Mike Damman

Program 3: No Time to Waste

|  |  |  |
| --- | --- | --- |
| **What**  \_ indicates program | **Pts** | **Due Nov 14th** |
| **External Documentation** | **2** |  |
| Your Name | 1 |  |
| Pasting this table into your memo | 1 |  |
| **Analysis (based ONLY the number of intersections and the number of streets)** | **30** |  |
| Time Analysis Worst case O()  of each function with explanation. (main is a function) | 5 |  |
| Space Analysis Worst case O()  of each function with explanation. (main is a function) | 5 |  |
| A test with at least 5 unique tests. | 10 |  |
| Discussion of your choice of data structures including your implementations and what choices you made and what you rejected. | 10 |  |
| **Program Style** | **18** |  |
| Your Name | 1 |  |
| Description of the problem in your own words | 1 |  |
| Overall Style (no global variables etc) use  of functions, classes | 10 |  |
| Pre/Post conditions for functions | 6 |  |
| **Functionality** | **50** |  |

**Analysis:**

**n -** number of vertices/intersections **m -** total number of streets

**Choice of Data Structure:**

Two-Dimensional Vector/Array: I initially began the program, reading all values into a 2D vector. This was largely as a way to easily read the data in, and then process it into a better data structure later. While space complexity would be relatively small, and traversal easy, I dismissed this fairly early on as it would require a large O(n\*(n+m)) time complexity when searching for the next street to include. I did in the end still use a 2D vector to just hold the data as it allowed easy access to each intersection and the routes connected to it. I could do this and avoid the time complexity draw backs by using a priority queue as well (see below).

Min Heap: I next considered using n Min Heaps attached to each intersection to hold the routes. Adding and deleting routes would be O(lg n), however the lowest delay time route would always rise to the top of the heap, allowing for less time searching for the next route. Using the heap in array form, though, I realized there could be cases where I would need to search deeper than the first route, and a heap is not completely ordered to ease that.

Priority Queue: This became the obvious choice for holding the routes. As inserted they became ordered in the queue so I would never have need of any other value than those on top. Furthermore, insertion into the queue takes time complexity constant. Initially I considered using a 2-dimensional priority queue, but I needed the vertices to be more accessible. In the end, I opted to hold the data in a 2D array, then as each intersection was visited, I would copy its streets over to a priority queue. This sorted them, and allowed me to only bother with the ones coming from vertices I had already explored. It came with a slightly larger space requirement per each intersection and street, but resulted in a time complexity of O(n+m).

**Worst Case Time Analysis**

|  |  |  |
| --- | --- | --- |
| **Function** | **Big O: Time** | **Description** |
| main | O(n + m) | The most time demanding process in main takes place in calling the findPath function. |
| findPath | O(n + m) | At its worst, findPath will process each intersection once and each route twice, resulting in O(n + 2m) = O(n + m). |
| printAnswer | O(n) | printAnswer loops through the intersections used for the minimum travel time. At its worst, it can only pass through each intersection once. |

**Worst Case Space Analysis**

|  |  |  |
| --- | --- | --- |
| **Function** | **Big O: Space** | **Description** |
| main | O(n + m) | main creates a vector to hold the intersections and a vector to hold the routes. |
| findPath | O(n + m) | findPath utilizes vectors for intersections and routes, as well as a priority queue holding copies of the routes. This results in  O(n + 2m) = O(n + m) |
| printAnswer | O(n) | printAnswer uses a stack, which at worst can contain an index number for each intersection. |

**Test Plan:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Case** | **Reason for Test** | **Actual Data** | **Predicted Output** | **Actual Output** |
| **1** | Test basic functionality given ordered input. | 3  1 2 8  1 3 4  0  1 3 | 1 2 3 ; 12 | 1 2 3 ; 12 |
| **2** | Test setting necessary route with highest delay time | 4  2 2 5 3 20  2 1 3 4 2  1 4 1  0  1 3 | 1 3 ; 20 | 1 3 ; 20 |
| **3** | Test multiple paths, optimum discovered first. | 5  3 3 4 5 3 4 1  2 3 1 1 2  1 2 4  2 2 3 5 6  1 2 9  1 2 | 1 4 2 ; 4 | 1 4 2 ; 4 |
| **4** | Test multiple paths, non-optimum discovered first | 5  2 3 3 4 6  3 1 2 3 7 5 6  1 4 5  0  1 4 7  2 4 | 2 1 4 ; 8 | 2 1 3 4 ; 10 |
| **5** | Test start and end the same | 2  1 2 4  1 1 5  1 1 | 1 ; 0 | 1 ; 0 |

