1 PROJECT: DEVELOPMENT OF QGIS TOOLS

1.1 Problem Statement

As can be demonstrated by the software analysis of the previous section, there are some weaknesses in the currently available software which may be applied to utility asset management, as much of the software is dated. New solutions are in the process of being developed and released by several industry providers which aim to largely modernize GIS-based utility asset management. One area which some clients may be interested in is how solutions may be combined with open source software. In recent years, the capabilities of offerings such as QGIS have improved and being able to combine asset management solutions with GIS software such as these will allow Merkator to further diversify their solution offerings. Some advantages to using an open source GIS for asset management include a reduction of potential administrative issues relating to licensing, participating in open-source communities, customization, as well as cost effectiveness.

Merkator is currently developing an asset management solution, “Marlin”, to satisfy modern user requirements. Whilst Marlin can run within an ESRI environment, it had not yet been tested in QGIS. Thus, the purpose of this project within the context of the internship was to test the capabilities of QGIS to interface with Marlin, as well as to assess general capabilities of QGIS with regards to implementing basic features of a utility editing network. A difficulty in testing Marlin on QGIS is that QGIS does not natively support .NET plugins, requiring intermediary code be set up through the supported languages of either C++ or Python.

In addition to being able to handle a basic utility network, it is also important that users can interact with this network and perform analysis. Typical analysis using UAM software may include address management, lifecycle management, and utility isolation tracing (also referred to as tracing).

Tracing refers to being able to find an end node in a network and sending a trace through the network which can return the origin point of the utility being supplied to that node, as well as the path taken to reach the node (Barnes and Harary, 1983). It is useful for several applications, such as tracing outage management from a node which is reported to be undergoing an outage; planning for maintenance by assessing affected nodes; or ensuring optimal network design.

This project can thus be divided into three connected sub-projects. The first is to develop an extension using Python that allows QGIS (developed using Python and C++) to interface with Marlin, which is developed in C# .NET. The second is to develop an extension which allows for users to design a utility network, as well as run basic tasks such as traces through the system. The final part is to test the capability of QGIS in running more advanced utility network tasks, in this case being a trace.

1.2 Materials and Methods

1.2.1 QGIS/.NET Interface

To create a framework with which to port Marlin functionality to QGIS, the first step (and the project undertaken during this internship), was to create an interface for the two softwares to connect to each other. A Python plugin was developed to do this.

To complete this task, data and tools were provided by Merkator. The data was a basic test utility network, showing several features connected in a simple manner. In addition to this, a simple Python Class (MarlinControllers) was provided by Merkator which allows a graphical window developed in .NET to be displayed in QGIS. This involved importing the common language runtime (clr) for Python and adding the location of the .NET DLL as a reference file.

Commented [A1]: ?

From here, the internship task was to extend this code so that, in addition to projecting the winform into QGIS, the winform was able to, through signals implemented with the pyQGIS Python API, display information regarding the current state of the utility network based on user interaction with the QGIS user interface. Information that may be relevant to a user working on utility networks may include the IDs of features which are added, changed or deleted; as well as any similar alterations to these feature attributes or geometry. Vice versa, upon user interaction with the .NET winform, any edits made using this should be received and processed in QGIS.

Additionally, as UAM data models are restricted in terms of how they may be edited due to various logical rules to ensure network integrity, these listeners were also designed to reject changes that may break utility network rules. For example, while editing a utility network, a user may choose to edit attributes (in the case of utility networks, this might be type, operating status, materials, and others). Before being able to commit these changes, network rules will typically check to see if such a change conforms to rules of the data model.

Additional functions were developed to check for validity of attributes (defined in this case by arbitrary rules). Functions were made to listen for any changes, and after doing checks such as these, return to the user if the change was made and to what the change was made; or if it was unsuccessful, which features were unsuccessfully changed and why.

1.2.2 QGIS Utility Network Extension

To contrast with the above methodology, which aims to test whether a utility network in QGIS can interact with an external software solution, this method aims to provide the tools with which a new utility network could be designed in QGIS. To do so, a basic utility network editing extension was built which allows for the creation of edge (representing pipes/electricity lines/etc.) and point (representing valves/junctions/supply, etc.) features and defines the rules which allow for these features to be added to, altered, or removed from the database. For the basic implementation of this extension, rules were defined which define the basic behavioral rules of the network.

1) Edge features must always have a start point and an end point. These points correspond to features in the point layer and can be used to model the connectivity of features across the network. Edge features which are not placed within a valid distance from point features are automatically removed.
2) If a new point is added which intersects an edge feature, the edge should be split in two, with the new point acting as end/start point to these two new edges respectively.
3) Point features must have a type describing their role in the utility network.

Commented [A2]: In ESRI/ArcGIS-terminology this would be called ‘behaviour’

The flowchart in Figure 4 shows how the extension was conceptualized, with these requirements in mind. Functions in pyQGIS were developed in order to realize each of these steps in the workflow.

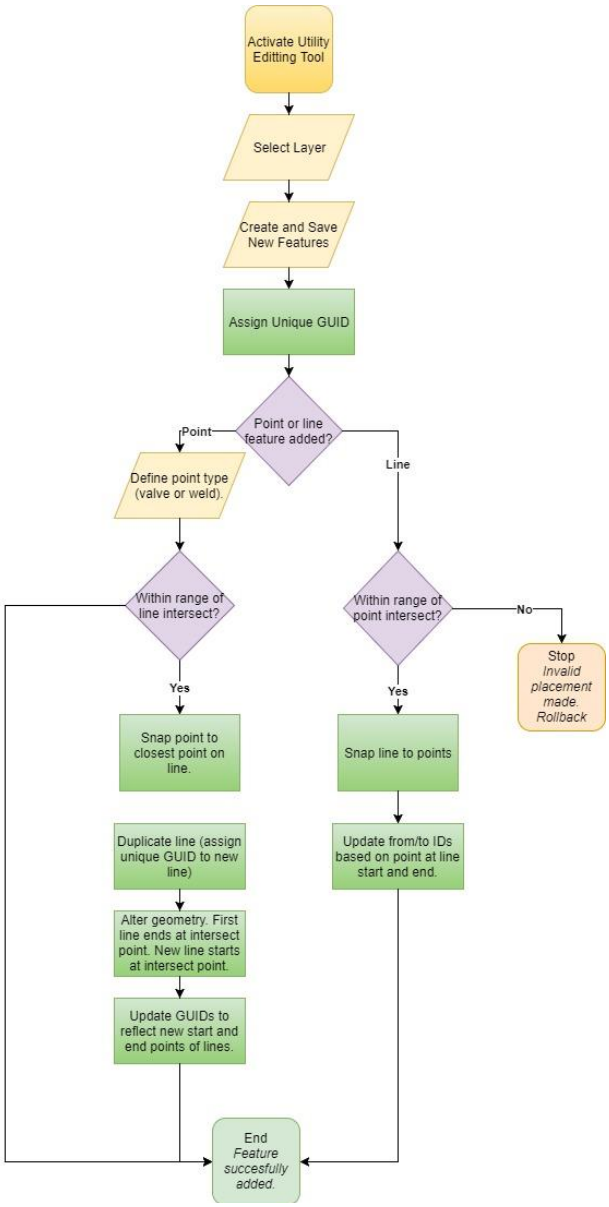

Figure 1: Flowchart for utility network editing extension.

1.2.3 QGIS Utility Trace Extension

To perform tracing in QGIS utility networks, an additional Python tool has been developed which allows for the finding of supply points in a flow system, based on a selected end or mid-point. This tool could be extended to further allow for other analysis to be traced, such as from supply points to end points (e.g. for calculating nodes which may be affected from a supply outage), or for bidirectional tracing. The flowchart in Figure 5 shows the planned workflow that the tool was developed to complete.

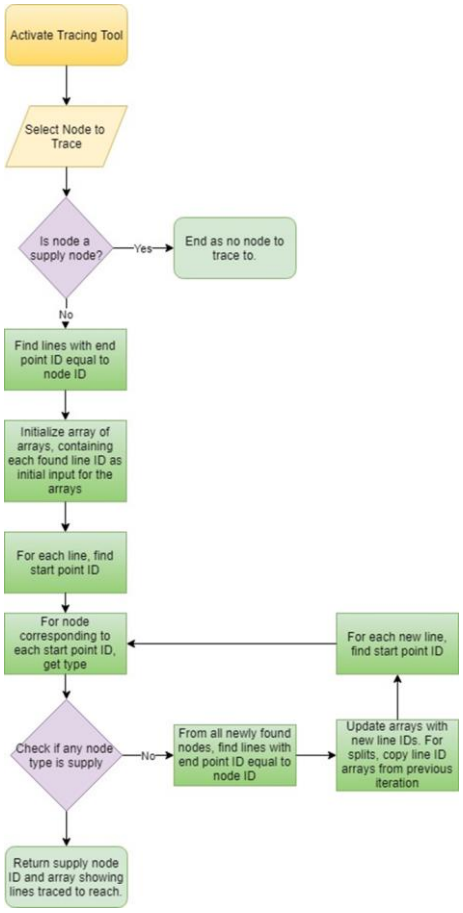

Figure 2: Flowchart for Utility Network Tracing Tool

1.3 Results & Discussion

1.3.1 QGIS/.NET Interface

To enable communication between QGIS and .NET, fifteen functions were developed using Python and the pyQGIS API, by extending the script which already is used to project a .NET winform in QGIS, with these functions running depending on the actions a user took in the QGIS UI. A summary of each function, and how they facilitate the interfacing of the two softwares is described below.

selectNow: Python listener. Will run when the user makes a new selection on the map. It will create a list of the selected features, and will send this list to the .NET winform, where it is also displayed. If the user begins editing, it will send signals to run the editing_started function.

editing_started: Will create listeners for a variety of user actions which may occur in an edit session. These various functions include for when attributes are changed, features are added, or geometries are changed; as well as when these changes (as well as deletions) are committed.

attribute_change /feature_add/geometry_change_validation: The validation of added or changed features checks that attributes and geometries occur in allowed ranges. Each of these functions will call additional functions “is_valid_geom” or “is_valid_attribute”. At present however, the check for valid geometry is simply passed, while the attribute check only checks that the new value is not null (see section 5.3.2 for a more detailed example of how geometry may be checked using pyQGIS).

features_deleted/features_added/attributes_added/attributes_deleted/attributes_valuechanged/geometries_changed_commit: The commit functions occur once a change is committed to the edit. They each make a list of feature changes, containing the feature ID, the initial value, and the changed value. This list is then sent to and displayed in the .NET winform. The purpose of these functions is to provide a means of logging any changes which are recorded in a UAM system, which is important in the ongoing maintenance of a utility network. This is done at commit instead of at the feature add or change validation step, so that only validated changes are recorded. In the above case, where validation fails, the script will implement a rollback function, cancelling all unvalidated functions.

The workflow between these functions allows for robust connectivity between QGIS and .NET, with this functionality being conceivably portable to the Marlin solution by Merkator. In addition to this, it is important that any workflows implemented on the .NET software side are visualized in QGIS. The base winform was extended to also provide user options to start editing, as well as to alter attributes in tables. This is possible in the same Python extension as was used for the previous functions, however in this case the Python signal listeners are developed to listen to button clicks from .NET, which are pushed through via the clr interface.

Further testing of this extension would involve replacing the basic winform in .NET with actual code from the Marlin solution, to ensure that workflows in this software are also able to be recorded through Python listeners and adequately visualized in QGIS. This project suggests that should be the case, as general connectivity has been established in both directions by using Python signal listeners. Additionally, there remain some bugs which need to be solved. For example, when a user selects a different layer whilst the extension is running, Python errors will occur if the new layer is of a different geometry type (unless the extension is reset). A debugging procedure will further strengthen the connection between the QGIS and .NET software. The code for this extension is included in Appendix B.

1.3.2 QGIS Utility Network Extension

The extension for the utility network editing implements several logic rules which are common within UAM. To utilize, the user must first specify a point layer and an edge layer in QGIS. From this point, it is possible to create a network from scratch by manually drawing the nodes or edges of a network. The script employs listeners for when selections or changes are made in QGIS, and firstly recognizes if the selected layer is of type point or line. Following this, any change to these layers (such as additions or geometry alterations) will be sent through a validation procedure, as well as a correction of geometry. In the case of points, this involves checking if a point is within a certain range of a pipe, in which case it is interpreted to intersect the pipe; whilst for lines it checks if the end points are within certain ranges of a node. Should either not be within range, in the case of points the point geometry will be left as is and it will be given a unique ID as a new point; while for lines, any new line not placed between two nodes is viewed as invalid and not added to the map. The script then employs a snapping function, which has differing behavior depending on whether a line or point feature is being altered. For point geometry, any line which is within a sufficient distance to be intersected will be divided into two where the point intersects, with the point

geometry being altered to be placed at the intersection point. For lines, the two end point geometries will be altered to begin and end at the two closest node points.

Combined with the above findings that QGIS can interface with .NET code, it can be surmised that it is possible to port Marlin into QGIS, and that the latter software has strong capabilities in terms of designing, viewing, analyzing and updating utility networks. Much as in the above case, some bugs remain, and these again relate to switching of selection layers. A solution to this problem, commonly applied to both this and the above extension, could greatly improve the utility of both. The code for this extension is included in Appendix C.

1.3.3 QGIS Utility Trace Extension

For the utility trace extension, a breadth first search was implemented. This is a means of searching through graph networks by looking at each node connected to the start node, and then each node connecting to those nodes; such that every node inspected in an iteration will be at the same distance from the start node (note that in this case, distance was defined in terms of number of nodes, rather than length of pipes). The code has been developed in line with the flowchart shown in Figure 5, such that nodes are inspected in an array of arrays, allowing for branching across a network.

This code could be extended, as there are some problems with the tracing method developed. Firstly, it only returns the closest supply point. UAM typically requires that all supply points and the paths to them be returned on a trace. This can have applications for if there is an outage on one path, or to calculate costs for different suppliers, amongst other things. The current code could be altered to accommodate this by not stopping the search when a supply node is found, but instead continuing it until there are no more supply nodes. The code for this extension is included in Appendix D.