Technical Documentation

# Program Flow and User Interface

|  |  |
| --- | --- |
|  | The client opens with a login interface. The user can enter a username and password. The password will be hidden by the dot character.  The user can also register a new account with the server by using the Register link. |
|  | Clicking the register link will open a new form which allows the user to enter new login credentials. There are several checks on the username and password to confirm they are suitable before the user will be allowed to register (See “Logging In and Registering” for details). |
|  | After registering an account, the register form will automatically close and the login form will refocus. Pressing Login will initiate the verification process (See “Logging In and Registering” for details). |
|  | This is the main window of the system. The window has several features. The selection boxes and number selectors on the left allow the user to select the parameters of the maze. Filling in all boxes whilst being connected to the server will unlock the Request Maze button which will send the information to the server where a maze will be generated (See “Maze Generation” for more detail.) Alternatively, by pressing Get Mazes, the selection box below the button will be populated with mazes the user has saved to the server. The user can load or delete the maze from the server with the relevant buttons. The panel on the right hosts dynamically generated charts showing local and global statistics for best times and mazes generated. |
|  | Here is the main form populated with data. The Request Maze button is now unlocked since each parameter has been filled in. Clicking the button will send the parameters to the server and a maze will be passed back to the client, being displayed in a new form. |
|  | Here is the Display form. The user can use WASD controls or the onscreen buttons to move the blue square through the maze. If they do this, a timer will start and be displayed, stopping when they reach the end of the maze. The Close button will then be unlocked, allowing access to the parameter form. The user can also use the selection box to select the algorithm they would like to request a server solve with. Pressing request solve will send the maze to the server which will send a solution back to the client before displaying the solution. The user also has the option to name their maze for a server save, or save it locally, which will open a File Explorer window where they can choose where to save the formatted image. |
|  | Here is the display form after requesting a solve using the Maze Routing algorithm. The solution is displayed with a purple line. The user can no longer attempt a manual solve. The close button has been enabled. Closing the window will refocus the parameter window and allow the user to begin the process again. |
|  | In the background, the server has been open the entire time. It logs all requests made to it in the console window. The client has exception handling on all server requests, since if the server closes unpredictably, there will be an unavoidable error which needs to be appropriately handled. (See “Network Exception Handling” for more details). |

# Project Structure

|  |  |
| --- | --- |
| Client | Server |
|  |  |
| The client’s code is mainly held within its form codebehind files. The Maze and Maze-derivative files it has contain only properties and [JsonConstructor] tagged constructors since they only need to build objects from the property structure to be read and displayed. This keeps the client lightweight. The client has a copy of every protocol file so it can send and receive the appropriate objects across the network. The client also makes use of a static class Globals (See “Global Variables” for more details.) | The server has many features in its extensive solution. It contains a folder of every protocol file so it can communicate correctly with clients. Each of these protocols have a dedicated service script in the Services folder, defining the management of the data in these requests, and what is sent back to the client. It also contains both the launchSettings.json and the appSettings.json files. These define how the server behaves (such as what port it operates across, whether it logs messages, what network protocol it uses, etc). It also contains the full definitions of all Maze classes and has the Solver and Solver-derivative classes, unlike the client. The server does not make use of global variables. |

# Techniques Used

## General Programming Techniques

## Network Exception Handling





Here is an example of how I have managed unpredictable server errors. When we make a network request, we can attach a deadline to it by providing a time: I have provided the current UCT time + 3 seconds for all deadlines, essentially forcing an exception after 3 seconds of server inactivity. Since we are turning an unpredictable server error into a predictable and specific error, we can catch that specific error by checking for the DeadlineExceeded status code, which is thrown when the deadline for a server request is exceeded. In the catch block, I have a procedure which is called whenever there is a server error that outputs an appropriate error message to a label on the form, as well as locking up buttons that interact with the server until a connection is re-established.

## Multithreading Exceptions

The code in my forms has many async and multithreaded methods. In a multithreaded form, threads cannot access objects from other threads without invoking a subroutine that interacts with it by proxy. However, the thread does not know when an object has been disposed, such as when a form closes, so an unpredictable error is thrown when trying to invoke access to a disposed object. Since this error causes no issue if it is ignored, we can stop a client crash if we specifically catch the ObjectDisposedException without handling it.

## Global Variables

See [PAGE] for the Globals.cs code.

Since excessive use of global variables makes code less robust, I have limited my usage to constant variables only. Furthermore, all global variables are prefixed with “g\_” to differentiate them from local variables. The only 2 non-constant globals are the username and userID, since these cannot be defined in the code as the system has many users. These are updated to the correct values at login and only read from afterwards. The version string is used to update the name of each form with the current client version. It is purely aesthetic but could have use in managing client updates. The cellWidth and cellHeight variables store the pixel width and height of cells in the maze, so the dynamically sized display form can autosize correctly. The keysize and iterations variables are used in hashing passwords (see “Logging In and Registering” for more details).

# Server Protocols

## Proto Files

See [PAGE] for the protocol files.

Protocol files end in the extension .proto and have their own language and syntax to define a service which operates across the network and the related request and response objects the service uses. The files have a set structure: first, the syntax is set. All my protocols are written in the proto3 syntax, so the top line of every file will set this. Next, the namespace the protocol will be implemented in is set. My project uses the Server namespace and Client namespace, so all my protocols are passed the Server namespace. Next, the service is defined. The service can contain many operations, represented by the rpc keyword. The operation is made up of the method name, the request object name, and the reply object name. Once all the operations have been defined in the service, the request and reply objects must be defined with what data they carry. This is done using the message keyword followed by the object name. Within the braces each variable is defined using a simple data type and an identifier. The number assigned to them is the order in which they must be passed to the collection when creating a request. Some of my protocols do not need to return a reply message to the client since they happen discreetly, such as incrementing stat values. In this case, you can import the google/protobuf/empty.proto, which when set as the reply object in the rpc definition, allows you to create a void network service. Once these protocols are defined, they have to be integrated into the project to allow the service code to be written (See “Project File References” for more details).

## Project File References

See [PAGE] for the client and server project files.

Since protocol files generate obfuscated code to facilitate the low-level transfer of data across the internet, they need to be referenced in the project file, so the compiler knows to create these files when the project is built. This is done by including the path to the file in a protobuf element within an itemgroup in the project file.

# Algorithms

## Database Management

## Maze Generation

### Recursive Backtrack Algorithm

### Growing Tree Algorithm

### Wilson’s Algorithm

## Maze Solving

### Depth-First Search

### Maze-Routing Algorithm

## Logging in and Registering

## Client-Server Interactions

## Other Algorithms