Lab 8 Finding single-source shortest paths in log-linear time by Dijkstra's algorithm

C S 372 Data Structures and Algorithms

November 12, 2019

In this lab, you will implement the Dijkstra's algorithm. You will use two types of data structure to construct a priority queue to be used in the algorithm: a linked list and a min-heap. You will test your implementation and also compare the performance of the two data structures on large graphs.

You have an opportunity to earn 300% extra credit by writing a map routing program that will find a route from an origin to a destination using a world travel map.

1 Requirement for the Dijkstra algorithm

Design the first Dijkstra's algorithm with the following function prototype that uses an unsorted linked list as priority queue:

```
void Dijkstra_list(Graph & G, Node & s)
{
  // Input: graph G, either undirected or directed, and
  //   a source node s

  // Output: the effect of the algorithm is that all nodes
  //   are marked with a distance from s

  // Perform Dijkstra's algorithm on the input graph from
  //   a given node s

  // Use an unsorted linked list for the priority queue
```

```
}
```

Design the second Dijkstra's algorithm with the following function prototype that uses a min-heap as priority queue:

```
void Dijkstra_heap(Graph & G, Node & s)
{
  // Input: graph G, either undirected or directed, and
  //   a source node s

  // Output: the effect of the algorithm is that all nodes
  //   are marked with a distance from s

  // Perform Dijkstra's algorithm on the input graph from
  //   a given node s

  // Use a min-heap for the priority queue
}
```

The effect of the Dijkstra's algorithm on the graph G is distance labels indicating the distance from source node s to every node in the graph. They will be stored within the `Graph` data structure. You can enhance the graph class from previous labs in more than one way. You are free to design your strategy and describe it in your lab report. The distance values must not be saved as global variables.

2 Technical notes

The priority queue is a dynamically changing data structure and its implementation can be subject to memory problems.

1. The graph data structure needs to be expanded to handle the edge weight.
2. The graph file will contain three numbers in a row, where the last number is the weight of the edge defined by the first two nodes. Your code for reading/writing a graph file will need to be updated to handle the weight as well.
3. You will also modify the `random_graph.R` file to randomly generate large positively weighted graphs.

4. When using an unsorted linked list for the priority queue, the remove-min operation can be achieved by finding the minimum element and then removing it from the list. To traverse a list, you cannot use an index but you can use an iterator. To remove an element from a list, you can use the `list::erase()` function.

5. The C++ function `make_heap()` by default makes a max-heap, not min-heap. You can make a min-heap by specifying a customized comparison function optional to `make_heap()`.

6. Your code must determine the position of a node in the heap in constant time before calling the decrease-key operation on the heap. You can maintain a heap index for each node to resolve this issue.

7. As the remove-min and decrease-key operations will modify the heap, the positions of each node in the heap must be updated accordingly. You can manage the positions of each node in the heap via the heap index.

8. You can use the following C++ function

   ```
   numeric_limits<double>::infinity()
   ```

   to define infinity ∞ for the initial distance of each node other than s. The header file `limits` must be included to use this function. This option is preferable to either -1 or a very large number from the software engineering point of view.

3 Test your implementations of Dijkstra's algorithms

Generate five examples of weighted graphs to test your code. The graphs do not have to be very large but must represent a variety: cyclic/acyclic, directed/undirected, and some having more than one connected component.

Your C++ program must include a main() function that calls the `testall()` function. When the program is compiled by a C++ compiler it generates a binary executable file that will run when invoked from the command line.

4 Plot the runtimes of the two Dijkstra's implementations

After testing the correctness of the the program, you will study how the runtime scales with the size of graph for the two methods. Use the random graph function you developed in Lab 4 to generate random graphs of increasing sizes.

You will produce four curves in R:

1. `Dijkstra_list` runtime as a function of number of nodes in the graph, given the number of edges.
2. `Dijkstra_heap` runtime as a function of number of nodes in the graph, given the number of edges.
3. `Dijkstra_list` runtime as a function of number of edges in the graph, given the number of nodes.
4. `Dijkstra_heap` runtime as a function of number of edges in the graph, given the number of edges.

Ideally, the first two curves should be plotted in the same figure for easy comparison; and the last two in another figure.

Your algorithm must be able to handle graphs with 10,000 or more nodes and 1,000,000 or more edges in the order of tens of seconds.

5 Extra credit (300%): Shortest routes between any two waypoints on earth

You will apply the Dijkstra's algorithm on a world travel map [Teresco, 2012] (`http://courses.teresco.org/metal/`) to find a shortest path from one place to another. The map contains 587,376 highway waypoints (nodes) and 665,270 road segments (edges) connecting them on the Earth.

5.1 Travel Mapping Graph file format

The map we will use is in a Travel Mapping Graph (tmg) simple format as described in http://courses.teresco.org/metal/graph-formats.shtml:

- The first line specifies the file format. It consists of three space-separated entries, the first of which must be “TMG”. The second is a version number, which much be “1.0”. The third entry is either “simple” or “collapsed” indicating whether the edge data is in simple format or collapsed edge format.
- The second line consists of two numbers: the number of vertices (waypoints) $|V|$, and the number of edges, $|E|$, (road segments that connect adjacent waypoints).
- The next $|V|$ lines describe the waypoints. Each line consists of a string describing a waypoint, followed by its latitude and longitude as floating-point numbers.
- The last $|E|$ lines describe the road segments. Each line consists of two numbers specifying the waypoint numbers (0-based and in the order read in from this file) connected by this road segment, followed by a string with the name of the road or roads that form this segment. For “collapsed” format graphs, a line describing an edge can optionally have a list of the space-separated latitudes and longitudes of shaping points that define a more accurate path (for both mapping and for computing distances) of the road segment.
- A “simple” format graph is simply a “collapsed” format graph with no edge shaping points.

You will download the file `tm-master-simple.tmg` from

`http://travelmapping.net/graphdata/tm-master-simple.tmg`

The file size is about 33.3 MB.

Please note that the graph should be treated as undirected.

5.2 The routing task

You will use the Dijkstra’s algorithm you have just developed to find the shortest path from a given original waypoint to a destination waypoint. For example, if the input pair of waypoints is

NM271@+X478507 35.807616 -104.446678

and
NY11C_N/US11_N 44.776047 -74.672871

you should print out a shortest path from the first waypoint (in New Mexico) to the second (in update New York) in the following fashion:

```
Origin:
NM271@+X478507 35.807616 -104.446678
[road segment 1]
[intermediate waypoint 1]
[road segment 2]
[intermediate waypoint 2]
[road segment 3]
...
Destination:
NY11C_N/US11_N 44.776047 -74.672871
Total distance: 2021 miles
```

If the two waypoints are on road-disconnected continents (e.g., one in Asia one in North America), your program should report that no road is possible to connect the two waypoints.

5.3 The distance between two waypoints

In order to compute the total distance, your code will use the latitude and longitude of each pair of waypoints (lat_1, lon_1) and (lat_2, lon_2) to estimate edge length in miles using the Haversine formula:

$$\Delta lon = lon_2 - lon_1 \tag{1}$$

$$\Delta lat = lat_2 - lat_1 \tag{2}$$

$$a = \sin^2(\Delta lat/2) + \cos(lat_1)\cos(lat_2)\sin^2(\Delta lon/2) \tag{3}$$

$$c = 2 \cdot \text{atan2}(\sqrt{a}, \sqrt{1-a}) \tag{4}$$

$$d = 3961 \cdot c \tag{5}$$

where 3961 is the radius of the Earth in miles.

6 Submission

Write a lab report to describe your lab work done in the following sections:

1. Introduction (define the background, motivation, and the problem),
2. Methods (provide the solutions),

   Describe how you implement the remove-min and decrease-key operations for the priority queue using the list and heap, respectively.

   Derive the theoretical asymptotic runtime of each algorithm.

   Describe your solution to the extra credit problem if done.
3. Results
   (a) visualize your five test graphs using R package `igraph`
   (b) show the four curves for empirical runtime on large graphs. Discuss if they appear to be consistent with theoretical expectation. If not, explain the possible reasons.
4. Discussion (general implications and issues).

   Create a graph with negative weights and show how your Dijkstra's algorithm implementation gives a wrong result.
5. Conclusions (summarize the lab and point to a future direction).

Submit the source code files and your lab report online.

References

[Teresco, 2012] Teresco, J. D. (2012). Highway data and map visualizations for educational use. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (pp. 553–558).: ACM.