<u>Shortest Cycle</u>

Jonathan Wenger

1. **The algorithm for solving this problem. You can, but should not need to, use pseudocode to describe the algorithm (conversational English, if effectively used, is fine).**

The algorithm I used to solve this problem was extremely (if not fully) based on Dijkstra's shortest path algorithm. Assuming the Graph is called Graph G and the Edge of interest is called Edge E, which goes from Vertex A to Vertex B, I started off by iterating through the list of edges to find the highest numbered vertex, so I would know how many vertices were in the graph. I then constructed an edge weighted graph with that many vertices (I adapted the implementation of the EdgeWeightedGraph from the Sedgewick textbook, page 643, making some minor changes, such as changing Bags to Lists, and making it work for undirected graphs). I then added all the edges from the list of edges to the graph, EXCEPT for the edge of interest.

After that, I used Dijkstra's shortest path algorithm to find the shortest path tree from Vertex B. I used the DijkstraSP class from here, making minor changes, such as making it work with the Edge inner class in ShortestCycleBase.java. I also used Sedgewick's IndexMinPQ class, as that is what the DijkstraSP class uses - IndexMinPQ can be found here. After getting the shortest path tree, I called DijkstraSP.pathTo(), with Vertex A passed in as the parameter. The list of edges returned from that method got me the shortest path from Vertex B to Vertex A (NOT using the edge of interest, as that edge was removed from the graph). I then added the edge of interest (Vertex A to vertex B) to the end of that list, and that list of edges is the shortest cycle that included the edge of interest.

2. **Prove the correctness of your algorithm.**

Before I start the proof, just a basic definition: A **cycle** in a graph is a non-empty trail in which only the first and last vertices are equal.

Because in this situation we are dealing with a simple undirected graph, that means there can be (at most) one edge between two vertices. Meaning, the minimum number of edges to create a cycle is three edges.

Assume the edge of interest is called Edge E, and it connects Vertex A to Vertex B. Because we know that the shortest cycle MUST contain the edge of interest, that means the cycle MUST contain the edge connecting A to B. In order for the list of vertices to be a cycle, the list must also include a list of edges (NOT including the edge of interest) that connect Vertex B to Vertex A. It has already been proven that Dijkstra's algorithm finds the shortest path between two given vertices. Because Dijkstra's algorithm finds the shortest path (NOT including the edge of interest) connecting B to A, and the edge of interest connects A to B, this algorithm works to find the shortest cycle with a given edge of interest.

3. **Prove that your algorithm meets the required performance constraints.**

The runtime of this algorithm is O(E(log V)), where V is the number of vertices in the graph and E is the number of Edges in the graph.

- The algorithm starts by finding the highest numbered vertex, to see how many vertices are in the graph. That subroutine has an O(E) runtime, as that subroutine requires iterating through all of the edges in the graph.
- The algorithm then constructs the graph, which we're not factoring into the runtime of this algorithm, as "the time cost to construct a graph is not part of the time complexity constraint" (Piazza @48).
- The algorithm then iterates through all of the edges and adds them to the graph (besides the edge of interest). Iterating through all of the edges is an O(E) subroutine, and adding them to the graph is an O(1) subroutine, making this total part of the algorithm have an O(E*1) runtime, which is just O(E).
- Then, this algorithm runs Dijkstra's algorithm to find the shortest path from Vertex B to Vertex A. The runtime of running Dijkstra's algorithm with one starting vertex (Vertex B) has a runtime of O(E(log V)) (as already proven, see here for more information, specifically the first "proposition" section, as well as the Shortest Path Intro to Algorithms Slides, slide 19).
- After that, the algorithm adds the edge of interest to the end of the list, which is an O(1) operation.

If you take all of the aspects of this algorithm, this algorithm has an O(E+E+E(log V)+1) runtime. However, because we are dealing with "big-O", we only care for the fastest growing part of the algorithm, which is E(log V), so this algorithm runs in O(E(log V)) time. As E(log V) grows slower than (E+V)(log V), this algorithm runs better than O(E+V)(log V)) time.