Functional specifications

Overview

Product Description

This software solution is a high-performance software system that provides a **REST API**[1], developed in **C++**[2], designed to compute the quickest path between two landmarks in the United States of America. The API is intended to handle a dataset with approximately 24 million nodes[3], ensuring results are within 1 second of the response time for an average laptop.

The key features of the solution include:

- Pathfinding using efficient algorithms
- Data validation
- Support for the user-selectable response formats in **XML**[4] and **JSON**[5]
- Error handling by standard REST API practices

Product Functional Capabilities

The primary functions of the software are:

1. **Pathfinding**: We must be able to find the quickest path between two landmarks based on a provided dataset (a CSV file of American landmarks in this case).

2. **Data validation**: A separate utility will verify the integrity of the provided dataset, ensuring graph[6] connectivity and the absence of loops.

3. **Response format selection**: The system supports user-selectable response formats, offering both XML and JSON outputs.

4. **Error handling**: The API will return appropriate status codes and error messages based on standard REST API error handling practices. Here is a table of the error codes and their corresponding messages:

Error Code	Error Message	Example Scenario
400	Invalid ID provided	Invalid source landmark ID
404	Landmark not found	Landmark ID not in dataset
500	Internal server error	Dataset not loaded or corrupted
503	Service unavailable	Too many requests

5. **Performance**: Ensure response times of under 1 second for an average laptop.

User Roles

Our current vision is for a single-user role, the API user.

The API user will interact with the software through HTTP requests, and will be responsible for:

- Providing the API with the necessary input data
- Interpreting the API's response

- Handling any errors that may occur during the process

Use Cases for All Operations

Use Case 1: Compute the shortest path

- **Primary Actor**: API User
- **Description**: The user provides the ID of the starting and ending landmarks. The system calculates the quickest path and returns the result in the user-selected format.
- **Preconditions**: The USA roads' dataset must be loaded and the local server must be running.
- **Postconditions**: The system returns the quickest path between the two landmarks in **XML** or **JSON** format.
- **Normal Flow**:
  1. The user sends an **HTTP**[7] request to the API with the source and destination IDs.
  2. The server processes the request and computes the quickest path.
  3. The server returns the result, including the travel time and ordered list of landmarks.

Use Case 2: Handle invalid input IDs

- **Primary Actor**: API User
- **Description**: The user provides an invalid ID as input. The system returns an error message.
- **Preconditions**: The USA roads' dataset must be loaded and the local server must be running.
- **Postconditions**: The system returns an error message.
- **Normal Flow**:
  1. The user sends an HTTP request to the API with an invalid ID.
  2. The server processes the request and returns an error message.

General Constraints

- The software must be able to handle a dataset with approximately 24 million nodes, as it represents the number of landmarks in the United States of America.
- The software must be able to return results within 1 second of the response time for an average laptop.
- For this project, only standard libraries (such as STL) and a minimal web server framework should be used for the implementation.
- We don't need a Graphical User Interface (GUI), interaction will be done through HTTP requests with the API.

Assumptions

Here's the list of assumptions made for this project:

1. The dataset is provided in a **CSV**[8] file format, ensuring a code stability.
2. The users will interact with the system only using standard REST API clients.
3. The performance benchmarks are based on an average laptop, similar to those provided by the school.

Related Software

The REST API will be developed in C++ and will use a minimal web server framework to handle HTTP requests. The API will be able to handle both XML and JSON response formats. The REST API will operate independently and will not directly interface with other software systems, except for common API testing tools (e.g., Postman, cURL).

Specific Function Descriptions

Description

The functions are hypothetical examples and should be replaced with the actual functions that will be implemented.

```
// Function to compute the shortest path between two landmarks
std::string computeShortestPath(int sourceID, int destinationID);

// Function to validate the dataset
bool validateDataset(std::string datasetPath);

// Function to select the response format
std::string selectResponseFormat(std::string format);

// Function to handle errors
std::string handleError(int errorCode);

// Function to ensure response times
bool checkResponseTime(double responseTime);

// Function to load the dataset
bool loadDataset(std::string datasetPath);

// Function to handle HTTP requests
std::string handleRequest(std::string request);
```

Inputs

The system accepts the following inputs via a GET request:

1. HTTP Request

Users will interact with the API by sending a GET request in the following format:

```
curl "http://localhost:18080/quickestpath?source={sourceID}&destination={destinationID}(&format=xml)"
```

- **Parameters**:
  - `source`: The ID of the source landmark.
  - `destination`: The ID of the destination landmark.
  - `format`: The response format (XML or JSON)
    - Optional parameter. If not provided, the response will be in JSON format.

2. Example Request

A typical request might look like this:

```
curl "http://localhost:18080/quickestpath?source=1&destination=10&format=xml"
```

3. Input Rules

- **Source and Destination IDs**:

    - The source and destination IDs must be valid integers corresponding to landmarks in the dataset.
    - The source and destination IDs must be different otherwise the system will return this output:

```
{
  "time": 0,
  "node_path": []
}
```

```
   <result>
    <time>0</time>
    <node_path></node_path>
   </result>
```

- **Response Format**:

    - By default, the response format is JSON.
    - The user can select the response format by providing the `format` parameter in the request.

- **Notes on Dataset**:

    - The dataset must be pre-loaded into the system before processing requests, ensuring the system can handle the request in less than 1 second, as required.

Processing

The system will process the inputs as follows:

1. **Request checking:** The system will check the validity of the HTTP request, if the request is valid (i.e., contains the required parameters), it will proceed to the next step.
2. **Pathfinding:** The system will compute the shortest path between the source and destination landmarks based on the provided dataset.
3. **Formatting:** The system will format the result in the user-selected format (XML or JSON).
4. **Result:** The system will return the result to the user.

Outputs

The system will output the following:

1. **Successful Response**:

    1. **Travel time**: The sum of the travel times between the landmarks.

    2. **Node path**: The ordered list of landmarks to reach the destination. Example:

    ```
    {
      "travel_time": 120,
      "node_path": [1, 5, 7, 10]
    }
    ```

```
    <result>
      <travel_time>120</travel_time>
      <node_path>
        <node>1</node>
        <node>5</node>
        <node>7</node>
        <node>10</node>
      </node_path>
```

2. **Error Response**:

    1. **Error code**: The error code returned by the system.
    2. **Error message**: The error message corresponding to the error code. Example:

```
{
  "error_code": 400,
  "error_message": "Invalid ID provided"
}
```

```
    <error>
      <error_code>400</error_code>
      <error_message>Invalid ID provided</error_message>
    </error>
```

External Interfaces

User Interfaces

There is no graphical user interface (GUI). Users will interact with the system through HTTP requests via command-line tools or API clients.

Hardware Interfaces

The software is designed to run on typical laptops with specifications similar to those provided by the school.

Software Interfaces

- The system will expose an REST API endpoint to compute the shortest path.
- Common API testing tools (e.g., Postman, cURL) can be used to interact with the API.

Communication Interfaces

- Communication with the system will only be done through HTTP requests.
- The API will support requests on `localhost` during and after development.

Performance

- The system is expected to return results within 1 second of the response time for an average laptop.
- Performance benchmarks will be based on the specifications provided by the school.

- It should support datasets with up to 24 million nodes without significant performance degradation.

Design Constraints

1. The software must be implemented in C++.
2. Only standard libraries (such as STL) and minimal frameworks required for the REST API are permitted.
3. No standard libraries or frameworks that are not part of the standard C++ library are allowed to be used.

Attributes

Security

Regarding security, the following measures will be taken:

- The system will not store any user data.
- The system will not have any user authentication or authorization mechanisms.
- The system will not have any user sessions or cookies.
- The system will not have any user input validation beyond the basic input validation required for the API to function correctly.
- The system will not have any user data encryption or decryption mechanisms.
- The system will not have any user data storage mechanisms.
- The system will not have any user data transmission encryption mechanisms.

Reliability, Availability, Maintainability

- The system will be designed to be reliable by ensuring that it can handle a large dataset and return results within 1 second of response time.
- The system will be designed to be available by ensuring that it can handle multiple requests concurrently.
- The system will be designed to be maintainable by using standard C++ libraries and minimal frameworks, making it easier to maintain and update in the future.

Configurability and Compatibility

- The system will support user-selectable response formats (XML and JSON).

Installability

- The system will be installed on the user's local machine.
- Clear installation instructions will be provided to the user by the user manual to ensure a smooth installation process.

Usability

- The API should handle invalid inputs gracefully and avoid crashes.

Additional Requirements

User documentation

The user documentation will include:

- Installation instructions

- API usage instructions
- Error handling instructions
- Response format selection instructions
- Simple examples of API usage
- QnA section

Other requirements

To ensure the success of the project, the solution should include a testing suite that covers:

- Unit tests for individual functions
- Integration tests for the API endpoints
- Performance tests to ensure response times are within the specified limits
- Error handling tests to verify that the system handles invalid inputs correctly

Development and Deployment

Development Environment

The recommended development environment includes:

- **C++ compiler:** GCC 10+ or Clang 11+
- **Web server framework:** Crowcpp or Restbed
- **IDE:** Visual Studio Code or CLion
- **Testing framework:** GitHub Actions

Deployment Environment

The deployment environment includes:

1. Clone the repository and build the project using the provided CMake configuration.
2. Configure the webserver to run the API on port 8080.
3. Ensure the dataset is loaded at the specified path during startup.
4. Use a reverse proxy like Nginx for production deployments to handle HTTPS.

Glossary

---

1. REST API - Representational State Transfer Application Programming Interface Source (https://en.wikipedia.org/wiki/Representational_state_transfer)↩

2. C++ - A general-purpose programming language created by Bjarne Stroustrup as an extension of the C programming language, or "C with Classes". Source (https://en.wikipedia.org/wiki/C%2B%2B)↩

3. Node - A Node, often referred to as a Vertex, is a fundamental unit of a graph. It represents an entity within the graph. In our case, a node represents a landmark in the United States of America. Source (https://www.geeksforgeeks.org/graph-terminology-in-data-structure/)↩

4. XML - Extensible Markup Language Source (https://en.wikipedia.org/wiki/XML)↩

5. JSON - JavaScript Object Notation Source (https://en.wikipedia.org/wiki/JSON)↩

6. Graph - A graph is a data structure that consists of a finite set of nodes (vertices) and a set of edges that connect the nodes. In our case, the nodes represent landmarks, and the edges represent the roads connecting the landmarks. Source (https://www.geeksforgeeks.org/graph-terminology-in-data-structure/)↩

7. HTTP - Hypertext Transfer Protocol Source (https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol)↩

8. CSV - Comma-Separated Values Source (https://en.wikipedia.org/wiki/Comma-separated_values)↩