Homework 7: Graphs
Due: 5:00 PM, Apr 23, 2018

Contents

1 Dijkstra’s Algorithm 2
   1.1 Shortest Distance 2
   1.2 Graph Implementation 2

2 Negative Edge Weights 4

3 Bellman-Ford Runtime 4

Objectives

By the end of this homework, you will know:

- all about negative edge weights

By the end of this homework, you will be able to:

- find the shortest path in a graph

How to Hand In

For this (and all) homework assignments, you should hand in answers for all the non-practice questions. For this homework specifically, this entails answering the Dijkstra’s Algorithm, Negative Edge Weights, and Bellman-Ford Runtime questions.

In order to hand in your solutions to these problems, they must be stored in appropriately-named files with the appropriate package header in an appropriately-named directory. The source code files should comprise the hw07.src package, and your solution code files, the hw07.sol package.

Begin by copying the source code from the course directory to your own personal directory. That is, copy the following files from /course/cs0180/src/hw07/src/*.scala to ~/course/cs0180/workspace/scalaProject/src/hw07/src:

- IGraph.scala
- DirectedGraph.scala
- IDijkstra.scala
Do not alter these files!

After completing this assignment, the following solution files should be in your 
 ~/course/cs0180/workspace/scalaProject/sol/hw07/sol directory:

- **Dijkstra’s Algorithm**
  - Dijkstra.scala containing
    * class Dijkstra, which extends IDijkstra with attributes (i.e., class parameters)
      graph: DirectedGraph and source: Vertex, with the methods:
      - findShortestDistance(destination: Vertex): Option[Double]
      - findShortestPath(destination: Vertex): Option[Seq[Vertex]]
  - DijkstraTest.scala containing class DijkstraTest and object DijkstraTest

- **Negative Edge Weights**
  - NegativeEdgeWeights.tex, which contains your answers to the Negative Edge Weights analysis question.

- **Bellman-Ford Runtime**
  - Bellman-Ford.tex, which contains your answers to the Bellman-Ford Runtime analysis question.

To hand in your files, navigate to the ~/course/cs0180/workspace/scalaProject/ directory, and run the command `cs018_handin hw07`. This will automatically hand in all of the above files. Once you have handed in your homework, you should receive an email, more or less immediately, confirming that fact. If you don’t receive this email, try handing in again, or ask the TAs what went wrong.

**Problems**

1 **Dijkstra’s Algorithm**

In this problem, you will be implementing Dijkstra’s algorithm for finding a shortest path from a source vertex to all other vertices in a graph.

1.1 **Shortest Distance**

Lois Lane, an up-and-coming reporter for the Daily Planet, is assigned to keep track of heroes, and report on them regularly. To do this in a quick and efficient manner, she decides to create a Directed Graph, where the vertices represent the houses of heroes, and the edges are the distances between the houses.

Take a look at the (not to scale) graph depicted in the figure below. Lois wants to go from Fortress of Solitude to the Batcave. What is her shortest path? Is it through Themyscira or Atlantis?
Given this problem, Lois can transform it into a Directed Graph, and use Dijkstra’s algorithm to find the quickest path from the Fortress of Solitude to the Batcave. The algorithm takes as input a source vertex, \( s \), and calculates the shortest distance from that source vertex to all other vertices in the graph. The pseudocode for Dijkstra’s algorithm was presented in lecture, and can be found in the corresponding lecture notes.

### 1.2 Graph Implementation

We have provided the `DirectedGraph` class for you, which represents a weighted, directed graph. A directed graph is comprised of a set of vertices, each of which is represented by a `Vertex`. Each `Vertex` maintains its own collection of connections to other vertices, which are represented by edges. Each `Edge` has an associated weight and `Vertex`. Each `Vertex` has a specific index associated with it, and each `Vertex` has a parent, which has type `Vertex option`. The parent field will be populated as you implement Dijkstra’s algorithm later in the homework. The final value of the parent will be either `None` if the current `Vertex` cannot be reached from the source vertex, or it will be the closest `Vertex` in the shortest overall path to the current `Vertex` from the source `Vertex`.

**Task:** Create a class `Dijkstra` that extends `IDijkstra`. This class needs to take in a `DirectedGraph` and a source `Vertex` as its inputs. Create a field in the class called `distances` (a `HashMap[Vertex, Double]`). Since this class implements `IDijkstra`, it will eventually need to implement `findShortestDistance` and `findShortestPath`. These methods will allow someone to find the shortest distances from the source vertex to other nodes, and find the shortest paths from this source vertex to other nodes. For now you may leave these methods unimplemented.

**Task:** First, implement Dijkstra’s algorithm in a helper method for this class named `dijkstra`. The method should take no parameters and use the class’s `DirectedGraph` and source vertex to populate the `distances` `HashMap` that will store distances from the source vertex to each vertex in the graph. Specifically, each `Vertex` key in the `HashMap` maps to the distance from the source node to that `Vertex`. Populate this `HashMap` using Dijkstra’s algorithm.

**Note:** The `Vertex` class has an index field - you don’t have to worry about error checking this. You can assume each of the index values for the vertices are distinct, and range from 0 to \( n - 1 \).
Note: You can assume that the source and destination are vertices in the graph, and that all edge weights are non-negative.

Task: Implement a method findShortestDistance that uses your implementation of Dijkstra’s algorithm to find the shortest distance from a source vertex to a destination. Your method should return the distance, optionally, meaning if no path exists, it should return None.

Task: Finally, implement a method findShortestPath that again uses your implementation of Dijkstra’s algorithm to find the shortest path from a source vertex to a destination. Your method should again return the path, optionally, meaning if no path exists, it should return None.

Note: You should include both the origin and destination nodes in your result.

2 Negative Edge Weights

You and your friend learned in lecture that Dijkstra’s algorithm doesn’t work on graphs with negative edge weights (why not?), but after some thought, your friend had an idea:

Add a large positive constant to every edge weight so that the “revised” edge weights are all positive. To find the shortest path between two vertices in the original graph, run Dijkstra’s algorithm on the revised graph, in which all weights are positive, and return the shortest path in this revised graph.

Task: Does your friend’s scheme work? If so, give an informal proof of correctness (i.e., justify that your algorithm works); if not, give a counterexample.

3 Bellman-Ford Runtime

You learned about the Bellman-Ford algorithm in class. Unlike Dijkstra, Bellman-Ford can handle negative edge weights. However, that additional flexibility comes at a price. As a refresher, here is the recurrence relation, where \( s \) is a fixed starting node, \( d \) is an arbitrary destination node, and \( l \) is the maximum path length:

\[
BF_s(d, l) = \begin{cases} 
0, & \text{if } s = d \\
\infty, & \text{if } s \neq d \text{ and } l = 0 \\
\min(BF_s(d, l - 1), \min_{(u, d) \in E} (BF_u(l - 1) + w(u, d))), & \text{otherwise}
\end{cases}
\]

What is the runtime of Bellman-Ford in the worst case? To answer this question, we’ll start naively: there are \( O(n^2) \) subproblems to solve—i.e., \( O(n^2) \) cells in the bottom-up dynamic programming table, where \( n \) is to the number of vertices, and \( m \) the number of edges.

Looking at the recurrence, all cells except the base cases receive the smallest value among a set of candidates. In each case, there are no more candidates than there are edges. So it’s definitely true that Bellman-Ford can’t do worse than \( O(n^2 \times m) \). But can it do better?

Task: In one to two paragraphs, explain why or why not. If you find that the runtime can be improved, state the potential better runtime explicitly (ex: \( O(\log(n)) \)), though note that this is
incorrect). If there are any types of graphs which would particularly benefit from this improvement, describe them.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/cs018/feedback](http://cs.brown.edu/courses/cs018/feedback)