Project 4

Due December 2, 2022 at 9:00 PM

You will be working with a partner for this project. This specification is subject to change at any time for additional clarification.

Desired Outcomes

- Exposure to GoogleTest and Google Mock
- Exposure to Expat XML library
- Exposure to Open Street Map file formats
- Use of git repository
- An understanding of how to develop Makefiles that build and execute unit tests
- An understanding of delimiter-separated-value files
- An understanding of XML files
- An understanding of Dijkstra's Algorithm
- An understanding of how to integrate a third-party C library in C++

Project Description

Navigation is a critical to modern society. Once source and destination coordinates are translated to vertices, the shortest or fastest path can be calculated to route the user to their destination. The goal of your project is to write a program that will be able to parse an OpenStreetMap (OSM) file for an area and then to find shortest/fastest routes as fast as possible. Additionally, the goal is to include multiple methods of transportation (biking vs walking and bus) in your search for fastest routes. You will be building a program that can find paths, print and save them. You will be building upon the classes from project 3. Some of the capabilities are extra credit, see the extra credit section to determine what is required, and what is extra credit.

The `CBusSystemIndexer` class you will be developing will index the `CBusSystem` information provided for ease of lookup of stops and routes. It will be helpful class in developing some of the later components of the project.

```
// CBusSystemIndexer member functions
// Constructor for the Bus System Indexer
CBusSystemIndexer(std::shared_ptr<CBusSystem> bussystem);

// Destructor for the Bus System Indexer
~CBusSystemIndexer();

// Returns the number of stops in the CBusSystem being indexed
std::size_t StopCount() const noexcept;

// Returns the number of routes in the CBusSystem being indexed
std::size_t RouteCount() const noexcept;
```

```
// Returns the SStop specified by the index where the stops are sorted by
// their ID, nullptr is returned if index is greater than equal to
// StopCount()
std::shared_ptr<SStop> SortedStopByIndex(std::size_t index) const noexcept;

// Returns the SRoute specified by the index where the routes are sorted by
// their Name, nullptr is returned if index is greater than equal to
// RouteCount()
std::shared_ptr<SRoute> SortedRouteByIndex(std::size_t index) const noexcept;

// Returns the SStop associated with the specified node ID, nullptr is
// returned if no SStop associated with the node ID exists
std::shared_ptr<SStop> StopByNodeID(TNodeID id) const noexcept;

// Returns true if at least one route exists between the stops at the src and
// dest node IDs. All routes that have a route segment between the stops at
// the src and dest nodes will be placed in routes unordered set.
bool RoutesByNodeIDs(TNodeID src, TNodeID dest,
     std::unordered_set<std::shared_ptr<SRoute> > &routes) const noexcept;

// Returns true if at least one route exists between the stops at the src and
// dest node IDs.
bool RouteBetweenNodeIDs(TNodeID src, TNodeID dest) const noexcept;
```

The `CDijkstraPathRouter` class you will be developing will implement the `CPathRouter` abstract interface. The `CDijkstraPathRouter` class will find the shortest path between source and destination vertices if one exists. At the core the shortest path finding algorithm must utilize Dijkstra's Algorithm, though you may optimize where available. The vertex IDs do not have to match the node or stop IDs used by the other classes. You will need to write a test program for the `CDijkstraPathRouter`.

```
// CDijkstraPathRouter member functions
// Constructor for the Dijkstra Path Router
CDijkstraPathRouter();

// Destructor for the Dijkstra Path Router
~CDijkstraPathRouter();

// Returns the number of vertices in the path router
std::size_t VertexCount() const noexcept;

// Adds a vertex with the tag provided. The tag can be of any type.
TVertexID AddVertex(std::any tag) noexcept;

// Gets the tag of the vertex specified by id if id is in the path router.
// A std::any() is returned if id is not a valid vertex ID.
std::any GetVertexTag(TVertexID id) const noexcept;

// Adds an edge between src and dest vertices with a weight of weight. If
// bidir is set to true an additional edge between dest and src is added. If
// src or dest nodes do not exist, or if the weight is negative the AddEdge
// will return false, otherwise it returns true.
bool AddEdge(TVertexID src, TVertexID dest, double weight, bool bidir =
    false) noexcept;
```

```
// Allows the path router to do any desired precomputation up to the deadline
bool Precompute(std::chrono::steady_clock::time_point deadline) noexcept;

// Returns the path distance of the path from src to dest, and fills out path
// with vertices. If no path exists NoPathExists is returned.
double FindShortestPath(TVertexID src, TVertexID dest, std::vector<TVertexID>
    &path) noexcept;
```

The `CDijkstraTransportationPlanner` class you will be developing will implement the `CTransportationPlanner` abstract interface. You will build upon the street map, bus system, bus system indexer, and path router to implement the transportation planner. The configuration of the transportation system will be provided as a parameter to the constructor.

```
// CDijkstraTransportationPlanner member functions
// Constructor for the Dijkstra Transportation Planner
CDijkstraTransportationPlanner(std::shared_ptr<SConfiguration> config);

// Destructor for the Dijkstra Transportation Planner
~CDijkstraTransportationPlanner();

// Returns the number of nodes in the street map
std::size_t NodeCount() const noexcept override;

// Returns the street map node specified by index if index is less than the
// NodeCount(). nullptr is returned if index is greater than or equal to
// NodeCount(). The nodes are sorted by Node ID.
std::shared_ptr<CStreetMap::SNode> SortedNodeByIndex(std::size_t index) const
    noexcept override;

// Returns the distance in miles between the src and dest nodes of the
// shortest path if one exists. NoPathExists is returned if no path exists.
// The nodes of the shortest path are filled in the path parameter.
double FindShortestPath(TNodeID src, TNodeID dest, std::vector< TNodeID >
    &path) override;

// Returns the time in hours for the fastest path between the src and dest
// nodes of the if one exists. NoPathExists is returned if no path exists.
// The transportation mode and nodes of the fastest path are filled in the
// path parameter.
double FindFastestPath(TNodeID src, TNodeID dest, std::vector< TTripStep >
    &path) override;

// Returns true if the path description is created. Takes the trip steps path
// and converts it into a human readable set of steps.
bool GetPathDescription(const std::vector< TTripStep > &path, std::vector<
    std::string > &desc) const override;
```

The `CTransportationPlannerCommandLine` class you will be developing will implement the command line interface for the `transplanner` program. The `transplanner` program is a command line program that will take in commands and execute what is requested.

```
// CTransportationPlannerCommandLine member functions
// Constructor for the Transportation Planner Command Line
CTransportationPlannerCommandLine(std::shared_ptr<CDataSource> cmdsrc,
  std::shared_ptr<CDataSink> outsink,
  std::shared_ptr<CDataSink> errsink,
  std::shared_ptr<CDataFactory> results,
  std::shared_ptr<CTransportationPlanner> planner);

// Destructor for the Transportation Planner Command Line
~CTransportationPlannerCommandLine();

// Processes the commands input to the
bool ProcessCommands();
```

An example CSV and OSM file set will be provided, the files will be used in project 4. Your tests should be built with them in mind, but you shouldn't load the files as part of the tests.

The Makefile you develop needs to implement the following:

- Must create `obj` directory for object files (if doesn't exist)
- Must create `bin` directory for binary files (if doesn't exist)
- Must compile cpp files using C++17
- Must link string utils and string utils tests object files to make `teststrutils` executable
- Must link `StringDataSource` and `StringDataSourceTest` object files to make `teststrdatasource` executable
- Must link `FileDataSource`, `FileDataSink`, `FileDataFactory` and `FileDataSSTest` object files to make `testfiledatass` executable
- Must link `StringDataSink` and `StringDataSinkTest` object files to make `teststrdatasink` executable
- Must link DSV reader/writer and DSV tests object files to make `testdsv` executable
- Must link XML reader/writer and XML tests object files to make `testxml` executable
- Must link `KMLWriter` and `KMLTest` object files to make `testkml` executable
- Must link `CSVBusSystem` and `CSVBusSystem` tests object files to make `testcsvbs` executable
- Must link `OpenStreetMap` and `OpenStreetMap` tests object files to make `testosm` executable
- Must link `DijkstraPathRouter` and `DijkstraPathRouter` tests object files to make `testdpr` executable
- Must link `BusSystemIndexer` and `CSVBusSystemIndexerTest` object files to make `testcsvbsi` executable
- Must link `TransportationPlannerCommandLine` and `TPCommandLineTest` object files to make `testtpcl` executable

- Must link `CSVOSMTransportationPlannerTest`, `CSVOSMTransportationPlannerTest` and other required object files to make `testtp` executable
- Must execute the `teststrutils`, `teststrdatasource`, `teststrdatasink`, `testfiledatass`, `testdsv`, `testxml`, `testcsvbs`, `testosm`, `testdpr`, `testcsvbsi`, `testtp`, and `testcl` test executables
- Must build the `transplanner` and `speedtest` executables once all tests pass
- Must provide a `clean` that will remove the `obj` and `bin` directories

Provided Helper Classes/Tests

- `FileDataSource`, `FileDataSink`, and `FileDataFactory` provide classes for accessing files/directories as sources/sinks
- `StandardDataSource`, `StandardDataSink`, and `StandardDataErrorSink` provide classes for accessing standard I/O as sources/sinks
- `STransportationPlannerConfig` provides a basic implementation for the configuration interface with the default assumptions
- `GeographicUtils` provides static methods for calculating distance, bearings, etc. and will be necessary for calculating the shortest/fastest paths
- `KMLWriter` and `kmlout` files provide a class and program that can convert saved paths into KML files that can be used on google maps
- `CSVBusSystemIndexterTest`, `CSVOSMTransporationPlannerTest`, `FileDataSSTest`, and `TPCommandLineTest` provide google tests for much of what you need to develop

Important Assumptions

- For shortest path assume only follow direction specified in the street map, so you can't go backward along a `"oneway"`
- For fastest path, assume can walk both directions regardless of `"oneway"`, bike/bus must follow `"oneway"`. Also, you cannot bike along paths that specify `"bicycle"` as `"no"`
- Assume busses route will take shortest path (don't worry about calculating fastest path)
- Walk speed must be taken from config `WalkSpeed` function (default is 3mph)
- Bike speed must be taken from config `BikeSpeed` function (default is 8mph)
- Bus follows speed limit and `DefaultSpeedLimit` function will provide the assumed speed limit if the way does not specify it (default is 25 mph)
- You cannot take your bike on the bus, so if you take the bus you must walk to it
- When creating the path description:
    - The direction must be based upon beginning point on a street to the end point of travelling on the street, not the beginning point and the next point
    - The direction is listed as **along** if the street name is known, and **toward** if the street name is not known, but the next street is known
    - When there are multiple options for taking a bus, the bus that will go the furthest will be taken. If multiple options are available, the bus with the earliest sorted name will be taken

You can unzip the given tgz file with utilities on your local machine, or if you upload the file to the CSIF, you can unzip it with the command:
`tar -xzvf proj4given.tgz`

You **must** submit the source file(s), your Makefile, README file, and `.git` directory in a tgz archive. Do a `make clean` prior to zipping up your files so the size will be smaller. You can tar gzip a directory with the command:
`tar -zcvf archive-name.tgz directory-name`

A working example can be found on the CSIF in `/home/cjnitta/ecs34/transplanner`. Your program is expected to have the same interface as the working example. The full interface can be listed by typing help after launching the program. A program that can convert saved paths into KML files has also been provided `/home/cjnitta/ecs34/kmlout`. Directions of how to view a KML file in google maps can be found at https://youtu.be/1HqQuHeGa38.

You should avoid using existing source code as a primer that is currently available on the Internet. You **must** specify in your readme file any sources of code that you have viewed to help you complete this project. All class projects will be submitted to MOSS to determine if students have excessively collaborated. Excessive collaboration, or failure to list external code sources will result in the matter being referred to Student Judicial Affairs.

Recommended Approach

The recommended approach is as follows:

1. Create a git repository and add your project 3 and provided files.
2. Update your project 3 Makefile to meet the specified requirements. The order of the tests to be run should the `teststrutils`, `teststrdatasource`, `teststrdatasink`, `testfiledatass`, `testdsv`, `testxml`, `testcsvbs`, `testosm`, `testdpr`, `testcsvbsi`, `testtp`, and `testcl`
3. Verify that your string utils, string data source, string data sink, file data source/sink, DSV reader, DSV writer, XML reader, XML writer, CSV Bus System, OSM, KML writer, tests all compile, run and pass.
4. Create the files and skeleton functions for BusSystemIndexer.cpp, DijkstraPathRouter.cpp, DijkstraPathRouterTest.cpp, DijkstraTransportationPlanner.cpp, TransportationPlannerCommandLine.cpp and transplanner.cpp.
5. Write tests for the `CDijkstraPathRouter` class. Each test you write should fail, make sure to have sufficient coverage of the possible data input. This will complete the test files as others will be provided for you.
6. Once tests have been written that fail with the initial skeleton functions, begin writing your `CBusSystemIndexer` functions. It should be noted that since the `CBusSystemIndexer`, `CDijkstraPathRouter`, and `CTransportationPlannerCommandLine` classes do not rely on one another for testing, they could potentially be done in any order.

7. Once the `CBusSystemIndexer` class is complete, begin writing your `CDijkstraPathRouter` functions.
8. Once the `CDijkstraPathRouter` class is complete, begin writing your `CDijkstraTransportationPlanner` functions.
   NOTE: the `GetPathDescription` function is extra credit!
9. Once the `CDijkstraTransportationPlanner` class is complete, begin writing your `CTransportationPlannerCommandLine` functions.
10. Once the `CTransportationPlannerCommandLine` and `CDijkstraTransportationPlanner` classes are complete, begin writing the `transplanner` program. There will be requirement to parse the command arguments but setting up of the config to create the `CTransportationPlannerCommandLine` is mainly what is needed.

Grading

The point breakdown can be seen in the table below. Make sure your code compiles on the CSIF as that is where it is expected to run. You will make an interactive grading appointment to have your assignment graded. You must have a working webcam for the interactive grading appointment. Project submissions received 24hr prior to the due date/time will received 10% extra credit. The extra credit bonus will drop off at a rate of 0.5% per hour after that, with no additional credit being received for submissions within 4hr of the due date/time.

Points	Description
10	Has git repository with appropriate number of commits
5	Has Makefile and submission compiles
5	Makefile meets specified requirements
5	Has DijkstraPathRouter google tests that fail with initial skeleton functions
5	Student DijkstraPathRouter google tests have reasonable coverage
5	Google tests detect errors in instructor buggy code
10	BusSystemIndexer functions pass all tests
10	DijkstraPathRouter pass all student tests
5	DijkstraPathRouter pass all instructor tests
10	DijkstraTransportationPlanner pass all tests (except path description)
5	TransportationPlannerCommandLine pass all tests
10	transplanner fully functional (except print)
5	speedtest fully functional and performs at least at baseline speed
10	Student understands all code they have provided

Extra Credit

There are two opportunities for extra credit on Project 4: Path Description and Speed Performance.

(Up to 20% extra) Path Description
Successfully implement the `GetPathDescription` function of the `CDijkstraTransportationPlanner` class. Must pass all tests, and `transplanner` must print the path as the provided version does for the same input.

(Up to 20% extra) Speed Performance
The speed test will test the speed of your `CDijkstraTransportationPlanner` against the baseline code. The idea is that your program would be part of a server that would be rebooted daily and then would handle as many queries as possible. The more queries your program can handle in a day, the fewer servers that would need to be in operation to handle the daily load. Your program will have a maximum of 30s to load the data and do any precomputation necessary to start handling the requests.

A `speedtest` program has been provide for the baseline (`speedtest_baseline`) and the optimized version (`speedtest_optimized`). The program will output the number of queries that could be completed in 24hr, it will also output a brief of the path distances/times. A `speedtest.cpp` source file that will calculate the number of queries has been provided. Do not modify the `speedtest.cpp` when constructing your `speedtest` program. A verbose listing of the paths can be created with the `--verbose` option. Speed comparisons will be done with compiler optimizations **disabled**. If your implementation outperforms the baseline `speedtest_baseline`, some extra credit will be available. The goal for full extra credit is to outperform the `speedtest_optimized`.

Helpful Hints

- Read through the guides that are provided on Canvas
- See http://www.cplusplus.com/reference/, it is a good reference for C++ built in functions and classes
- Use lenth(), substr(), etc. from the string class whenever possible.
- If the build fails, there will likely be errors, scroll back up to the first error and start from there.
- You may find the following line helpful for debugging your code:
  `std::cout<<__FILE__<<" @ line: "<<__LINE__<<std::endl;`
  It will output the line string "FILE `@ line: X`" where FILE is the source filename and X is the line number the code is on. You can copy and paste it in multiple places and it will output that particular line number when it is on it.
- Make sure to use a tab and not spaces with the Makefile commands for the target
- `make` will not warn about undefined variables by default, you may find the `--warn-undefined-variables` option very helpful

- The debug option for make can clarify which targets need to be built, and which are not. The basic debugging can be turned on with the `--debug=b` option. All debugging can be turned on with the `--debug=a` option.
- Make sure to use a `.gitignore` file to ignore your object files, and output binaries.
- Do not wait to the end to merge with you partner. You should merge your work together on a somewhat regular basis (or better yet pair program).
- Use `CStringDataSource` and `CStringDataSink` to test your reader and writer classes for DSV and XML.
- You will probably want to use static functions in your classes for the callbacks to the library calls that require callbacks. The call data (`void *`) parameter that the functions take and the callbacks pass back as a parameter, should be `this` from your object.
- You may find https://www.xml.com/pub/1999/09/expat/index.html helpful for describing the libexpat functions. You are not going to need to use every function in the Expat library.
- The OSM XML file format is described https://wiki.openstreetmap.org/wiki/OSM_XML. Though not necessarily important for this project, the tag features are described are https://wiki.openstreetmap.org/wiki/Map_features.
- Use the `--gtest_output=xml:filename` to create test results files that can be used to prevent rerunning tests that have already succeeded.
- Use the `--gtest_filter=-Test.Subtest1:Test.Subtest2` to skip google tests that won't pass. This can be helpful for skipping say the `CSVOSMTransporationPlanner.PathDescription` test that is extra credit.