# CS 225 Final Project | Results

### File I/O

We tested the file I/O with our own basic .txt files, consisting of words on separate lines, words separated by commas, and a combination of both, all easy to test.

---

### BFS

The basic tests were making sure the constructors worked and that all edges can be accounted for. The main test for BFS was the pathing, and we made sure our implementation worked with a simple graph.

```
38  TEST_CASE("BFS pathing works", "[BFS]") {
39      std::vector<Vertex> vertices = {"one", "two", "three"};
40      BFS bfs(vertices, "one");
41      bfs.addEdge(Edge("one", "two"));
42      bfs.addEdge(Edge("one", "three"));
```

---

### Dijkstra's Algorithm

For the graph constructor, we inputted my own set of vertices and edges and made sure it constructed only those vertices and edges. For the main algorithm code, we inputted custom weighted and unweighted graphs, and the results matched the correct results obtained from a third-party simulation. Different data input formats were tested, specifically "Vertex,Vertex,Weight" and "Vertex,Vertex". We also created a test case for when a graph had a single heavy-weight path and another path consisting of many light-weight edges to make sure my Dijkstra's implementation truly worked.

```
68  //Test for single heavy-weight path vs. many light-weight paths
69  TEST_CASE("test_dijkstras_2", "[dijkstras]") {
70      std::vector<std::vector<std::string>> data_2 = {
71          {"1", "3", "1"},
72          {"3", "4", "1"},
73          {"4", "5", "1"},
74          {"5", "6", "1"},
75          {"6", "7", "1"},
76          {"7", "8", "1"},
77          {"8", "2", "1"},
78          {"1", "2", "10"},
79      };
```

```
Search complete! Extracting path...
1-3
3-4
4-5
5-6
6-7
7-8
8-2
```

Finally we ran my code on the actual dataset.

```
138     //optional longest route of 15 -> 343 -> 6592 -> 581
139     vector<Edge> path_5 = test_5.Dijkstras_Helper("15", "581");
140
```

```
TERMINAL  PROBLEMS  OUTPUT  DEBUG CONSOLE  1: bash

Search complete! Extracting path...
15-3401
3401-581
```

---

**Landmark Path Algorithm**

Our landmark path algorithm was made more efficient with running a BFS on the landmark vertex since our dataset is undirected and unweighted. Testing for this algorithm was similar to testing for Dijkstra's. We made sure the constructors created

only the necessary vertices and edges, and that it worked on different graphs we made.

```
36  TEST_CASE("test_landmark_path", "[landmark]") {
37    std::vector<Edge> testPath = test.runLandmarkPath("1", "5", "0");
38    REQUIRE(testPath.size() == 3);
39    REQUIRE(testPath[0].getLabel() == "0-1");
40    REQUIRE(testPath[1].getLabel() == "0-2");
41    REQUIRE(testPath[2].getLabel() == "2-5");
42  }
43
44  TEST_CASE("test_landmark_path2", "[landmark]") {
45    std::vector<Edge> testPath = test.runLandmarkPath("4", "7", "6");
46    REQUIRE(testPath.size() == 5);
47    REQUIRE(testPath[0].getLabel() == "2-4");
48    REQUIRE(testPath[1].getLabel() == "5-2");
49    REQUIRE(testPath[2].getLabel() == "6-5");
50    REQUIRE(testPath[3].getLabel() == "6-5");
51    REQUIRE(testPath[4].getLabel() == "5-7");
52  }
```

The last test was using the musae-twitch dataset, which worked as intended.

```
54  TEST_CASE("test_landmark_dataset", "[landmark]") {
55    std::vector<std::string> predata = fileio::file_to_vector("tests/data/musae_ENGB_edges.csv");
56    std::vector<std::vector<std::string>> afterdata = fileio::csv_to_tokens(predata);
57
58    Landmark dataTest = Landmark(afterdata);
59    std::vector<Edge> testPath;
60    Graph testGraph = dataTest.getGraph();
61
62    vector<Edge> path = dataTest.runLandmarkPath("15", "581", "15");
63
64    REQUIRE(testGraph.vertexExists("15"));
65  }
```

```
TERMINAL  PROBLEMS  OUTPUT  DEBUG CONSOLE

15-3401
3401-581
```

---

**Final Thoughts**

A nice addition to our project could be taking in data that described each user (vertex), such as account age, frequency of watching mature content, average views of streams,

and twitch partner status. This would allow us to create more complex logic, fleshing out our code and providing data that would, in turn, help make more accurate predictions if the prediction part is ever implemented.