COP 3530 - Project 2 Tests

Aiden Zepp

2023-11-14

Test Cases

The following are the test cases created for the *Project 2* assignment using the *Catch2* testing framework.

Test 1

```
TEST_CASE("GRAPH :: Insert :: Same Nodes For Both Data Fields", "[Insert]")
{
    Graph graph = Graph();
    std::string origin = "";
    std::string target = "";

    for(int i = 0; i < 10; i++)
    {
        graph.Insert(origin, target);
    }

    // There should only be 1 adjacency list stored for both data fields.
    REQUIRE(graph.from.size() == 1);
    REQUIRE(graph.into.size() == 1);

    // There should only be 1 key of `""` stored for both data fields.
    REQUIRE(graph.from.find("") != graph.from.end());
    REQUIRE(graph.into.find("") != graph.into.end());

    // There should only be 10 values in each adjacency list for both data fields.
    REQUIRE(graph.from.at("").size() == 10);
    REQUIRE(graph.into.at("").size() == 10);

    // There should only be values of `""` stored for each adjacency list's values.
    for(int i = 0; i < 10; i++)
    {
        REQUIRE(graph.from.at("").at(i).empty());
        REQUIRE(graph.into.at("").at(i).empty());
    }
}
```

Test 2

```
TEST_CASE("GRAPH :: Insert :: `from` Matches Expectation", "[Insert]")
{
```

```
    Graph graph = Graph();
    std::pair<std::string, std::string> pairs[15] = {
            { "google.com", "google.com" },
            { "google.com", "gmail.com" },
            { "google.com", "yahoo.com" },
            { "google.com", "facebook.com" },
            { "google.com", "youtube.com" },
            { "bing.com", "google.com" },
            { "bing.com", "gmail.com" },
            { "bing.com", "yahoo.com" },
            { "bing.com", "facebook.com" },
            { "bing.com", "youtube.com" },
            { "test.com", "google.com" },
            { "test.com", "gmail.com" },
            { "test.com", "yahoo.com" },
            { "test.com", "facebook.com" },
            { "test.com", "youtube.com" },
    };

    // Insert the pairs from `pairs`.
    for(const auto& pair : pairs)
    {
        std::string origin = pair.first;
        std::string target = pair.second;

        graph.Insert(origin, target);
    }

    // The `from` should be storing `target` under the `origin`'s adjacency list.
    for(const auto& pair : pairs)
    {
        std::string origin = pair.first;
        std::string target = pair.second;
        std::vector<std::string> list = graph.from.at(origin);

        REQUIRE(std::find(list.begin(), list.end(), target) != list.end());
    }
}
```

Test 3

```
TEST_CASE("GRAPH :: Insert :: `into` Matches Expectation", "[Insert]")
{
    Graph graph = Graph();
    std::pair<std::string, std::string> pairs[15] = {
            { "google.com", "google.com" },
            { "google.com", "gmail.com" },
            { "google.com", "yahoo.com" },
            { "google.com", "facebook.com" },
            { "google.com", "youtube.com" },
            { "bing.com", "google.com" },
            { "bing.com", "gmail.com" },
            { "bing.com", "yahoo.com" },
```

```
            { "bing.com", "facebook.com" },
            { "bing.com", "youtube.com" },
            { "test.com", "google.com" },
            { "test.com", "gmail.com" },
            { "test.com", "yahoo.com" },
            { "test.com", "facebook.com" },
            { "test.com", "youtube.com" },
    };

    // Insert the pairs from `pairs`.
    for(const auto& pair : pairs)
    {
        std::string origin = pair.first;
        std::string target = pair.second;

        graph.Insert(origin, target);
    }

    // The `into` should be storing `origin` under the `target`'s adjacency list.
    for(const auto& pair : pairs)
    {
        std::string origin = pair.first;
        std::string target = pair.second;
        std::vector<std::string> list = graph.into.at(target);

        REQUIRE(std::find(list.begin(), list.end(), origin) != list.end());
    }
}
```

Test 4

```
TEST_CASE("GRAPH :: PageRank :: PageRank Does Not Error For Power of 0", "[PageRank]")
{
    Graph graph = Graph();
    std::pair<std::string, std::string> pairs[15] = {
            {"google.com", "google.com"},
            {"google.com", "gmail.com"},
            {"google.com", "yahoo.com"},
            {"google.com", "facebook.com"},
            {"google.com", "youtube.com"},
            {"bing.com",   "google.com"},
            {"bing.com",   "gmail.com"},
            {"bing.com",   "yahoo.com"},
            {"bing.com",   "facebook.com"},
            {"bing.com",   "youtube.com"},
            {"test.com",   "google.com"},
            {"test.com",   "gmail.com"},
            {"test.com",   "yahoo.com"},
            {"test.com",   "facebook.com"},
            {"test.com",   "youtube.com"},
    };

    // Insert the pairs from `pairs`.
```

```
    for (const auto &pair: pairs) {
        std::string origin = pair.first;
        std::string target = pair.second;

        graph.Insert(origin, target);
    }

    // Calculate PageRank for power of 0.
    Graph::Page p0 = graph.PageRank(0);

    // There should only be 7 nodes.
    REQUIRE(p0.size() == 7);
}
```

Test 5

```
TEST_CASE("GRAPH :: PageRank :: PageRank Properly Returns Default", "[PageRank]")
{
    Graph graph = Graph();
    std::pair<std::string, std::string> pairs[15] = {
            { "google.com", "google.com" },
            { "google.com", "gmail.com" },
            { "google.com", "yahoo.com" },
            { "google.com", "facebook.com" },
            { "google.com", "youtube.com" },
            { "bing.com", "google.com" },
            { "bing.com", "gmail.com" },
            { "bing.com", "yahoo.com" },
            { "bing.com", "facebook.com" },
            { "bing.com", "youtube.com" },
            { "test.com", "google.com" },
            { "test.com", "gmail.com" },
            { "test.com", "yahoo.com" },
            { "test.com", "facebook.com" },
            { "test.com", "youtube.com" },
    };

    // Insert the pairs from `pairs`.
    for(const auto& pair : pairs)
    {
        std::string origin = pair.first;
        std::string target = pair.second;

        graph.Insert(origin, target);
    }

    // Calculate PageRank for power of 0 and 1.
    Graph::Page p0 = graph.PageRank(0);
    Graph::Page p1 = graph.PageRank(1);

    // There should only be 7 nodes.
    REQUIRE(p0.size() == 7);
    REQUIRE(p1.size() == 7);
```

```
    // Each node should have `1 / |V|` ranks as default.
    for(const auto& pair : p0)
    {
        REQUIRE(pair.second == 1.0 / 7.0);
    }
    for(const auto& pair : p1)
    {
        REQUIRE(pair.second == 1.0 / 7.0);
    }
}
```

Test 6

```
TEST_CASE("GRAPH :: PageRank :: Power of 2 Matches Expectation", "[PageRank]")
{
    Graph graph = Graph();
    std::pair<std::string, std::string> pairs[7] = {
            { "google.com", "gmail.com" },
            { "google.com", "maps.com" },
            { "facebook.com", "ufl.edu" },
            { "ufl.edu", "google.com" },
            { "ufl.edu", "gmail.com" },
            { "maps.com", "facebook.com" },
            { "gmail.com", "maps.com" },
    };

    // Insert the pairs from `pairs`.
    for(const auto& pair : pairs)
    {
        std::string origin = pair.first;
        std::string target = pair.second;

        graph.Insert(origin, target);
    }

    Graph::Page provided = graph.PageRank(2);
    Graph::Page expected = {
            { "facebook.com", 0.20 },
            { "gmail.com", 0.20 },
            { "google.com", 0.10 },
            { "maps.com", 0.30 },
            { "ufl.edu", 0.20 },
    };

    // Pre-comparison: Make sure both sizes are the same with 5 nodes.
    REQUIRE(provided.size() == 5);
    REQUIRE(expected.size() == 5);

    // Compare `provided` against `expected`.
    for(const auto& pair : provided)
    {
        double multiplier = std::pow(10.0, 2);
```

```
        std::string node = pair.first;
        Graph::Rank rank = std::round(pair.second * multiplier) / multiplier;

        REQUIRE(expected.find(node) != expected.end());
        REQUIRE(expected.at(node) == rank);
    }
}
```

Test 7

```
TEST_CASE("GRAPH :: Helper :: `GetSize` Matches Expectation", "[Helper]")
{
    Graph graph = Graph();
    std::pair<std::string, std::string> pairs[15] = {
            { "google.com", "google.com" },
            { "google.com", "gmail.com" },
            { "google.com", "yahoo.com" },
            { "google.com", "facebook.com" },
            { "google.com", "youtube.com" },
            { "bing.com", "google.com" },
            { "bing.com", "gmail.com" },
            { "bing.com", "yahoo.com" },
            { "bing.com", "facebook.com" },
            { "bing.com", "youtube.com" },
            { "test.com", "google.com" },
            { "test.com", "gmail.com" },
            { "test.com", "yahoo.com" },
            { "test.com", "facebook.com" },
            { "test.com", "youtube.com" },
    };

    // Insert the pairs from `pairs`.
    for(const auto& pair : pairs)
    {
        std::string origin = pair.first;
        std::string target = pair.second;

        graph.Insert(origin, target);
    }

    // Graph should have the same number of inserts for each.
    REQUIRE(graph.from.size() == graph.into.size());
    REQUIRE(graph.from.size() == graph.GetSize());
    REQUIRE(graph.into.size() == graph.GetSize());

    // Insert a malicious de-synchronization.
    graph.from[""].push_back("");

    // Graph should return the maximum size (i.e., `from.size()`).
    REQUIRE(graph.from.size() != graph.into.size());
    REQUIRE(graph.from.size() == graph.GetSize());
    REQUIRE(graph.into.size() != graph.GetSize());
```

```
    // Insert a malicious de-synchronization. (after fixing previous)
    graph.from.erase("");
    graph.into[""].push_back("");

    // Graph should return the maximum size (i.e., `into.size()`).
    REQUIRE(graph.from.size() != graph.into.size());
    REQUIRE(graph.from.size() != graph.GetSize());
    REQUIRE(graph.into.size() == graph.GetSize());
}
```

Test 8

```
TEST_CASE("GRAPH :: Helper :: `GetRank` Matches Expectation", "[Helper]")
{
    Graph graph = Graph();

    //
    // --- Without Values ---
    //

    // The size should be 0.
    REQUIRE(graph.GetSize() == 0);

    // The `GetRank` method should return 0.0 to avoid division by zero error.
    REQUIRE(graph.GetRank() == 0.0);

    //
    // --- With Values ---
    //

    std::pair<std::string, std::string> pairs[15] = {
            { "google.com", "google.com" },
            { "google.com", "gmail.com" },
            { "google.com", "yahoo.com" },
            { "google.com", "facebook.com" },
            { "google.com", "youtube.com" },
            { "bing.com", "google.com" },
            { "bing.com", "gmail.com" },
            { "bing.com", "yahoo.com" },
            { "bing.com", "facebook.com" },
            { "bing.com", "youtube.com" },
            { "test.com", "google.com" },
            { "test.com", "gmail.com" },
            { "test.com", "yahoo.com" },
            { "test.com", "facebook.com" },
            { "test.com", "youtube.com" },
    };

    // Insert the pairs from `pairs`.
    for(const auto& pair : pairs)
    {
        std::string origin = pair.first;
        std::string target = pair.second;
```

```
        graph.Insert(origin, target);
    }

    // The size should be 7.
    REQUIRE(graph.GetSize() == 7);

    // The `GetRank` method should return 1.0 / 7.0.
    REQUIRE(graph.GetRank() == 1.0 / 7.0);
}
```

Test 9

```
TEST_CASE("GRAPH :: Helper :: `GetList` Matches Expectation", "[Helper]")
{
    Graph graph = Graph();

    //
    // --- Without Values ---
    //

    // The size should be 0.
    REQUIRE(graph.GetSize() == 0);

    // The `GetRank` method should return an empty vector.
    REQUIRE(graph.GetList("", Graph::Flow::From).empty());
    REQUIRE(graph.GetList("", Graph::Flow::Into).empty());

    //
    // --- With Values ---
    //

    std::pair<std::string, std::string> pairs[15] = {
            { "google.com", "google.com" },
            { "google.com", "gmail.com" },
            { "google.com", "yahoo.com" },
            { "google.com", "facebook.com" },
            { "google.com", "youtube.com" },
            { "bing.com", "google.com" },
            { "bing.com", "gmail.com" },
            { "bing.com", "yahoo.com" },
            { "bing.com", "facebook.com" },
            { "bing.com", "youtube.com" },
            { "test.com", "google.com" },
            { "test.com", "gmail.com" },
            { "test.com", "yahoo.com" },
            { "test.com", "facebook.com" },
            { "test.com", "youtube.com" },
    };

    // Insert the pairs from `pairs`.
    for(const auto& pair : pairs)
    {
```

```
        std::string origin = pair.first;
        std::string target = pair.second;

        graph.Insert(origin, target);
    }

    // The size should be 7.
    REQUIRE(graph.GetSize() == 7);

    // There should be 5 nodes that flow *from* each of the following.
    REQUIRE(graph.GetList("google.com", Graph::Flow::From).size() == 5);
    REQUIRE(graph.GetList("bing.com", Graph::Flow::From).size() == 5);
    REQUIRE(graph.GetList("test.com", Graph::Flow::From).size() == 5);

    // There should be 3 nodes that flow *into* "google.com".
    REQUIRE(graph.GetList("google.com", Graph::Flow::Into).size() == 3);

    // There should be 0 nodes that flow *into* "bing.com" and "test.com".
    REQUIRE(graph.GetList("bing.com", Graph::Flow::Into).empty());
    REQUIRE(graph.GetList("test.com", Graph::Flow::Into).empty());
}
```