Homework 7: Graphs
Due: 5:00 PM, Apr 19, 2019

Contents

1 Exam Conditions and Collaboration Policy
1.1 Exam Conditions for Implementing Dijkstra

2 Dijkstra's Algorithm (Exam Conditions)
2.1 Shortest Distance
2.2 Graph Implementation
2.2.1 Code Skeleton
2.3 Testing Expectations

3 Negative Edge Weights

4 Dijkstra as Dynamic Programming?

Objectives

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

- find the shortest path in a graph

By the end of this homework, you will know:

- how Dijkstra’s algorithm behaves with negative edge weights
- how Dijkstra’s algorithm relates to dynamic programming

1 Exam Conditions and Collaboration Policy

For this assignment, implementing Dijkstra’s algorithm will be under exam conditions. The other two problems are under our regular collaboration policy; this means that you can, and should, discuss these concepts!

1.1 Exam Conditions for Implementing Dijkstra

The exam conditions here are identical to those on the heaps homework. Specifically:
• No collaboration of any kind is allowed on this portion of the homework.
• TAs cannot answer any questions beyond clarification ones.
• To avoid inconsistent answers across staff, all questions on the first part of the assignment should be posted privately on Piazza. Piazza will default all posts to private while this assignment is out.

How to Hand In

In order to hand in your solutions, 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 \texttt{hw07.src} package, and your solution code files, the \texttt{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 \texttt{/course/cs0180/src/hw07/src/*\.scala} to \texttt{~/course/cs0180/workspace/scalaproject/src/hw07/src}:

- \texttt{IGraph.scala}
- \texttt{DirectedGraph.scala}
- \texttt{IDijkstra.scala}

Make sure to also copy our sol code over as well. Copy the following files from \texttt{/course/cs0180/sol/hw07/sol/*\.scala} to \texttt{~/course/cs0180/workspace/scalaproject/sol/hw07/sol}:

- \texttt{ConstructorTest.scala}

Do not alter these files!

After completing this assignment, the following solution files should be in your \texttt{~/course/cs0180/workspace/scalaproject/sol/hw07/sol} directory:

- Dijkstra's Algorithm
  - \texttt{Dijkstra.scala} containing
    * \texttt{class Dijkstra, which extends IDijkstra with attributes (i.e., class parameters) graph: DirectedGraph and source: Vertex)}, with the methods:
      - \texttt{\textbackslash findShortestDistance(destination: Vertex): Option[Double]}
      - \texttt{\textbackslash findShortestPath(destination: Vertex): Option[List[Vertex]]}
  - \texttt{DijkstraTest.scala} containing \texttt{class DijkstraTest} and \texttt{object DijkstraTest}
  - \texttt{ConstructorTest.scala} containing \texttt{class ConstructorTest} and \texttt{object ConstructorTest}

- Negative Edge Weights
2 Dijkstra’s Algorithm (Exam Conditions)

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.

2.1 Shortest Distance

Ash Ketchum, an up-and-coming Pokémon trainer, is traveling across the United States to complete his American Pokédex via Amtrak. To do this in a quick and efficient manner, he decides to create a Directed Graph, where the vertices represent the different cities along the route, and the edges are the distances between the cities.

Take a look at the (not to scale) graph depicted in the figure below. Ash wants to go from Seattle to Orlando. What is his shortest path? Is it through San Antonio or Chicago?

Given this problem, Ash can transform it into a Directed Graph, and use Dijkstra’s algorithm to find the quickest path from Seattle to Orlando. 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.

2.2 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 (another common term for Nodes), 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 id 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 (it is taking the place of the parent HashMap outlined in lecture). 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 (a skeleton of this class appears later in the handout). This class takes 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, fill in the implementation of Dijkstra’s algorithm in the method named dijkstra in the code skeleton. 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 id field - you don’t have to worry about error checking this. You can assume each of the id values for the vertices are distinct, and range from 0 to n – 1, where n is the total number of vertices in the graph.

Note: Assume that the source and destination are vertices in the graph, and that all edge weights are non-negative. You do NOT need to do any error checking around these assumptions.

Task: Implement the 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 the 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 resulting path. In the case
where the origin is the destination, only include the node once. If there are multiple paths with the same shortest length, return any one of them as your answer.

2.2.1 Code Skeleton

Here is a skeleton for your Dijkstra class. Copy this into your solution and modify it with your implementation.

```scala
class Dijkstra(graph: DirectedGraph, source: Vertex) extends IDijkstra {

  private val distances = new HashMap[Vertex, Double]()

  def dijkstra() {
    /**
     * Performs Dijkstra's algorithm from the given source Vertex, filling the
     * distances HashMap with distance from the source.
     */
    val ordering = Ordering[Double].on[(Vertex, Double)](-1 * _._2)
    val pq = PriorityQueue[(Vertex, Double)]()(ordering)

    // FINISH FILLING IN HERE
  }

  override def findShortestDistance(destination: Vertex): Option[Double] = {
    // FINISH FILLING IN HERE
  }

  override def findShortestPath(destination: Vertex): Option[List[Vertex]] = {
    // FINISH FILLING IN HERE
  }
}
```

2.3 Testing Expectations

As far as testing goes, you can expect that you are given a valid graph. This graph is not necessarily connected, meaning there may not exist a path between any given pair of vertices.

You may assume that the graph is nonempty, and the input source and graph are not null. You may also assume source is contained in graph. In addition, you may assume all edge-weights are positive (so not zero). You may assume there are no self-loop (edges that connect a vertex to itself).

We will mainly be looking for testing a variety of graphs, mainly “edge-case” graphs. The different edge-cases should cover a variety of potential paths between sources and destinations. If you think you have a good edge case graph, be sure to test it for both findShortestDistance and findShortestPath.

**Task:** Fill out DijkstraTest.scala, following the specifications above.
3 Negative Edge Weights

In class, we only did examples of Dijkstra with positive weights on the edges. Dijkstra's algorithm doesn’t work on graphs with negative edge weights (see the notes for some discussion of why not). 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 argument of correctness (i.e., justify that the algorithm works); if not, give a counterexample.

4 Dijkstra as Dynamic Programming?

This problem asks you to think about the relationship between Dijkstra’s algorithm and dynamic programming.

**Do NOT try to use Dynamic Programming in your implementation for the exam conditions question. For that, follow the implementation strategy outlined in class.**

In Dynamic Programming, we create an array (1 or 2 dimensional) with the dimensions of the problem at hand, then implement a recurrence relation to update cells of the array based on values in adjacent cells.

In some ways, Dijkstra’s algorithm feels similar: the distances HashMap is effectively an array (indexed by vertices instead of numeric indices), and we keep updating the values of the cells based on the values of adjacent vertices in the graph.

This raises a question: **could we have written a recurrence relation for Dijkstra and used either forward or backward dynamic programming to implement the algorithm instead of the priority-queue-based approach that you’re using in your implementation?**

**Task:** Writing a recurrence relation in the style we did for DP problems earlier in the semester does not work well. Your task is to figure out (and explain) why. Answer the following questions as you explain your reasoning.

1. Why does it seem difficult to write a recurrence relation (of the style we used in prior DP problems) for Dijkstra?

2. If we had been able to write a recurrence and implement it as we did for earlier DP problems, how would the run time and space consumption of such an approach compare to the Dijkstra’s algorithm presented in class (the one you are implementing)?

3. Describe the characteristics of a problem that would make it suitable for implementation via Dynamic Programming. Then explain how Dijkstra’s algorithm either does or does not meet those criteria.
**Hint:** Remember to think about the subproblems.

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: [https://cs.brown.edu/courses/cs018/feedback](https://cs.brown.edu/courses/cs018/feedback)