Development of an Interactive Web-Based Platform for Energy Network Trace Analysis and Visualization

Álvaro Lastra Aragoneses
MEng Telecommunications & MSc Smart Grids
Universidad Pontificia Comillas and University of Strathclyde
Glasgow, UK

Supervisor: Bruce Stephen
Department of Electronic and Electrical Engineering
University of Strathclyde
Glasgow, UK

***Abstract*—This thesis presents the development of the NAVI Trace Toolkit, a web-based platform designed to modernise trace analysis workflows within energy distribution networks. Trace analysis is a powerful technique to visualise and diagnose the structure and behaviour of electrical grids. However, the lack of a dedicated trace development environment has limited its implementation at Scottish Power Energy Networks (SPEN). Existing trace development workflows rely on executing manual Python scripts, which lack real-time feedback, flexibility, and accessibility for trace developers.**

The NAVI Trace Toolkit integrates energy network analysis with modern web software development and a user-centred design approach to address these limitations. Built using Python, Flask, HTML5, CSS3, and JavaScript, the platform provides an interactive interface that supports real-time trace execution, dynamic styling, and modular architecture. It allows the developers to load multiple subnetworks, customise visual outputs, and develop trace logic using built-in functions within a responsive and intuitive environment.

The theoretical foundation of the toolkit is based on Hoel et al.'s trace framework, adapted to energy networks with rich attribute metadata. This project contributes to the trace framework by integrating a new function for identifying critical paths using a custom weight function.

The Critical Installation Path (CIP) use case demonstrates a DFS-based and a scalable Dijkstra-based approach, balancing analytical depth with computational feasibility. This realistic use case validates the platform's capabilities and illustrates a practical application of the critical path function.

The application runs locally, requires no administrative privileges, and reduces setup time to under five minutes. A user experience evaluation assessed the platform's efficiency, achieving 1.64 tasks per minute, and effectiveness, reaching 100% task completion. User survey feedback from participants reflected high customer satisfaction.

***Index Terms*—trace analysis, energy networks, smart grids, geospatial visualization, software, user experience**

I. Introduction

Scottish Power Energy Networks (SPEN) utilises the NAVI platform to visualise the energy network and provide engineers with insights. Among its various capabilities, one of the most powerful is trace analysis—an analytical tool used to understand, visualise, and diagnose the structure and behaviour of the energy network. This involves highlighting nodes and lines on the network topology to visually represent data, such as tracing upstream from a given impedance to the source. This process helps to determine the network's path, components and electrical characteristics.

Trace Analysis can be extended to various energy network applications; however, SPEN lacks a dedicated trace development platform. The current workflow for trace developers relies on executing basic Python scripts locally, which is time-consuming and does not have real-time feedback. This particularly frustrates users when they want to make minor changes and iterate quickly to refine the code, hindering productivity and experimentation.

This master's thesis aims to create the NAVI Trace Toolkit, a web-based platform that enables data scientists and engineers to develop and manage traces efficiently. This interdisciplinary project combines novel energy network trace analysis approaches with software engineering principles and user experience design to create an innovative and practical application.

A. Scope and Outline

The work scope is focused on developing a web-based platform that demonstrates how to build an application using Python and JavaScript to enable trace developers to create complex and specific energy network trace analyses. Although the application is built on SPEN's NAVI platform, it is designed to be replicable, allowing any person or organisation to build the platform for their own trace development needs.

The platform provides a flexible, developer-friendly environment that supports the creation and management of trace logic. The application ultimately bridges the gap between rigid commercial GIS software and overly generic network graph libraries, offering a custom solution for energy network analysis.

Security hardening is considered beyond the scope of the thesis. Although the platform is web-based, it is intended to be executed locally; therefore, advanced security measures are not a priority of this project. As such, server deployment is also excluded from the scope of the project; however, the web design is modular and extensible, allowing for future deployment on a server if required.

II. Literature review and theoretical background

A. Trends and Importance of Analysis Tools in Distribution Networks

The evolution of distribution grids—driven by decentralisation, digitalisation, and decarbonisation—has led to a growing need for advanced analytical tools to support network optimisation and decision-making [1]. These tools are essential for integrating Distributed Energy Resources (DER), Electric Vehicles (EVs), and Smart Grid technologies, while maintaining grid reliability, efficiency, and resilience. [2]

IEEE literature highlights the importance of new analytical tools for tasks such as load forecasting, protection, electric power quality, and power factor improvement [2]. Networks are evolving from passive, radial configurations to more dynamic and bidirectional networks. Traditional manual or static analysis methods may no longer be sufficient.

Furthermore, new privatisation trends and deregulation of the distribution grid have increased the risk of the electrical grid becoming uncompetitive in a new market that can no longer rely on regulatory protection. In this new context, innovative tools are required to remain competitive, and making better use of spatial data is key to achieving efficiency in a capital-intensive market. [1]

B. Web Technologies in Electrical Grid

Bui et al. present a new paradigm for energy networks in which the Smart Grid (SG) benefits from internet technologies to become more interoperable and accessible [3]. The paper highlights the importance of referencing internet standardisation bodies such as the Internet Engineering Task Force (IETF) and the World Wide Web Consortium (W3C).

The IETF created the Constrained RESTful Environments (CoRE) working group to develop a RESTful protocol for constrained environments, resulting in the Constrained Application Protocol (CoAP) [4]. Other working groups in the IETF and W3C also developed standards to ease the integration of web-based protocols into the smart grid. 6LoWPAN working group efforts were focused on delivering IPv6 internet connectivity to constrained WPAN devices [5]. Meanwhile, the W3C's Efficient XML Interchange (EXI) worked on XML data compression, achieving up to 90% storage reductions [6].

Bui et al. developed a proof of concept for IoT web visualisation developed using Java and Google Maps, in which they present the network topology of connected devices. This work demonstrates the scalability and interoperability of web-based solutions. Similarly, Eren et al. present a web-based dispatcher information system for the electrical grid, YTBS (Yük Tevzi Bilgi Sistemi) [7]. This real-time, centralised monitoring platform presented by the TSO validates the viability of web-based platforms in grid operations.

In summary, web technologies offer a compelling response to the energy networks' needs. CoAP and data compression standards such as EXI highlight the efforts from the IETF and W3C to enable efficient web technologies implementation in energy grids. At the same time, applications like YTBS and the IoT platform by Bui et al. demonstrate the practical viability of web-based grid management platforms.

C. Traditional Approaches for Energy Network Topological and Spatial Analysis

In the past three decades, topological and network analysis of the grid have been performed using GIS, which private companies typically develop. GIS platforms offer powerful tools for managing and analysing spatial data, making them ideal for visualising and operating complex energy infrastructures [1].

SPEN adopted the NAVI platform, its GIS application to visualise and operate its distribution network. NAVI supports SPEN in making both business and technical decisions, ultimately leading to more efficient grid management.

Among the various capabilities offered by NAVI, trace analysis attracts attention as a particularly valuable tool. GIS proprietary software platforms, such as ESRI ArcGIS [8] or GE Smallworld [9], often include similar trace tools. Trace analysis is applied across utility networks to better understand their structure and behaviour; in the context of energy networks, typical applications include finding the nearest upstream protective device, calculating voltage drops, and optimising the balance of power flows in the network.

These GIS platforms include trace functions like finding the de-energised features and optimal switch configurations. The ESRI ArcGIS framework is the most advanced tool available for developing traces in proprietary software, which currently has eight built-in functions for tracing utility networks [10].

D. Web technologies for GIS

Integrating web technologies into geospatial applications has significantly transformed how networks are visualised and analysed. Web GIS platforms are emerging as a solution in the energy network management to enable dynamic map rendering, collaborative user interaction, and data-driven decision making.

Kuridža's research about the benefits of web GIS applications highlights the advantages of web-based platforms, including cross-platform execution regardless of the operating system, interface with external services, open standards, simplicity, and ubiquity on top of the technologies [11].

Developers can define their architecture based on GIS developer platforms (e.g., MapBox GL, CARTO), JavaScript open-source libraries (e.g., Turf, Leaflet, OpenLayers), or a combination of those tools. Own developed platforms include server-side data handling and RESTful APIs for communication [11].

One of the most prominent JavaScript libraries is MapLibre GL JS [12], an open-source tool for rendering vector maps using WebGL. MapLibre is a fork of MapBox GL JS, which is a proprietary software for developers that was once open source. MapLibre is an appropriate solution for many applications, including energy networks, due to its:

1. Custom styling capabilities,

2. Integration with open standards like GeoJSON and Mapbox Style Specification, and

3. Interoperability with frameworks like React, Angular, and maps like MapTiler and OpenStreetMap.

Fournier et al. developed an interactive decision support tool based on web maps for equitable energy planning, which was developed with stakeholders caring about social and environmental justice [13]. This project demonstrates the viability of web technologies for visualising energy networks for DER deployment. The web mapping tool was able to report imbalances between DER supply potential and grid capacity limits, helping in energy transition efforts. The web mapping platform used in this solution was ArcGIS Online, although open-source web-mapping software was considered.

E. Limitations of Existing Tools

There is currently no solution in the market that offers the flexibility to develop trace analysis tailored to specific cases in the electrical grid, while also being compatible with SPEN's distribution network data format. Existing Python network libraries are too generic to be applied to energy networks, and commercial Geographic Information System (GIS) software solutions are either too rigid to enable developers to create custom traces or lack a trace analysis tool.

For example, ArcGIS, which has the best trace tool in the market, is still constrained by the user interface, which has no code panel [14]. It does not allow users to combine multiple functions, preventing users from pipelining and creating tailored traces. While the user interface may be accessible to someone unfamiliar with trace development, it might limit developers who want to implement a particular trace logic or require a sandbox to experiment with it.

Commercial GIS platforms are mostly closed-source and license-restricted, hindering their adaptability for research. In many cases, their tracing functions are hardcoded and not easily extensible. Aside from the Oliver and Hoel (Esri) framework [15], there is currently no widely recognised theoretical framework, and it is not currently implemented in an open-source platform where developers can create traces upon those functions.

Additionally, open-source libraries such as NetworkX [16] or Graph-tool [17] provide powerful graph analysis functions, including shortest path, predecessors' retrieval, and cycles detection. Nevertheless, these libraries are too generic for direct application in electrical trace analysis. These libraries lack specific electrical domain applications such as impedance modelling and protective device behaviour. As a result, applying them to an energy network requires large amounts of custom code.

Moreover, SPEN has already developed its trace analysis tool in NAVI, eliminating the need for an external license platform. The NAVI Trace Toolkit addresses these challenges by building on SPEN's existing NAVI infrastructure. This project is based on SPEN NAVI developers' current Python scripts for trace development. Updating the script-based workflow is a logical step to enhance usability, accessibility, and long-term maintainability, since the current toolkit implementation lacks a unified trace framework and a user-friendly interface. A web-based implementation can provide an intuitive graphical interface that simplifies interaction with trace functions, reduces the learning curve for non-trace developers, and supports modular development.

In conclusion, existing commercial GIS platforms like ArcGIS offer built-in tracing functions; however, they are limited by rigid user interfaces and closed-source architectures. Open-source Python libraries like NetworkX and Graph-tool offer strong capabilities for graph analysis; however, they do not include the specific features needed for trace analysis in energy networks. These limitations prevent SPEN engineers and data analysts from innovating and customising traces.

F. Theoretical Background: Modelling Energy Networks for Trace Analysis

The theoretical background is based on Hoel et al.'s trace framework for utility networks [15], which is the only formal theoretical framework for trace analysis. Their framework will be adapted to the energy networks domain for the NAVI Trace Toolkit.

The energy network can be formally defined as a graph $U = (J, E)$, where:

J are the junctions representing physical components such as transformers, switches, fuses, meters or intersection points of lines. Each junction $j \in J$ has a geospatial coordinate (x, y) in a Euclidean plane.

E are the physical connections between junctions, i.e., the electric lines. Each edge $e \in E$ is modelled as a pair (u, v) connecting junctions u and v.

This graph abstraction allows efficient trace analysis by leveraging graph algorithms. The model's connectivity defines whether two features are logically connected, regardless of geometric coincidence. For example, a transformer does not directly touch a line but is connected. Two logically connected junctions may not be traversable if, for example, a protective device is open, or a line is disabled due to a fault condition.

Each junction and edge carries a dictionary of network attributes; nullable numeric, string, or Boolean values representing real-world properties such as: conductor material, cross-sectional area, impedance, reactance, entity id, OpenDSS simulation outputs, and installation metadata.

Attributes are used to compute metrics for analysis or to control traversability—whether electricity can flow in a path between connected features. For example, the ENABLE line attribute set to zero is not traversable. The protective device NORMALPOSITION_[A—B—C] attribute indicates if it is closed. If the normal position attribute is set to one, it is closed and traversable; if set to zero, it is open and not traversable. Additionally, if the breakpoin_phase[a—b—c] attribute is false, no breakpoint (i.e., open or fault condition) is present on any phase, thus not blocking traversal. The same attributes work at other junctions, such as switches and fuses, which are also normally closed.

The barriers are specific locations $B \subseteq U$ where traversability must terminate, they are based on physical device types or logical state constraints.

Finally, the filters are applied after traces are executed; thus, they do not affect traversability. They allow the subnetwork to be discovered. The filters can consist of a set of barrier nodes or categories.

PROBLEM FORMULATION AND STATEMENT

The core problem in trace analysis of energy networks can be formalised as follows:

Given:

An energy network $U = (J, E)$, where:

- J is the set of junctions (e.g., transformers, protections, switches, fuses)
- E is the set of edges (i.e., electric cables or logical connections)
- Each $j \in J$ and $e \in E$ have associated network attributes (e.g., status, voltage rating, type)
- A set of starting points $s \in S$ from which the trace begins the analysis.
- A set of barriers $b \in B$ that restricts traversability at determined locations.
- A traversability expression T, which stops the traversal based on a barrier function Boolean expression.
- The analysis function type (e.g., upstream, downstream, loops).

Find:

The subnetwork $U \subseteq U'$ that is reachable from S under the abovementioned constraints and the traversability definition.

TRACE TYPES

The trace framework supports multiple trace types and allows configurable and scalable trace analysis on energy networks. The main trace types developed by Hoel et al. are:

- Shortest path trace, which finds the shortest path between two starting points.
- Loops trace detects loops in the network and helps detect redundancies.
- Subnetwork trace extracts all traversable lines and devices from a subnetwork controller.
- In distribution energy networks, controllers are the transformers or the substations, the network's energy source.
- A subnetwork $SN \subseteq U$ is defined as a connected subset of the network that includes at least one network subnetwork controller. All junctions and edges in the subnetwork must be traversable, and subnetworks may overlap.
- Subnetwork controller trace, which identifies the controllers of the subnetwork.
- Upstream trace, which finds all controllers supplying power to a location.
- Downstream trace, which identifies all features receiving power from a location. It results from all reachable junctions and edges not belonging to the upstream trace.

TRACE CONFIGURATION EXAMPLE

To configure traces for tailored operations, we must include control over traversability, starting points, barriers, filters, and functions. Here is a brief explanation on how to configure a trace for calculating the grid and the number of customers affected by an electric fault:

- First, we need to define the fault location. That would be the starting point S.
- Then we set the barriers B to include open switches, faulted lines or any device that blocks traversal due to the fault or its state.
- For this operation, we run a downstream trace to identify all the junctions that are no longer reachable due to the fault. It simulates the propagation of the fault and identifies the disconnected areas.
- Finally, a filter is applied to get the junctions representing the customer nodes, and we count the number of customers based on the customer nodes' attributes.

This fault trace example can be applied to many business cases. It can be used for fault response prioritisation depending on the number of high-priority customers (e.g., hospitals). It can improve customer communication and transparency by informing affected customers about expected restoration times and service updates. The fault trace map can help maintenance teams to identify affected areas and prioritise restoration efforts quickly. This trace operation can simulate the impact of a fault or a planned maintenance, assisting engineers in assessing which customers will be affected.

THEORETICAL BACKGROUND: DIJKSTRA'S ALGORITHM

Dijkstra's algorithm is a foundational method in graph theory and network optimisation for finding shortest paths from a single node in a network. It is a node labelling and greedy algorithm that only works with non-negative edge weights.

Dijkstra's algorithm progressively selects the node with the smallest tentative distance from the source and updates the distances of its neighbours. Each node is assigned a label consisting of two attributes $(d(i), p(i))$, where $d(i)$ is the current shortest known distance from the source to the node i, and $p(i)$ is the predecessor node of the current shortest path to the node i.

The step-by-step algorithm for finding the shortest paths from the starting node s to all other nodes in the network is defined as follows [18]:

- Assign an initial tentative upper bound length to each node. Initially, the source node is assigned a distance of zero $d(s) = 0$, and infinity to the rest of the nodes $d(i) = \infty \; \forall i \neq s$. Label node s with $(0, -)$.
- Assuming c_{ij} is the weight between the two nodes i and j, select the labelled node i with $d(i)$ minimum. Node i is now scanned, and scanned nodes can never be labelled again. For each edge with weight c_{ij}, compute $d(j) = \min\{d(j), d(i)+c_{ij}\}$. Each neighbour node of i is marked as labelled.

- Repeat step 2 until all nodes are scanned, not labelled.

The correctness of the algorithm assumes that all weights are non-negative, meaning that once a node is scanned, the shortest path cannot improve any further.

Its time complexity is $O(E + J \log J)$ when implemented with Fredman and Tarjan's Fibonacci heap priority queue, making it suitable for large-scale graphs [18]. This algorithm is presented because it will be utilised for the critical path function.

III. NAVI Trace Toolkit Contributions

A. Trace Development Demonstrator

Proof-of-concept application showing how web technologies (HTML5, CSS3, JavaScript, Python Flask, MapLibre GL JS) can support trace development.

B. Python-Based Trace Framework

The backend extends Hoel et al. framework with a critical path function, functions tailored to the needs of SPEN, a modular trace logic for scalability and specific implementations for electrical utilities.

C. UX Evaluation

A usability study with SPEN developers measured: interface clarity, ease of configuration. effectiveness & efficiency, and overall satisfaction.

D. Use Case Demonstration

Critical Installation Path (CIP) prioritises maintenance tasks and prevents faults, validating the toolkit in practice.

E. Industrial Relevance

- Local Python app, deployable in SPEN's IT environment.
- Setup under 5 minutes.
- Independent of external tools.

IV. Methodology

The methodology combines user-centred design, agile development, and testing in a corporate environment with trace developers. The process was divided into four key phases:

A. Requirements Gathering and Domain Exploration

The project began with researching the traces domain and exploring the business necessities. This phase of the project involved:

- Researching trace development: learning about current trace analysis tools in the market and realising there was no platform for trace development.
- Deciding that the NAVI Trace Toolkit would be built on the foundations of the D. Hoel et al. framework.
- Reviewing current trace development workflow in SPEN: understanding current Python scripts and NAVI platform.
- Conducting informal interviews: identifying pain points in the current trace development workflow.

From the interviews, several usability pain points were identified:

- Manual script editing for repetitive tasks: users must modify Python scripts to define traces and set SSID subnetworks. Locating and modifying the correct parameters is often confusing.
- Lack of real-time UI feedback: there is no graphical interface with real-time feedback of trace results.
- Monolithic architecture: the running shell script executes all Python scripts simultaneously, even when only one task is needed. This is time-consuming and reduces flexibility.
- Error-prone scripts: users reported many errors during NAVI Trace initial execution (e.g., curl request failing, map loading issues).
- Accessibility: non-developers struggled to engage with trace development due to technical barriers. For example, the shell running script was not executable for Windows users, requiring a Linux terminal environment. Linux terminal environment installation required an IT request, delaying onboarding.

This phase helped to define the main functional requirements of the platform:

- UI SSID: the application must include a UI input where users can load a subnetwork by introducing its SSID. The application must support several SSID handling.
- Real-time execution and feedback: the application must provide real-time feedback when the user runs a trace—for example, notifying the user when trace logic is loading, showing error traceback, or displaying traces on a map when loading is finished.
- UI for repetitive tasks: the UI must have a section for changing the size and colour of nodes and lines by type. It should also allow the user to configure the features of default nodes and lines.
- Python code execution: the application must support real-time Python execution code. It should allow the user to input trace scripts, execute them, and view output (i.e., errors and results).
- Trace framework: the application must provide functions that allow the user to write trace scripts efficiently. It must give comprehensive documentation with examples to help understand the functions.

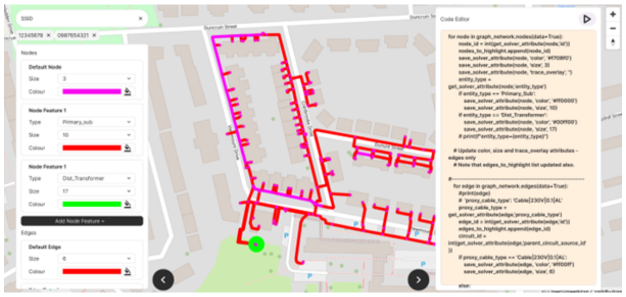
B. Wireframing and initial proposal

Once requirements were gathered, wireframes were created to get the application's visual representation and illustrate the intended user flow. These wireframes were presented to the stakeholders, including the manager, for feedback and approval. The wireframes served as a graphic tool to refine scope, align expectations with stakeholders before development began, and as a roadmap to guide the development toward the final product.

Figure 1 shows the wireframe proposal developed using Figma [19]. This user interface aims to fulfil user requirements while providing a great user experience. Besides the map, the UI comprises four main elements that the user can interact with:

- Map: the interactive map on the background is the primary visualisation canvas of the traces, enabling intuitive interaction. When the user hovers over the traces, information about the nodes and lines is displayed to facilitate the understanding of the network.
- Subnetwork input: a search bar is included at the top, where the user can input an SSID to load the subnetwork on the map. The user can load as many subnetworks as they want.
- Features panel: the features panel enables the user to configure basic nodes and lines settings.
- Code panel: this panel will include a Python code editor for script editing. Code highlighting and editor functionalities are crucial, since the user needs a smooth coding experience that does not hinder their performance.

The buttons positioned along the bottom of the two panels allow the user to hide the panels, providing a bigger and cleaner view of the map.



Size	3
Colour	🟪
Type	Primary_sub
Size	10
Colour	🟥
Type	Dist_Transformer
Size	17
Colour	🟩
Size	6
Colour	🟥

```
for node in graph_network.nodes(data=True):
    node_id = int(get_solver_attribute(node,'id'))
    nodes_to_highlight.append(node_id)
    save_solver_attribute(node, 'color', '#f708f0')
    save_solver_attribute(node, 'size', 3)
    save_solver_attribute(node, 'trace_overlay', '')
    entity_type =
get_solver_attribute(node,'entity_type')
    if entity_type == 'Primary_Sub':
      save_solver_attribute(node, 'color', '#ff0000')
      save_solver_attribute(node, 'size', 10)
    if entity_type == 'Dist_Transformer':
      save_solver_attribute(node, 'color', '#00ff00')
      save_solver_attribute(node, 'size', 17)
    # print(f"entity_type={entity_type}")

  # Update color, size and trace_overlay attributes -
edges only
  # Note that edges_to_highlight list updated also.

#-----------------------------------------------------
  for edge in graph_network.edges(data=True):
    #print(edge)
    # 'proxy_cable_type': 'Cable|230V|0.1|AL'
    proxy_cable_type =
get_solver_attribute(edge,'proxy_cable_type')
    edge_id = int(get_solver_attribute(edge,'id'))
    edges_to_highlight.append(edge_id)
    circuit_id =
int(get_solver_attribute(edge,'parent_circuit_source_id'
))
    if proxy_cable_type == 'Cable|230V|0.1|AL':
      save_solver_attribute(edge, 'color', '#ff00ff')
      save_solver_attribute(edge, 'size', 6)

    else:
```

Fig. 1. Wireframe proposal.

C. Agile development process

The application's development followed an agile methodology, with 2-week sprint cycles for continuous refinement of features based on feedback. This phase was characterised by regular check-ins, incremental implementation of features based on priority, and updates to the requirements based on user input and technical feasibility. This user-centred approach ensures user involvement on the platform and responds to real user needs. Agile methodology kept the criteria aligned with SPEN's operational context.

D. User Testing and Evaluation

Once the application was fully operative and the core functionality was implemented, the application was tested by five SPEN engineers and trace developers. The evaluation consisted of three categories:

- Quantitative task-based testing: users were asked to complete tasks (e.g., installing the app, launching it, running a trace), and completion times were recorded.
- Quantitative feedback survey: a questionnaire with a Likert scale was conducted to assess user experience on usability and performance. Users were also asked to report the time spent running the previous Python scripts for trace development.
- Qualitative feedback: users provided open feedback on the usability, interface aesthetics, and overall experience.

The evaluation was conducted on SPEN corporate computers to ensure the platform's operation in corporate settings.

V. Implementation

The NAVI Trace Toolkit was implemented as a modular web application tailored for SPEN's operational environment. It uses a Flask Python backend and a responsive HTML/CSS frontend, with MapLibre GL JS for map rendering. Subnetworks are visualised using GeoJSON layers, and trace logic is executed in real time via a built-in Ace Editor. The backend exposes RESTful endpoints for trace execution, subnetwork loading, and user guidance, and integrates a custom trace library based on Hoel et al.'s framework.

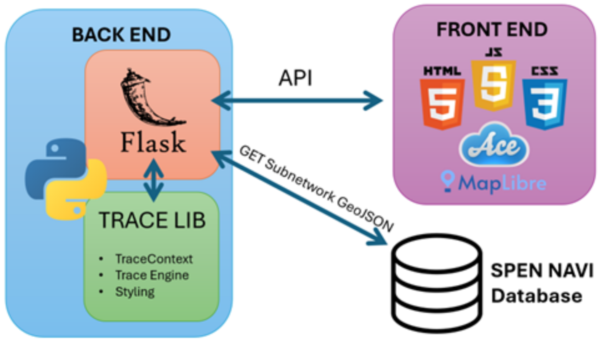


Fig. 2. The NAVI Trace Toolkit System Architecture

A. Front-end Implementation

The front-end was developed using standard web technologies—HTML and CSS—providing a clean and responsive interface. It allows the user to interact with network data and run trace logic.

1) Map: The interactive map is implemented with MapLibre GL JS, using OpenStreetMap as the source of map tiles. Traces are displayed on the map using MapLibre GL JS layers. The layers are rendered using GeoJSON sources; each subnetwork has a GeoJSON associated with it. Two layers are created for each subnetwork: one for junctions, using Point GeoJSON geometry, and another for lines, using LineString GeoJSON geometry. If the user wants to display a message on the map with the Python script, a new layer is created for messages.

2) Code Editor: The code panel is implemented using the Ace Editor, an embeddable, lightweight, and highly customizable code editor written in JavaScript. It provides features commonly found in plain-text native editors, such as Sublime Text or Vim. Key features of the Ace Editor are syntax highlighting and editing capabilities such as line wrapping, line numbering, multiple line editing, and tab spacing control.

3) Other front-end implementations: A hiding functionality was implemented in both panels to enhance user interaction and optimise screen map view, combining CSS transitions, JavaScript, and local storage management (Figure 3). This solution provides a smooth and persistent user experience. The sliding function is encapsulated in a JavaScript function, which selects the element, retrieves the current state, and uses the CSS translateX function to change the panel's position. To improve data exploration, listeners were added to the map to detect when the user hovers over the lines and nodes of the network. Upon hover, a function that displays the element's information is triggered (Figure 3).

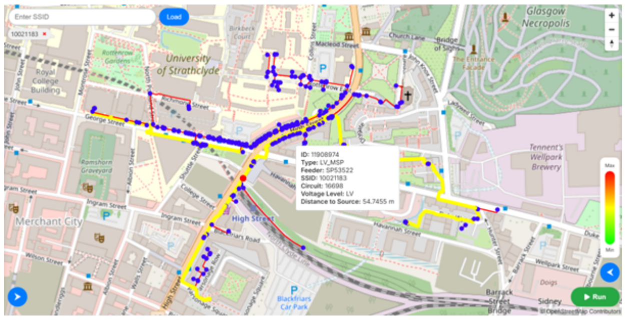

ID: 11908974
Type: LV_MSP
Feeder: SP53522
SSID: 10021183
Circuit: 16698
Voltage Level: LV
Distance to Source: 54.7455 m

Fig. 3. Panels Hidden with Node Data Displayed on Hover

Local storage is implemented to persist the user-defined configurations and session data: node and lines UI configuration, SSIDs of the loaded subnetworks, Python script, and the hiding state of the panels. This approach retains user preferences and session context, providing a more personalised user experience.

B. Back-end Implementation

The back-end system was developed using Flask, a Python framework for web development. The back-end serves as the front-end interface's computational engine and data orchestrator. Its main tasks are handling front-end user requests, parsing from the NAVI API fetched data to the NAVI Trace Toolkit data format, executing trace logic, and generating geospatial outputs for visualisation.

1) API REST Endpoints: The API exposes multiple RESTful endpoints that support data retrieval and custom trace execution. The endpoints are designed to be stateless and modular, enabling seamless integration with the front-end and scalability. The backend has five endpoints described below:

1) / (Index Route): It loads the main HTML interface using Flask's render_template function. This route does not process data and is only used for initial page load.
2) /subnetwork (GET/POST): This route handles requests to load specific subnetwork data based on its SSID. If the corresponding file does not exist locally, it fetches it on the NAVI SPEN API, stores it, and processes it using the execute_trace() function. The trace result is formatted to GeoJSON using the create_geojson() function and returned to the front-end for rendering. This endpoint supports code customisation and node and line styling, so that when the user loads the subnetwork, it has the user-configured styling.
3) /code (POST): This endpoint executes user-configured styling and Python code across one or multiple subnetworks. It reads the graph data from local storage and applies the user-configured styling and code using the execute_trace() function. It generates a GeoJSON file for each subnetwork. The output of the user's code execution is captured using Python's contextlib.redirect_stdout, allowing the endpoint to return the code output and trace result. Additionally, messages for map display are handled through helper functions.
4) /delete-ssid (POST): This route removes cached graph and GeoJSON files associated with an SSID. This endpoint handles proper data lifecycle management and maintains a clean work environment.
5) /user-guide (GET): This endpoint dynamically renders a Markdown-based user guide into HTML using the markdown Python library.

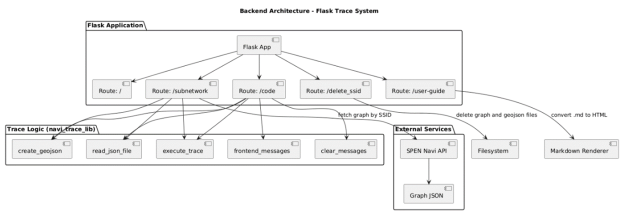


Fig. 4. Backend architecture and Flow

2) Trace Engine: The backend uses a custom trace library (`navi_trace_lib`), encapsulating the energy network trace analysis logic. Key helper functions used by the REST API are:

- `read_json_file()`: Parses from JSON format to NetworkX graph.
- `execute_trace()`: Executes the trace algorithm based on the user's code and configuration.
- `create_geojson()`: Converts trace results into GeoJSON format suitable for front-end MapLibre GL rendering.
- `frontend_messages` and `clear_messages()`: Manage displayable messages on MapLibre GL.

The trace library includes built-in functions that support advanced energy analysis and visualisation, making the development of new traces more efficient. These functions are based on the theoretical framework proposed by Hoel et al. They fall into four categories:

- **Trace Styling:** Functions that allow users to customise the visual representation of network elements.
- **Messaging:** Functions that support user map annotation, e.g., `send_message()` displays text labels on the map at specified coordinates.
- **Utilities:** Functions such as `get_color_gradient_continuous()` for gradient colour schemes and `get_node_by_id()` or `get_entity_id()` for entity lookup.

- **Network Exploration:** Functions that implement trace algorithms for interactive network exploration. The `trace()` function provides a unified interface to:
  - Upstream and downstream tracing
  - Loop detection
  - Shortest path computation
  - Subnetwork extraction and control

Additionally, a custom `critical_path()` function computes the shortest path from a source to each node, including filtering by entity type. Pseudocode:

TABLE I
CRITICAL PATH FUNCTION PSEUDOCODE USING WEIGHTED DIJKSTRA TRAVERSAL

```
Input:   ctx: TraceContext object with the network graph
         source_node_id: ID of the starting node
         weight_func: Function to compute edge weights
         filters: Set of entity types to filter
Output:  max_distance: Maximum shortest-path weight among filtered nodes
         path_nodes: List of nodes forming the critical path
1        Initialize an empty weighted graph G'
2        For each edge (u, v) in the original graph:
3           Try:
4              Compute weight ← weight_func(u, v, edge_data)
5           Catch error:
6              Set weight ← ∞
7           Add edge (u, v) with weight to G'
8        Run Dijkstra's algorithm on G' from source_node_id
9        Store shortest path distances and paths: lengths, paths
10       Initialize max_distance ← 0
11       Initialize max_node ← null
12       For each node in the original graph:
13          If node.entity_type ∈ filters:
14             If node is reachable (node ∈ lengths):
15                If lengths[node] > max_distance:
16                   Update max_distance ← lengths[node]
17                   Update max_node ← node
18       Retrieve path_nodes ← paths[max_node]
19       Return (max_distance, path_nodes)
```

C. Libraries

The NAVI Trace Toolkit uses Python libraries to support its capabilities:

- **Markdown** [20]: Render Markdown to HTML.
- **Requests** [21]: Retrieve network graph from NAVI API.
- **Ujson** [22]: Efficient JSON parsing/manipulation for GeoJSON.
- **NetworkX** [16]: Store subnetworks and implement trace algorithms.
- **Flask** [23]: Web framework for routing, endpoints, and HTML rendering.

D. Setup and Application Launcher

Two `.bat` files allow users to install and launch the toolkit without command-line interaction:

- `install.bat`: Creates Python environment and installs required libraries.
- `run.bat`: Launches the application within the environment.

A user guide supports setup, including Python installation, executing `install.bat`, and running `run.bat`.

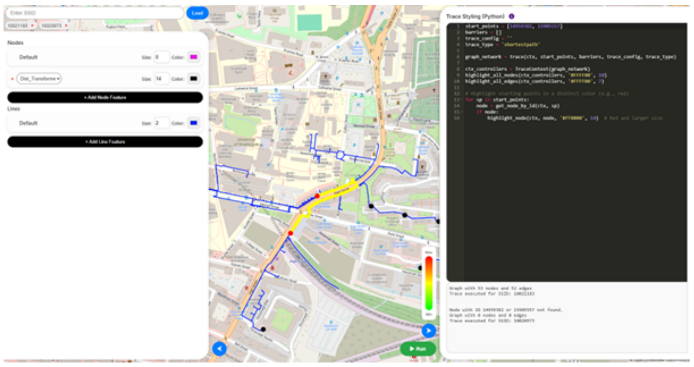
VI. VISUAL RESULT AND INTERFACE DESCRIPTION

The NAVI Trace Toolkit provides a rich, interactive interface (Figure 5) that comprises geospatial visualisation, trace styling configuration, and trace scripting capabilities. The visual output includes the main components designed in the wireframe phase: the map display, the SSID loader, the feature sidebar, and the code editor panel.

The central element of the interface is the dynamic map rendered using MapLibre GL. The map visualises the energy network, with nodes and edges styled according to parameters defined by the user.

On the left side of the interface, a configuration panel enables the user to define the properties for nodes and lines. In Figure 5, default nodes are turned off by setting their size to 0, while transformers are highlighted in black. Lines are displayed in blue. This styling capability enhances the clarity of the network representation and aids in exploratory analysis.

Above the styling panel, the user finds the SSID loader to display multiple subnetworks. The SSIDs of these subnetworks are shown below the SSID input. This feature enables multi-network and comparative analysis, improving analytical flexibility.



```
start_points = [14959382, 15900557]
barriers = []
trace_config = ''
trace_type = "shortestpath"

graph_network = trace(ctx, start_points, barriers, trace_config, trace_type)

ctx_controllers = TraceContext(graph_network)
highlight_all_nodes(ctx_controllers, '#FFFF00', 10)
highlight_all_edges(ctx_controllers, '#FFFF00', 7)

# Highlight starting points in a distinct color (e.g., red)
for sp in start_points:
    node = get_node_by_id(ctx, sp)
    if node:
        highlight_node(ctx, node, '#FF0000', 14)  # Red and larger size
```

```
Graph with 53 nodes and 52 edges
Trace executed for SSID: 10021183

Node with ID 14959382 or 15900557 not found.
Graph with 0 nodes and 0 edges
Trace executed for SSID: 10020973
```

Fig. 5. The NAVI Trace Toolkit Application

The right side of the interface features the Python code editor, where the user can write and execute trace scripts. The Python editor supports the built-in functions developed in this project for tailored trace development—such as `trace()`, `highlight_node()`, and `TraceContext()`. The user can execute traces, providing real-time updates directly on the map by pressing the run button next to the right panel. In Figure 5, a shortest path trace is executed between two nodes, with styles applied to the results. Below the editor, the user can find UI feedback, including trace outputs and error messages (e.g., missing nodes).

A. Application comparison

1) User Flows: The NAVI Trace toolkit has significantly enhanced user flows compared to the previous implementation. The system now supports a modular, interactive, and user-friendly interface that enables users to perform more flexible and efficient trace analysis.

a) Current User Flow: The NAVI Trace toolkit's current flows allow the user to:

- Load subnetworks individually by selecting the SSIDs, enabling the user to select multiple subnetworks within a single session.
- Modify node and line features dynamically, depending on their attributes and entity types.
- Run Python scripts using the code panel and the built-in functions to execute trace logic and apply styling.
- Refer to the markdown documentation to understand how to use the functions.

All these features are integrated into a web interface that instantly displays results, without needing to refresh or reload external files.

b) Previous User Flow: By contrast, the previous implementation had no user interface or modularity. Thus, the user was required to:

- Execute all steps in a single shell script, which runs all Python scripts, including data loading, styling, trace execution, and HTML rendering.
- Refresh the static HTML file generated at the end of execution to view the results.
- Limit analysis to one network at a time, constraining the comparison and user flows across multiple networks.
- Operate without integrated documentation or examples, depending solely on guidance from expert trace developers.

The previous workflow was functional but rigid and time-consuming, only accessible to users with specialised trace programming skills.

TABLE II
USER FLOW COMPARISON BETWEEN PREVIOUS IMPLEMENTATION AND THE NAVI TRACE TOOLKIT

Feature	**Previous Implementation**	**NAVI Trace Toolkit (Current)**
Subnetwork Loading	Manual, one at a time	Interactive, multiple SSIDs
Node/Edge Styling	Script-based and static	Dynamic, UI-driven or scripted
Script Execution	Full script required	Modular, panel-based
Result visualization	Refresh the HTML file	Real-time map rendering
Documentation Access	No formal documentation	Documentation with new functions and integrated with a Markdown HTML viewer
User Interface	None	Interactive web-based UI

B. Execution Time Comparison

Performance experiments were conducted using a corporate Dell laptop with an Intel i7 processor. Each test was repeated 100 times to ensure statistical reliability.

- **Trace Execution:** The average time to execute a trace in the NAVI Trace toolkit on a single network is 1.25 seconds, with a standard deviation of ± 0.27 seconds. This metric reflects the time to run the trace logic and render the result on the user interface.
- **Subnetwork Loading:** Loading and rendering of a single subnetwork, i.e., input an SSID in the NAVI Trace Toolkit, takes 3.92 seconds on average with a standard deviation of ± 0.55 seconds. This workflow comprises retrieving the subnetwork from the SPEN NAVI server, generating the GeoJSON, and rendering it in the application. This metric compares the application performance with the previous bulk implementation, which followed a similar workflow but lacked a user interface.
- **Previous Monolithic Workflow:** The previous Python-based workflow took an average of 10.73 seconds to complete, equivalent to the current application's subnetwork loading workflow, with a standard deviation of ± 1.28 seconds. This included all operations executed in a single shell script.

Additionally, experiments were conducted with two to nine subnetworks loaded simultaneously to assess network trace scalability performance. The `subnetwork()` function was selected for the experiment because it traverses the entire subnetwork, exploring all nodes and edges. The results show that the execution time increases exponentially with the number of subnetworks. Despite this performance, the NAVI Trace Toolkit is still more efficient than the previous implementation with one subnetwork, even when handling up to 9 subnetworks. This improvement in multi-network rendering and loading times is attributed to the modular architecture of this project, as well as the use of efficient algorithms and high-performance modules such as `ujson`.

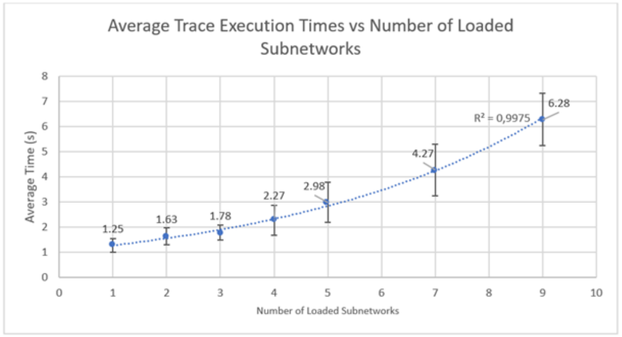


Fig. 6. Average Trace Execution Times vs Number of Loaded Subnetworks

TABLE III
EXECUTION TIME COMPARISON BETWEEN THE NAVI TRACE TOOLKIT AND PREVIOUS IMPLEMENTATION

Scenario	**Average Time (s)**	**Standard Deviation (s)**
Current Trace Execution (1 subnetwork)	1.25	0.27
Current Subnetwork Loading	3.92	0.55
Previous Workflow (1 subnetwork)	10.73	1.28
Current Trace Execution (9 subnetworks)	6.28	1.04

C. User study results and evaluation

1) Task-based Performance Evaluation: The engineers were asked to complete the initial setup and five additional tasks in the app. Each participant had 30 minutes to complete the setup and the five tasks. ISO 9241 defines usability of the users based on effectiveness and efficiency metrics [24]. Effectiveness measures how wholly and accurately the users achieve their goals:

$$\text{Effectiveness (\%)} = \frac{\text{Number of completed tasks}}{\text{Number of Tasks}} \times 100$$

Efficiency measures the time expended in relation to the results:

$$\text{Efficiency} = \frac{\sum_{j=1}^{R}\sum_{i=1}^{N} n_{ij}}{\sum_{j=1}^{R}\sum_{i=1}^{N} t_{ij}}$$

Where:

- R: Number of users
- N: Number of tasks
- n_{ij}: result for task i by user j; 1 if completed, 0 otherwise
- t_{ij}: time spent by user j to complete task i (if not completed, time measured until user gives up)

Initial setup took an average of 5.05 minutes with a standard deviation of 0.64 minutes. The effectiveness of the user test was 100%. The efficiency was 1.64 tasks per minute, demonstrating high usability.

2) Comparative Setup Time Analysis: In contrast, the previous toolkit implementation required significantly more time to set up: 40% of users reported setup times between 1 and 1.5 hours, 40% required between 30 and 60 minutes, and 20% completed setup between 15 and 30 minutes.

3) Survey-Based UX Evaluation: A structured survey using a 1-5 Likert scale (1 = Strongly Disagree, 5 = Strongly Agree) evaluated nine dimensions:

1) The toolkit helps me complete trace-related tasks efficiently.
2) The interface is clear and easy to navigate.
3) The visual design helps me understand the trace results and the energy network.
4) The toolkit loads quickly and responds promptly.
5) The interface works well on my screen resolution and setup.
6) It is faster than the previous version.
7) It is more intuitive than the previous version.
8) I prefer the new toolkit because it provides built-in trace functions.
9) Overall, I prefer the new toolkit over the previous one.

The average ratings are plotted in Figure 7.

All dimensions averaged over 4, showing that tasks were completed efficiently and the interface was intuitive and clear, with Question 3 receiving the highest rating. The toolkit responded quickly, displayed well on user screens, and the built-in trace functions were positively rated. Overall, users preferred the new toolkit.

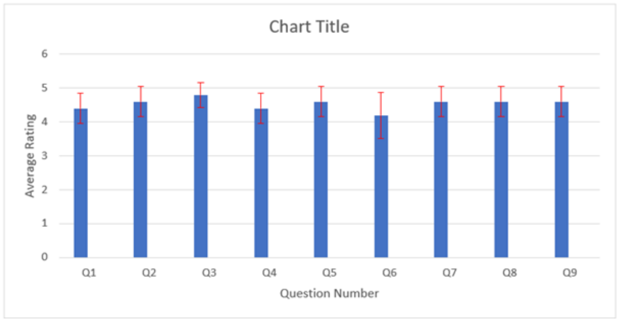


Fig. 7. Average Likert Scores for User Experience Questions

4) Open-Feedback and Feature Recommendations: Users suggested the following enhancements:

- Onboarding for new trace developers.
- Import/export feature for Python scripts and collaborative workflows.
- Export trace data structure (e.g., CSV).
- Minor improvements: auto-zoom to loaded subnetworks, resizable code panel.

These recommendations highlight interest in collaborative features and improving accessibility for new users.

VII. Use Case

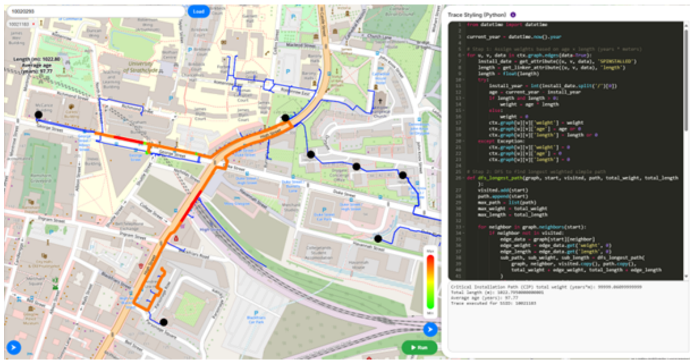
In energy distribution networks, maintaining the integrity and reliability of infrastructure is crucial. As these networks age, the likelihood of faults and safety hazards increases [25]. The components of the distributed energy network deteriorate over time, making proactive identification and intervention essential for preventing failures. Traditionally, energy distribution companies have relied on periodic inspections or reactive maintenance after problems occur; however, these methods can be costly and risky.

To analyse the age of the network using data traces, we introduced the concept of the Critical Installation Path (CIP). This data-driven metric helps identify the most vulnerable path within the network. The CIP is defined as the electrical path composed of the oldest and longest connected lines, calculated by multiplying the age of each line (years since installation) by its length (meters). The CIP weight metric serves as a proxy for infrastructure exposure, capturing both the temporal degradation and the spatial extent of the network. The age of the lines indicates how components degrade and become obsolete over time, while the length of the lines represents their physical footprint and the potential area affected by any failures. Therefore, older and longer lines are more likely to disrupt service continuity.

A. First Approach: Depth-First Search (DFS) for Longest Weighted Path

In the initial method for computing the CIP, a custom Depth-First Search (DFS) algorithm was implemented to traverse the network and identify the longest simple path with the highest cumulative CIP weight. The DFS algorithm explores all possible simple paths starting from each node in the subnetwork. The largest CIP is tracked through the iteration and updated when a path with a higher CIP is found. After the DFS algorithm is run with all the nodes, the most critical installation path is highlighted using a continuous colour gradient based on age.

This approach provides a global view of the subnetwork's longest CIP infrastructure chain, which may require attention. However, since DFS explores all possible paths, it is computationally intensive in large or densely connected networks. Additionally, it does not account for energy network logic, which is relevant in real-world energy distribution scenarios.



```
from datetime import datetime

current_year = datetime.now().year

# Step 1: Assign weights based on age × length (years * meters)
for u, v, data in ctx.graph.edges(data=True):
    install_date = get_attribute((u, v, data), 'SPINSTALLED')
    length = get_linker_attribute((u, v, data), 'length')
    length = float(length)
    try:
        install_year = int(install_date.split('/')[0])
        age = current_year - install_year
        if length and length > 0:
            weight = age * length
        else:
            weight = 0
        ctx.graph[u][v]['weight'] = weight
        ctx.graph[u][v]['age'] = age or 0
        ctx.graph[u][v]['length'] = length or 0
    except Exception:
        ctx.graph[u][v]['weight'] = 0
        ctx.graph[u][v]['age'] = 0
        ctx.graph[u][v]['length'] = 0

# Step 2: DFS to find longest weighted simple path
def dfs_longest_path(graph, start, visited, path, total_weight, total_length
    ):
    visited.add(start)
    path.append(start)
    max_path = list(path)
    max_weight = total_weight
    max_length = total_length

    for neighbor in graph.neighbors(start):
        if neighbor not in visited:
            edge_data = graph[start][neighbor]
            edge_weight = edge_data.get('weight', 0)
            edge_length = edge_data.get('length', 0)
            sub_path, sub_weight, sub_length = dfs_longest_path(
                graph, neighbor, visited.copy(), path.copy(),
                total_weight + edge_weight, total_length + edge_length
            )
```

```
Critical Installation Path (CIP) total weight (years*m): [illegible]
Total length (m): 1022.7950000000001
Average age (years): 97.77
Trace executed for SSID: 10021103
```

Fig. 8. Critical Installation Path (CIP) Trace with First Approach (DFS), with Transformers Highlighted in Black

Let J be the number of junctions and E the number of edges. The time complexity of this DFS approach is:

- Overhead complexity due to computing the weights of each edge adds a linear time complexity $O(E)$.
- Standard DFS has $O(J+E)$ time complexity. However, in this approach, DFS explores all possible simple paths from every node. A simple path is a path that does not repeat any nodes.
- In an undirected graph with V nodes, the number of simple paths can be up to $O(2^J)$.
- Since the algorithm runs a DFS simple path search on every node, the total time complexity becomes: $O(J2^J)$.

This approach is not scalable for large and dense networks. It serves as a conceptual baseline but requires a computationally feasible and operationally relevant reformulation.

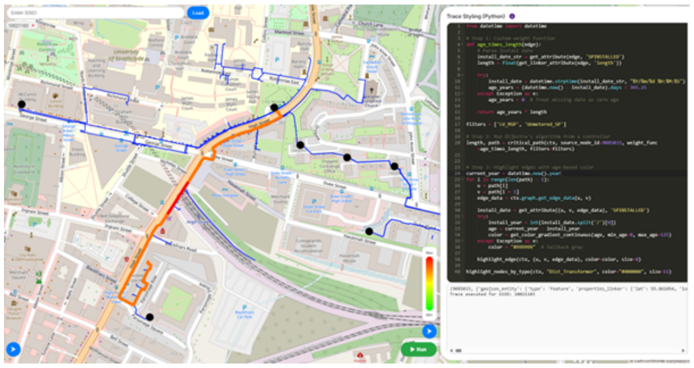
B. Second Approach: Dijkstra with Custom Weight Function

Identifying the longest path in a network is a well-known NP-hard problem. In the context of energy distribution, where networks can have thousands of nodes and edges, using an exhaustive search method such as DFS becomes impractical.

To address this, we introduced constraints grounded in energy network logic. During initial testing, we observed that some paths identified as critical (high CIP) did not represent vulnerable customer connections. For instance, if a transformer lies in the middle of a path (Figure 8), a fault on one side may not impact customers on the other side due to protective devices that isolate the faulted section. This insight led to a more realistic formulation: instead of searching globally, we define an energy source point, typically a transformer or network controller, and trace outward using Dijkstra's algorithm.

Although Dijkstra is designed to find shortest paths, not longest ones, it becomes useful when we reinterpret the problem: we are not seeking the longest path overall, but rather the critical path from an energy source to a customer. We first compute with Dijkstra the shortest path for each customer, then look for the longest path among the shortest.

If a customer can be reached via multiple paths—one with high CIP and another with low CIP—the presence of the low CIP prevails over the high CIP because it ensures resilience. Consequently, the lowest CIP is the most significant for each node, which makes Dijkstra's algorithm appropriate.



```
from datetime import datetime

# Step 1: Custom weight function
def age_times_length(edge):
    # Parse install date
    install_date_str = get_attribute(edge, 'SPINSTALLED')
    length = float(get_linker_attribute(edge, 'length'))

    try:
        install_date = datetime.strptime(install_date_str, "%Y/%m/%d %H:%M:%S")
        age_years = (datetime.now() - install_date).days / 365.25
    except Exception as e:
        age_years = 0  # Treat missing date as zero age

    return age_years * length

filters = ['LV_MSP', 'Unmetered_SP']

# Step 2: Run Dijkstra's algorithm from a controller
length, path = critical_path(ctx, source_node_id=[illegible], weight_func
    =age_times_length, filters=filters)


# Step 3: Highlight edges with age-based color
current_year = datetime.now().year
for i in range(len(path) - 1):
    u = path[i]
    v = path[i + 1]
    edge_data = ctx.graph.get_edge_data(u, v)

    install_date = get_attribute((u, v, edge_data), 'SPINSTALLED')
    try:
        install_year = int(install_date.split('/')[0])
        age = current_year - install_year
        color = get_color_gradient_continuous(age, min_age=0, max_age=125)
    except Exception as e:
        color = "#999999"  # fallback gray

    highlight_edge(ctx, (u, v, edge_data), color=color, size=8)

highlight_nodes_by_type(ctx, "Dist_Transformer", color="#000000", size=15)
```

```
([illegible]), {'geojson_entity': {'type': 'Feature', 'properties_linker': {'lat': 55.861054, 'lo
Trace executed for SSID: 10021103
```

Fig. 9. Critical Installation Path (CIP) Trace Using Second Approach (Dijkstra), with Transformers Highlighted in Black

This approach dramatically reduces computational complexity by avoiding exhaustive path calculation and aligns with energy network principles, balancing computational feasibility and operational relevance.

The step-by-step breakdown describing the CIP analysis using Dijkstra is:

1) Define the custom weight function: CIP $=$ age $\times$ length.
2) Run the `critical_path()` function with Dijkstra's algorithm from a controller node. Compute the shortest paths to all nodes, filter customer nodes, and identify the customer with maximum CIP value.
3) Highlight the critical path with edges coloured using a gradient based on age.

The time complexity of this algorithm is:

- Dijkstra's algorithm: $O(E + J \log J)$ using a Fibonacci heap priority queue.
- Computing the custom weight function: $O(E)$.
- Filtering customer nodes and finding maximum CIP: $O(V)$.

Total complexity: $O(E + J \log J)$, making it scalable for large energy networks.

VIII. Conclusion and Future Work

The NAVI Trace Toolkit marks a transformative step in energy network trace development, evolving from a rigid, script-based workflow into a modular, interactive, and user-friendly

TABLE IV
DIJKSTRA-BASED CIP TRACE PSEUDOCODE WITH CUSTOM WEIGHT FUNCTION

```
Input:   graph: Network graph with metadata (installation date, length)
         source_node: Defined transformer or controller
         filters: Set of customer node types (e.g., LV_MSP, Unmetered_SP)
Output:  critical_path: Shortest path to the customer with the highest CIP
         Highlighted edges: Visual representation of the path

1    Define CIPWeight(edge):
2        installDate ← get installation date from edge
3        length ← get physical length from edge
4        age ← compute age from installDate using current date
5        return age × length
6    filters ← CustomersEntityTypes
7    sourceNode ← predefined transformer or controller
8    (Distance, Path) ← critical_path(graph, sourceNode,
     CIPWeight, filters)
9    For each consecutive edge (u, v) in Path:
10       age ← compute age from edge installation date
11       color ← map age to gradient colour
12       highlightEdge(u, v, color = color, size = medium)
13   Return Path
```

platform. By incorporating a user-friendly web interface, real-time Python execution, and integrated tracing functions, the toolkit enables engineers to conduct intricate trace analyses with enhanced efficiency and accessibility.

The CIP use case demonstrates the toolkit's real-world analytical capabilities. The DFS-based approach, while conceptually valuable, was computationally intensive. The Dijkstra-based method offers a scalable, practical solution that aligns with energy network principles and enables targeted assessments of infrastructure risk.

Future work includes:

- Onboarding modules to assist new users in understanding trace concepts.
- Expanding trace function library and enhancing critical path functionality with probabilistic failure models.
- Integration with external systems, such as SCADA and the Common Information Model (CIM), to improve interoperability.

These developments will enhance the NAVI Trace Toolkit as a comprehensive and scalable energy network analysis solution.

REFERENCES

[1] P. A. Longley, M. F. Goodchild, D. J. Maguire and D. W. Rhind, *Geographical Information Systems: Principles, Techniques, Management and Applications*, West, Sussex: Wiley Publishing Company, 1999.

[2] A. A. Sallam and O. P. Malik, *Electric Distribution Systems*, 2nd ed., Hoboken, NJ: Wiley-IEEE Press, 2019.

[3] N. Bui, A. P. Castellani, P. Casari and M. Zorzi, "The Internet of Energy: A Web-Enabled Smart Grid System," *IEEE Network*, vol. 26, pp. 39–45, July 2012.

[4] Z. Shelby, K. Hartke and C. Bormann, "The Constrained Application Protocol (CoAP)," 2014. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7252. [Accessed Aug 2025].

[5] P. Thubert, C. Bormann, L. Toutain and R. Cragie, "IPv6 over Low-Power Wireless Personal Area Network (6LoWPAN) Routing Header," 2017. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc8138. [Accessed Aug 2025].

[6] D. Peintner and R. Kyusakov, "Efficient XML Interchange (EXI) Format 1.0 (Second Edition)," 2014. [Online]. Available: https://www.w3.org/TR/exi/. [Accessed Aug 2025].

[7] S. Eren et al., "A ubiquitous Web-based dispatcher information system for effective monitoring and analysis of the electricity transmission grid," *Energy*, vol. 116, pp. 1044–1056, Oct 2016.

[8] Esri, "ArcGIS Geospatial Platform Overview," 2025. [Online]. Available: https://www.esri.com/en-us/arcgis/geospatial-platform/overview. [Accessed Jul 2025].

[9] G. E. Vernova, "Geospatial Network Management (Smallworld GIS)," 2025. [Online]. Available: https://www.gevernova.com/software/products/geospatial-network-management-smallworld-gis. [Accessed Jul 2025].

[10] Esri, "Configure a trace," 2025. [Online]. Available: https://pro.arcgis.com/en/pro-app/3.3/help/data/utility-network/configure-a-trace.htm. [Accessed Jul 2025].

[11] B. Kuridža, "Potentials and Limitations of Web GIS in the Utility Industry," MSc in Geoinformatics, Aalborg University, Copenhagen, 2019.

[12] "MapLibre GL JS Documentation," 2025. [Online]. Available: https://maplibre.org/maplibre-gl-js/docs/. [Accessed Aug 2025].

[13] E. D. Fournier, F. Federico, R. Cudd and S. Pincetl, "Building an interactive web mapping tool to support distributed energy resource planning using public participation GIS," *Applied Geography*, vol. 152, p. 102877, Jan 2023.

[14] D. Oliver and E. Hoel, "Exploring the ArcGIS Utility Network Trace Framework," 2023. [Online]. Available: https://www.esri.com/arcgis-blog/products/utility-network/data-management/exploring-the-arcgis-utility-network-trace-framework. [Accessed Jul 2025].

[15] D. Oliver and E. G. Hoel, "A Trace Framework for Analyzing Utility Networks: A Summary of Results," Redlands.

[16] A. Hagberg, D. Schult and P. Swart, "NetworkX," 2025. [Online]. Available: https://networkx.org/. [Accessed Jul 2025].

[17] T. Peixoto, "Graph-tool," 2025. [Online]. Available: https://graph-tool.skewed.de/. [Accessed Jul 2025].

[18] M. L. Fredman and R. E. Tarjan, "Fibonacci Heaps And Their Uses In Improved Network Optimization Algorithms," in *25th Annual Symposium on Foundations of Computer Science*, Singer Island, 1984.

[19] Figma, "Figma," 2025. [Online]. Available: https://www.figma.com/. [Accessed Aug 2025].

[20] Python-Markdown, "Python-Markdown — Python-Markdown 3.8.2 documentation," 2025. [Online]. Available: https://python-markdown.github.io/. [Accessed Aug 2025].

[21] K. Reitz, "requests · PyPI," 2025. [Online]. Available: https://pypi.org/project/requests/. [Accessed Aug 2025].

[22] UltraJSON, "Ultra fast JSON decoder and encoder written in C with Python bindings," 2025. [Online]. Available: https://github.com/ultrajson/ultrajson. [Accessed Aug 2025].

[23] Pallets, "Welcome to Flask — Flask Documentation (3.1.x)," 2025. [Online]. Available: https://flask.palletsprojects.com/en/stable/. [Accessed Aug 2025].

[24] International Organization for Standardization, "ISO 9241-11:2018," 2018. [Online]. Available: https://www.iso.org/standard/63500.html. [Accessed Aug 2025].

[25] L. K. Mortensen, H. R. Shaker and C. T. Veje, "Relative fault vulnerability prediction for energy distribution networks," *Applied Energy*, vol. 322, p. 119449, Jun 2022.