece, giving denominations of 25, 10, 5, and 2. We can make change for almost any amount, but our approach of using the largest possible coin at each step will fail to find some of those combinations. An easy one to see is 6 cents. Once we've taken a nickel, there is no way to get to 6 cents, even though we clearly could have done so by skipping the nickel and taking 3 of the 2-cent coins.

The algorithm above where we always take as many of the largest possible coin as possible at each step is an example of a general class of algorithms called *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.

In other cases, like certain instances of the change-making problem, it will fail to find a solution where one exists.

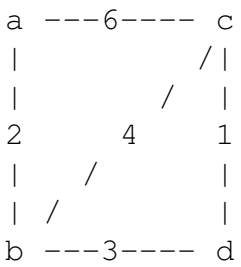
Minimum Spanning Tree (MST) Algorithms

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 an upcoming lab.

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, in our case 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:

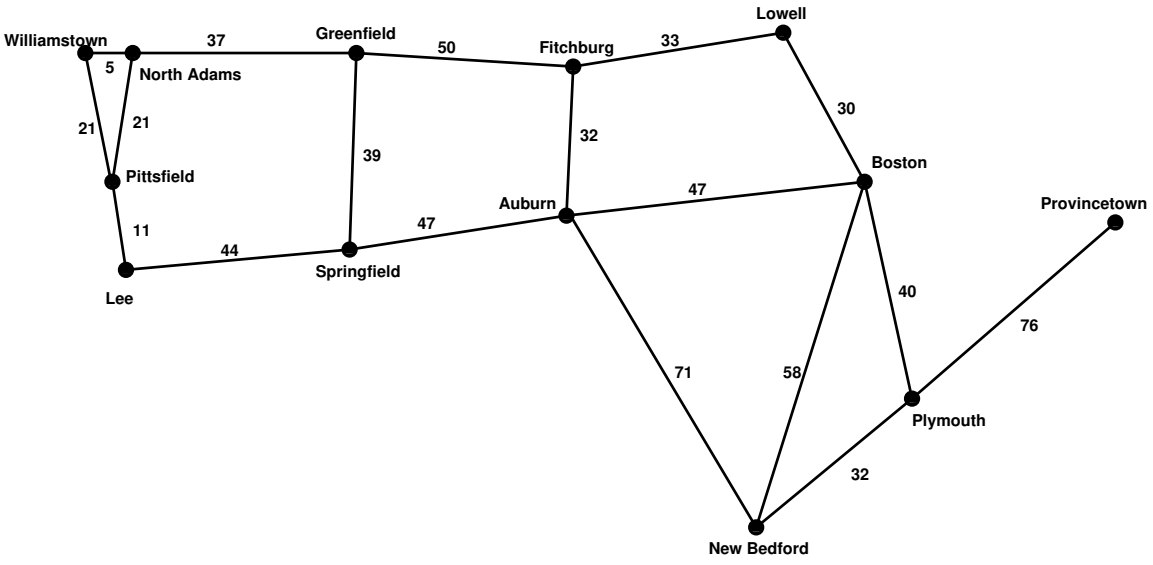
ALGORITHM DIJKSTRA(G, s)
//Input: $G = (V, E)$ a graph with weighted edges
//Input: $s \in V$, the starting vertex
$T \leftarrow$ an empty map
$PQ \leftarrow$ an empty priority queue
// mark all vertices in V as unvisited
for all $v \in V$ **do**
    $v.visited \leftarrow$ **false**
// s is found at distance 0
$T.add(s, (0, null))$
$s.visited \leftarrow$ **true**

// add each edge from s to PQ
for all $(s, v) \in E$ **do**
    $PQ.add((s, v), (s, v).cost)$
// main loop
while $T.size < G.size$ **and not** $PQ.empty$ **do**
    // remove edges from PQ until empty or we find one connecting
    // a visited vertex to an unvisited vertex
    repeat
        $(u, v) \leftarrow PQ.remove$
    until $u.visited \neq v.visited$ **or** $PQ.empty$
    // WLOG, assume u is visited (*i.e.*, is in T) and v is
    // unvisited (not in T), meaning we found a way to v
    $vcost \leftarrow T.get(v).cost + (u, v).cost$
    $T.add(v, (vcost, (u, v)))$
    $v.visited \leftarrow$ **true**
    // for each w, a vertex adjacent to v
    for all $(v, w) \in E$ **do**
        if $w.visited =$ **false then**
            $PQ.add((v, w), vcost + (v, w).cost)$
return T

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.

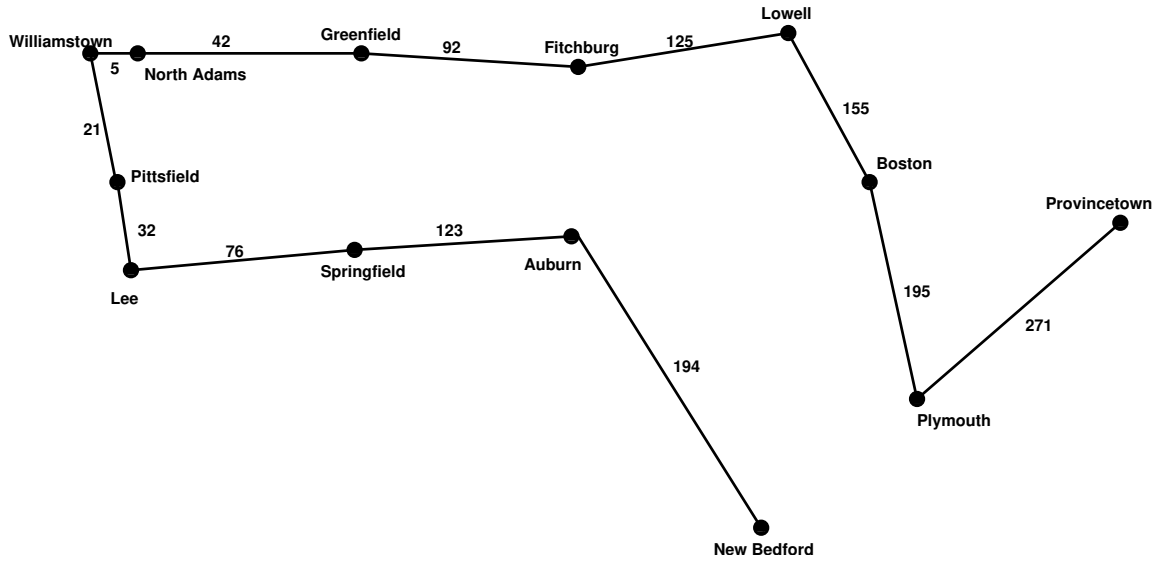
Disclaimer: Many details still need to be considered, but this is the essential information needed to implement the algorithm. But this is good to work through some examples.

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.



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.