Evan Magee & Jomiloju Odumosu
Professor Jang
CmpSc 463 - 001
12/2/2025

Description

This project aims to create a tool to help with finding the best paths for evacuating or finding shelter during certain storm events. This project does so through the usage of graphs which represent certain junctions, locations, or buildings on the Abington campus. The main graph is a standard weighted graph which uses the distance between the junctions in feet as the edge weight. There also exists a graph for the canvas to display locations and elevations of each location. The program also simulates different weather events such as an extreme snowstorm or blizzard, and an extreme storm situation. The application also contains a small user interface and allows for mouse/touch interaction for selecting current locations.

Significance of the Project

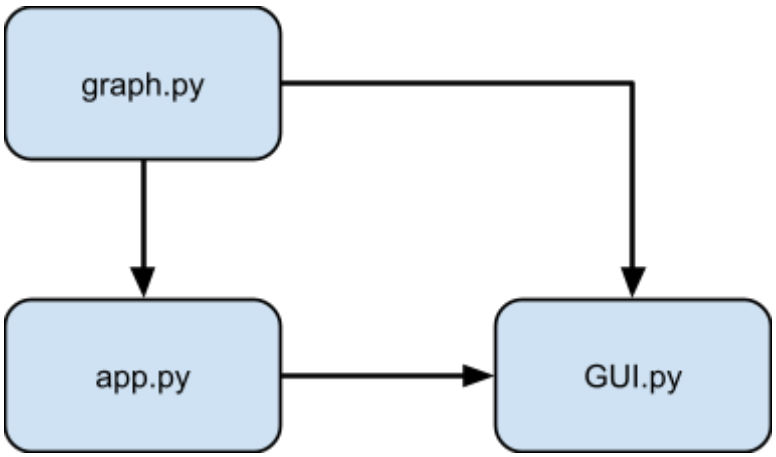
The project is meaningful because it addresses safety and decision-making in emergency situations. Users can simulate flooding, snowstorms, and fire hazards, observing how paths become blocked and which routes remain viable. Its novelty lies in combining interactive visualization with real-time algorithmic adjustments. By integrating user input, pathfinding, and hazard simulation, the tool demonstrates practical applications of graph algorithms in safety planning and urban design. This approach provides insights that extend beyond classroom exercises into real-world emergency preparedness scenarios.

Code Structure

The implementation of this program was done using a one backend file, one frontend file, and a graph storage file. The backend file, app.py, runs most of the path creation functions. The frontend file, GUI.py, runs all other functions of the program as well as the main GUI of the program. The graph file, graph.py, stores all the map information like node edges and the map locations.
The overall code structure is as follows:

1. Helper Functions
    - This section contains functions for utilities like selecting nodes, updating labels, changing colors, and blocking/reopening edges.
2. Stored Graphs
    - Abington Map
        - Contains nodes and their neighbors for the Abington campus

    - Abington Locations
        - Contains x,y locations for where to display nodes on the GUI canvas
    - Abington Elevations
        - Contains the elevation of the nodes in feet
    - Woodland Map
        - Contains nodes and their neighbors for the Woodland building
    - Woodland Locations
        - Contains x,y locations for where to display nodes on the GUI canvas
    - Woodland Elevations
        - Contains the elevation of the nodes in feet
3. Graph Functions
    - shortest_path()
    - k_shortest_paths()
    - closest_building_path()
    - closest_exit_path()
4. GUI Functions
    - User interactions
    - Menu updates
    - Drawing paths
    - Color updates
    - Weather simulations
    - Path closing and reopening
5. Main Method
    - Runs GUI, draws maps, loads graphs, connects callbacks, and waits for user interaction.



Algorithms

- app.py
    - shortest_path(graph, start, end)
        - O((n + e) log n)
        - This function uses Djiskras algorithm to find the shortest path from the given starting node to the given ending node for the input graph. This is done using the weights or distances between each edge.
    - k_shortest_paths(graph, start, end, k=5)
        - O((k * n) * log(k * n)
        - This function finds up to k shortest simple paths from start to end using a heap-based BFS. Avoids revisiting nodes in a single path.
    - closest_building_path(graph, start)
        - O(n)
        - This function runs shortest_path from the given start location to each of the designated building nodes (as set in graph.py) and stores each result into an array. The array of results is then sorted and the first provided path is considered the shortest path to the nearest building, that path is returned.

    - closest_exit_path(graph, start)
        - O(n)
        - This function runs shortest_path from the given start location to each of the exits in the Woodland building (as set in graph.py) and stores each result into an array. The array of results is then sorted and the first provided path is considered the shortest path to the nearest exit, that path is returned.

- graph.py
    - *Contains No Algorithms*

- GUI.py
    - swap_screen()
        - O(n)
        - This function clears the UI and resets all dictionaries

    - set_destination(d)
        - O(1)
        - This function is given an input d and then sets the destination global variable to d.

- set_current_node(n)
  - O(1)
  - This function is given an input n and then sets the current node global variable to n.

- set_location_text(node)
  - O(1)
  - This function, when given a node, verifies that the node is on the graph, then sets location text in the GUI (canvas) to show the name of the given node.

- set_path_name_id(id)
  - O(1)
  - This function, when given an id, sets the path id text in the GUI (canvas) to the given id.

- set_path_length(length)
  - O(1)
  - This function, when given a length, sets the path length text in the GUI (canvas) to the given length.

- set_node_color(node, color)
  - O(1)
  - This function inputs a node and a given color string or hex code. This node is then verified and gets its color set to the given color. The node is then raised to make sure it is displayed on top of other canvas elements.

- set_node_active_color(node, color)
  - O(1)
  - This function inputs a node and a given color string or hex code. This node is then verified and gets its active color (the color it's given when it is clicked on) set to the given color. The node is then raised to make sure it is displayed on top of other canvas elements.

- update_neighbors_menu()
  - O(n)
  - This function resets the neighbor selection menu for when a new node is selected. It makes sure that only the neighbors of the

currently selected node are shown. If node has not neighbors, it will instead display “no neighbors”

- reset_node_colors()
  - O(n)
  - This function resets all nodes to their proper color depending on what is occurring with said nodes.

- clear_path_visuals()
  - O(n)
  - This function loops through all items in the path_items array and deletes these items from the canvas object. The path_items array is then set to empty and reset_node_colors() is called. This clears all the path visuals and resets the map visuals to the default state.

- reopen_paths()
  - O(n)
  - This function sets the current graph to a copy of the default graph and deletes all line objects in the blocked_items global array. The blocked edges and items arrays are then cleared and the clear_path_visuals() function is called to return the map to its default state.

- update_isolated_nodes()
  - O(n)
  - This function loops through all nodes on the Abington Map and checks to see if it has 0 edges or is isolated due to closed paths from weather or other events. The node is determined to be isolated, it is set to yellow to indicate so and to match the closed path coloring.

- get_closest_exit()
  - O(n)
  - This function calls the closest_exit_path function and displays the returned path on the map.

- get_closest_path()
  - O(n)
  - This function calls the closest_building_path function and displays the returned path on the map.

- get_k_paths(k=10)
  - O(k * n)
  - This function computes up to k shortest paths from the current location to a selected destination, it will then filter out the paths that include blocked edges. It then displays the first path onto the map.\

- display_path()
  - O(n)
  - This function draws a path from k_paths on the canvas map, it skips blocked edges and updates path info.

- close_path(u, v)
  - O(n)
  - This function blocks an edge between 2 given nodes (u & v) and updates the graph and visuals to remove this edge.

- close_selected_path()
  - O(n)
  - This function closes the edge from the current node to the selected neighbor of that node. It updates the neighbor menu as well for manual path closing.

- on_click(event)
  - O(n)
  - This function handles canvas clicks to select a node, updates neighbors, and highlights the selected node.

- forFlooding(threshold)
  - O(n)
  - This function automatically closes all paths where either node is below the flood elevation threshold input by user or by default threshold.

- forSnowStorm(threshold)
  - O(n)
  - This function automatically closes all paths where the incline between nodes is too steep for snowstorm safety by the input threshold or by the default threshold.

- forFireScenario()
  - O(n)
  - This function simulates a fire by closing a few specific edges as if there were a fire in their location. This is for the Woodland building map.

- clear_screen()
  - O(n)
  - This function takes the display window and iterates through all of its widgets or elements and destroys said elements. This clears the screen or produces a blank screen. It is used for mainly switching windows (this program does not do that) and ensuring the window is blank for when the program is run.

- main_screen()
  - O(n)
  - Initializes the Abington map GUI by loading the map image, creating nodes & labels, setting up buttons/menus, and binding click events.

- woodland_building()
  - O(n)
  - Initializes the Abington map GUI by loading the map image, creating nodes & labels, setting up buttons/menus, and binding click events.

Verification of Algorithms

To verify that the algorithms used in the evacuation tool were implemented correctly, several forms of testing and validation were conducted. Because the core logic of the system relies on graph-based pathfinding and dynamic edge manipulation, each algorithm was evaluated using controlled inputs, known outputs, and stress-case scenarios.

1. Verification of shortest_path()

The shortest_path() function implements Dijkstra's Algorithm. To verify correctness:

- Known Graph Testing:
  The algorithm was tested on small, manually constructed graphs where the true shortest path was already known. The function correctly returned both the

expected path and distance in all cases.

- Edge-case Testing:
    - Start and end nodes being the same
    - A path where multiple shortest paths have equal weight
    - Nodes connected by only one available route
      In all scenarios, the algorithm produced correct and stable results.
- Blocked Path Testing:
  Paths were intentionally removed using the GUI's block/unblock feature. The algorithm successfully avoided removed edges and recalculated the new shortest available route.

2. Verification of k_shortest_paths()

The k-shortest paths algorithm was tested by:

- Running it on graphs with multiple route alternatives
- Ensuring that:
    - Paths were returned in strictly increasing order of total cost
    - No duplicate paths appeared
    - All paths were valid and traversable
- When k exceeded the total number of possible alternative paths, the function returned only the valid existing paths without errors.

3. Verification of closest_building_path()

This algorithm loops through the list of designated shelter buildings, runs shortest_path() to each, and returns the minimum. Verification involved:

- Creating cases where the closest building by distance was not the closest by route length (because a long edge weight made a farther building closer in path

cost)

- Blocking and reopening paths to confirm the function dynamically adapted
- Ensuring sorting correctly identified the shortest valid resulting path

4. Verification of GUI-related algorithms

Functions such as close_path(), update_isolated_nodes(), display_path(), and node color-setting functions were verified through:

- Visual inspection of the canvas during interaction
- Manually blocking specific edges and confirming that:
    - The edge visually changed color
    - The graph data structure updated correctly
    - Isolated nodes were highlighted and recognized by the system
- Clicking through multiple nodes to confirm correct active/normal color transitions
- Ensuring all visual path items were cleared properly by clear_path_visuals()

All algorithmic components performed consistently and produced correct outputs when compared to expected results.

Functionalities

The evacuation tool provides a complete interactive system that combines graph algorithms, weather simulation, and an intuitive GUI. The major functionalities are:

1. Interactive Campus Map

- Displays all nodes (junctions, buildings, or locations) on the Abington campus
- Allows the user to select any node by mouse or touchscreen

- Highlights selected nodes and updates labels such as:
    - Current location
    - Destination
    - Path ID
    - Path length

2. Shortest Path and Routing

- Computes the shortest path between any two nodes using Dijkstra's algorithm
- Supports calculation of the k shortest paths, giving multiple route alternatives
- Computes the closest shelter building based on path distance rather than geographic distance

3. Path Visualization

- Draws the computed path directly on the canvas
- Highlights all nodes and edges along the chosen path
- Displays path metadata such as total distance
- Allows users to clear the visual path display at any time

4. Dynamic Path Blocking (Simulation of Storm Conditions)

Paths can be blocked manually or automatically:

- Manual Block/Unblock:
  Clicking edges allows the user to close or reopen paths.
- Weather Simulations:

    - Flooding Mode: Edges below a defined elevation threshold are closed
    - Snowstorm Mode: Edges above or below certain conditions are removed
- The graph updates immediately and recalculates available routes

5. Node Isolation Detection

- The program identifies nodes that become isolated due to storms or manual closures
- Isolated nodes are visually marked (e.g., turned yellow)
- Ensures the user is aware when a location cannot reach any building or safe path

6. Graph Reset and Recovery

- `reopen_paths()` fully restores the graph to default
- All blocked items are removed from the canvas
- Path visuals and color changes are cleared

7. Clean Window Control

- Full screen clearing with `clear_screen()`
- Ensures that UI resets correctly when restarting or switching views

Execution Results

- Abington Map

1. Find Closest Building
    - Select (click) node/location on map

- Press “Find Closest Building” Button

- Path to closest building is displayed

2. Find Path to Building

- Select (click) node/location on map

- Select a building from the “Choose Location” drop down

- Click “Find Path to Building” Button

- Shortest path to the selected building is displayed


Abington Evacuation Tool
Location: Z
Path ID: 1
Length: 864ft
Springhouse
Woodland Building
Lares Building
Lion Shrine
Rydal Building
Sutherland Building
Sutherland Auditorium
Tennis Court
Athletic Building
Cloverly Building
Find Closest Building
Reopen Paths
Flooding Scenario
Find Path to Building
Snowstorm Scenario
Woodland Building
Next Path
Close Path
Switch Map

3. Next Path

- With a path already displayed or shown, press the “Next Path” button.

- The next shortest path or an alternative path will be displayed


Abington Evacuation Tool
Location: Z
Path ID: 2
Length: 940ft
Springhouse
Woodland Building
Lares Building
Lion Shrine
Rydal Building
Sutherland Building
Sutherland Auditorium
Tennis Court
Athletic Building
Cloverly Building
Find Closest Building
Reopen Paths
Flooding Scenario
Find Path to Building
Snowstorm Scenario
Woodland Building
Next Path
Close Path
Switch Map

4. Close Path

- Select (click) a node on the map and select one of its neighbors in the blank/neighbor drop down (lower left of window).

- Press the “Close Path” button

- The closed path is then displayed with a hazard line on the map

5. Reopen Paths

- With a lot of closed paths and/or routes on screen, press the “Reopen Paths” button.

- All paths and routes are then cleared on the map.

6. Flooding Scenario

- Press the “Flooding Scenario” button to see how flooding may close paths on the Abington Campus

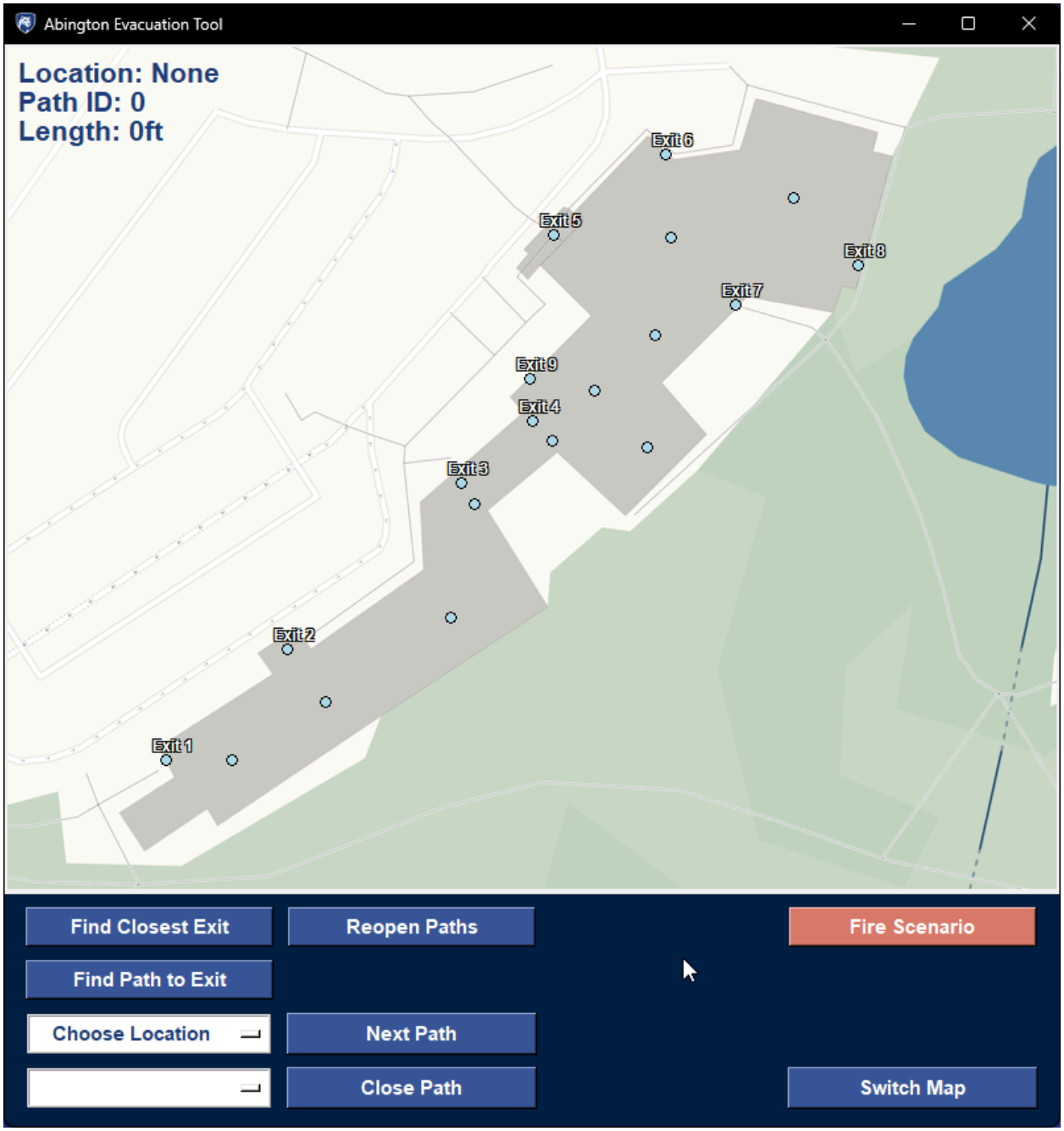
7. Snowstorm Scenario

- Press the “Snowstorm Scenario” button to see how a snowstorm may close paths on the Abington Campus.

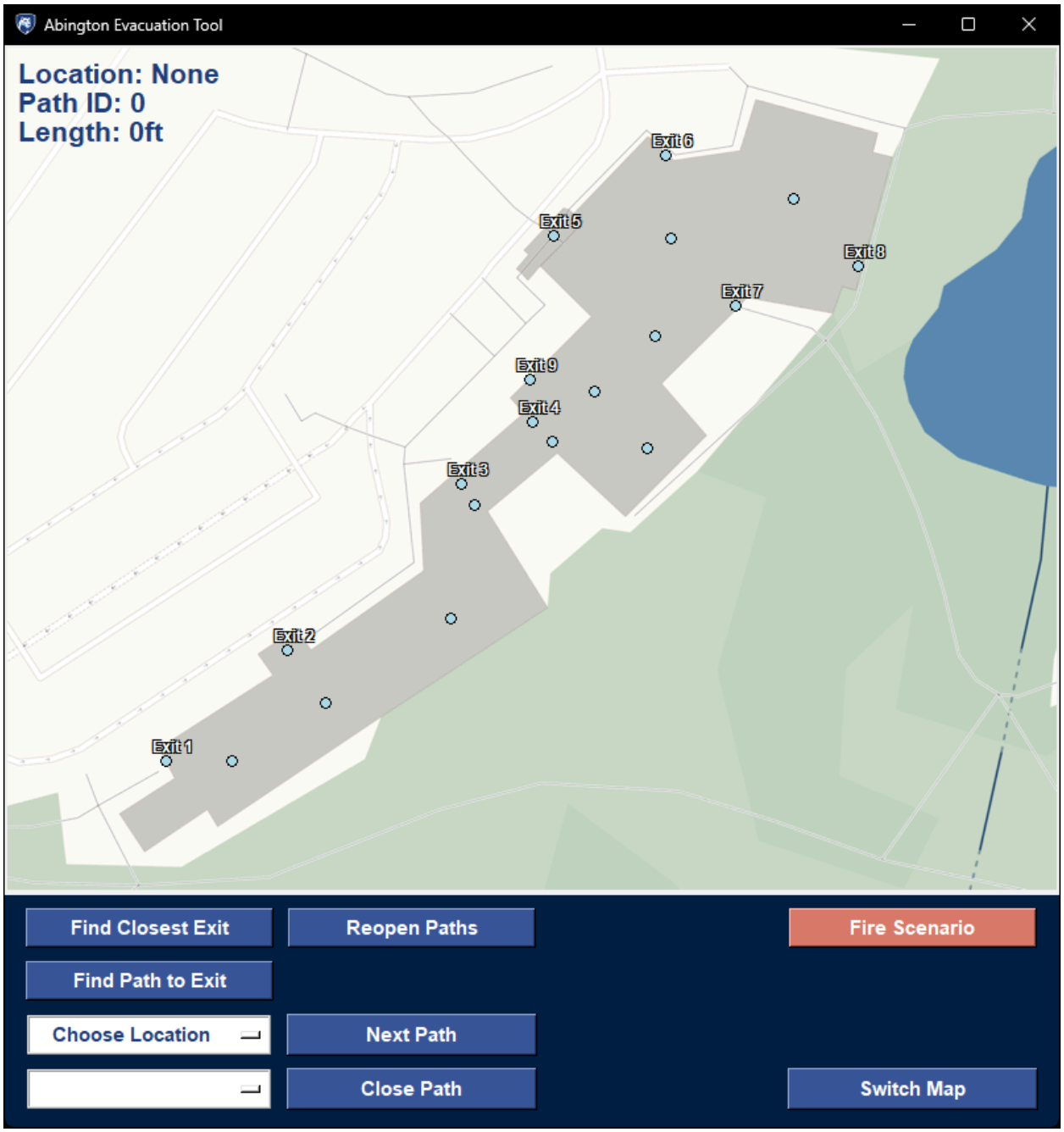
8. Switch Map

- Press the “Switch Map” button to switch from the Abington campus map to the Woodland Building map.

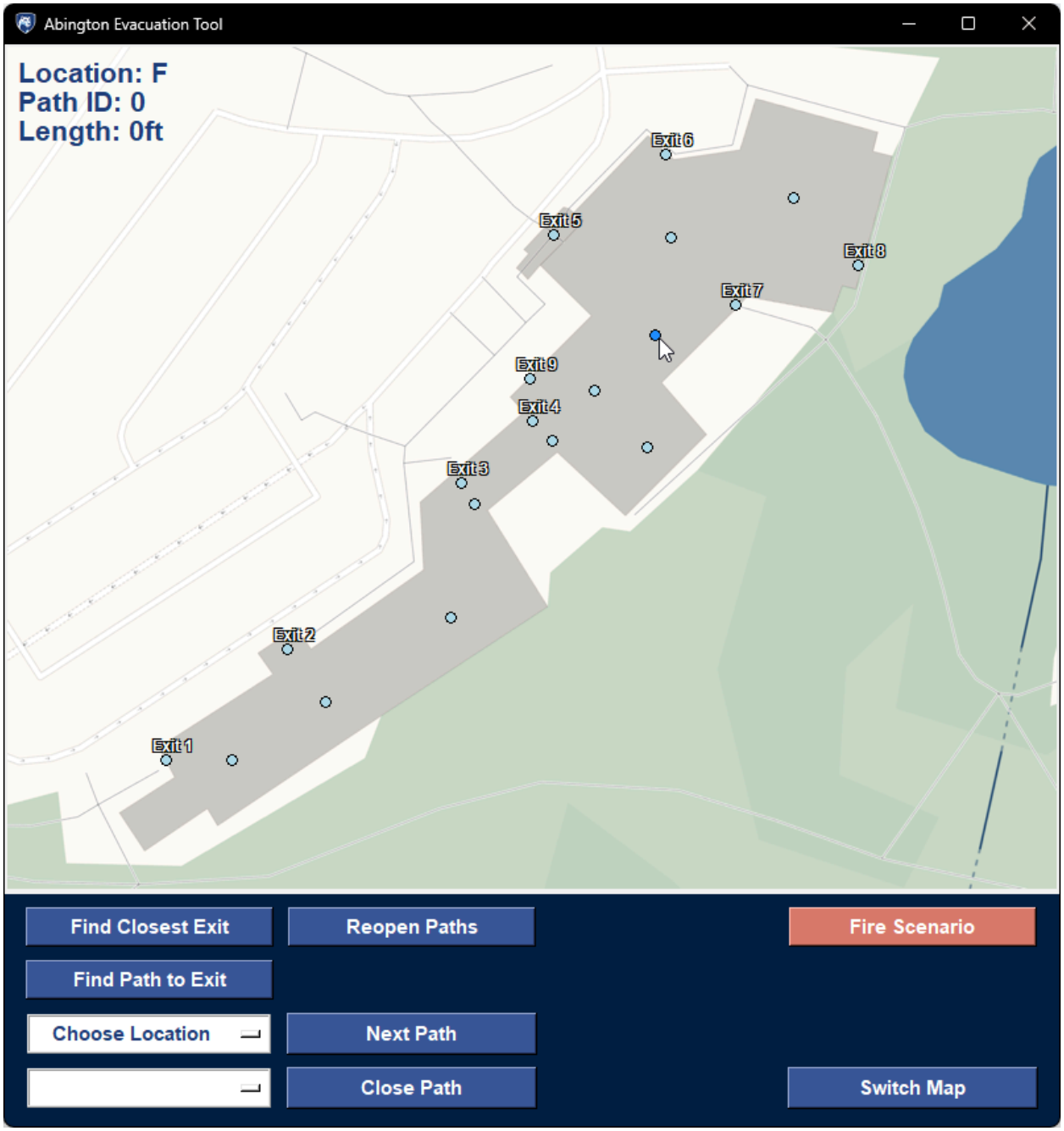
- The Woodland building map is then displayed with its according buttons and interface.



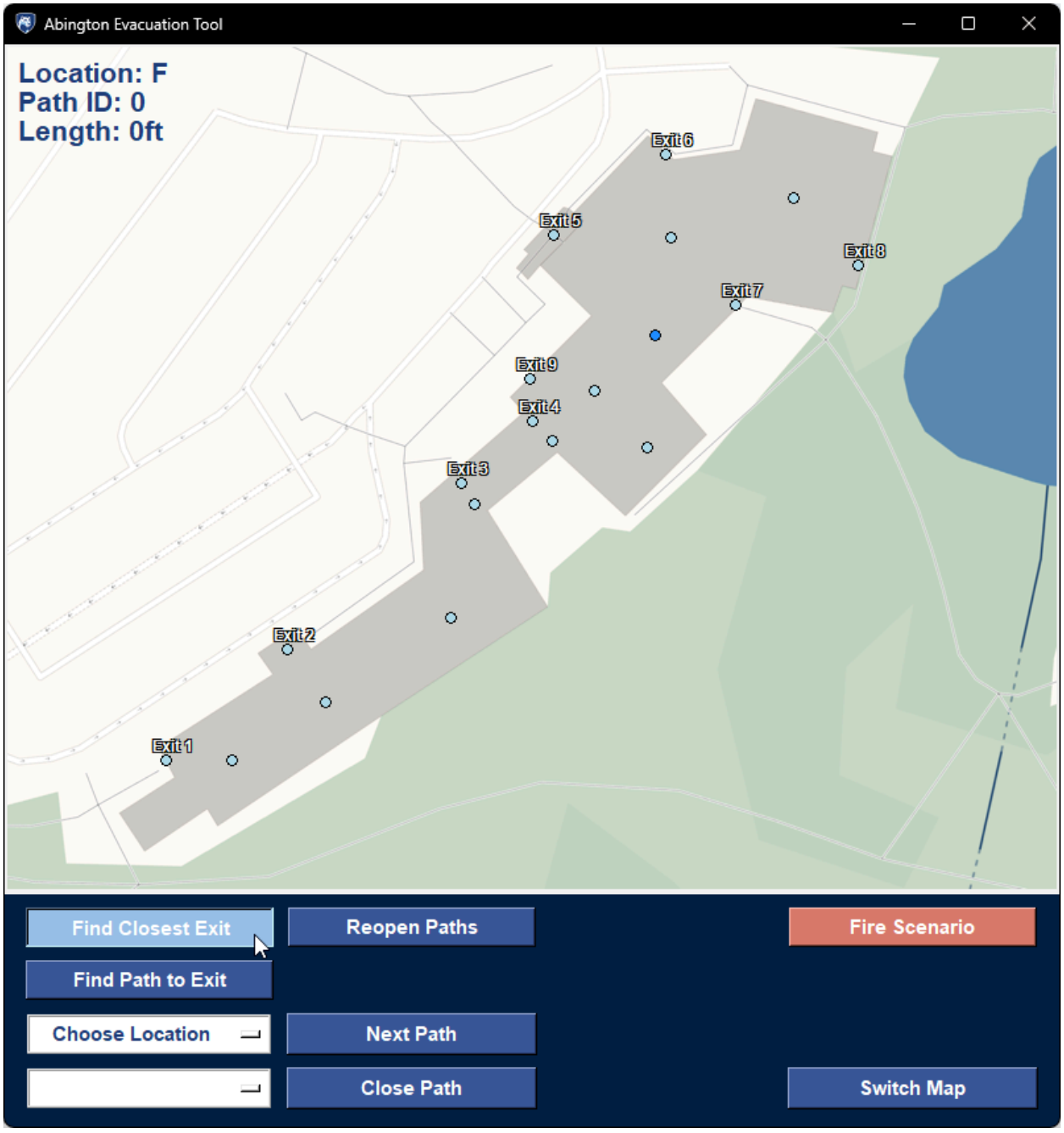
- Woodland Building Map



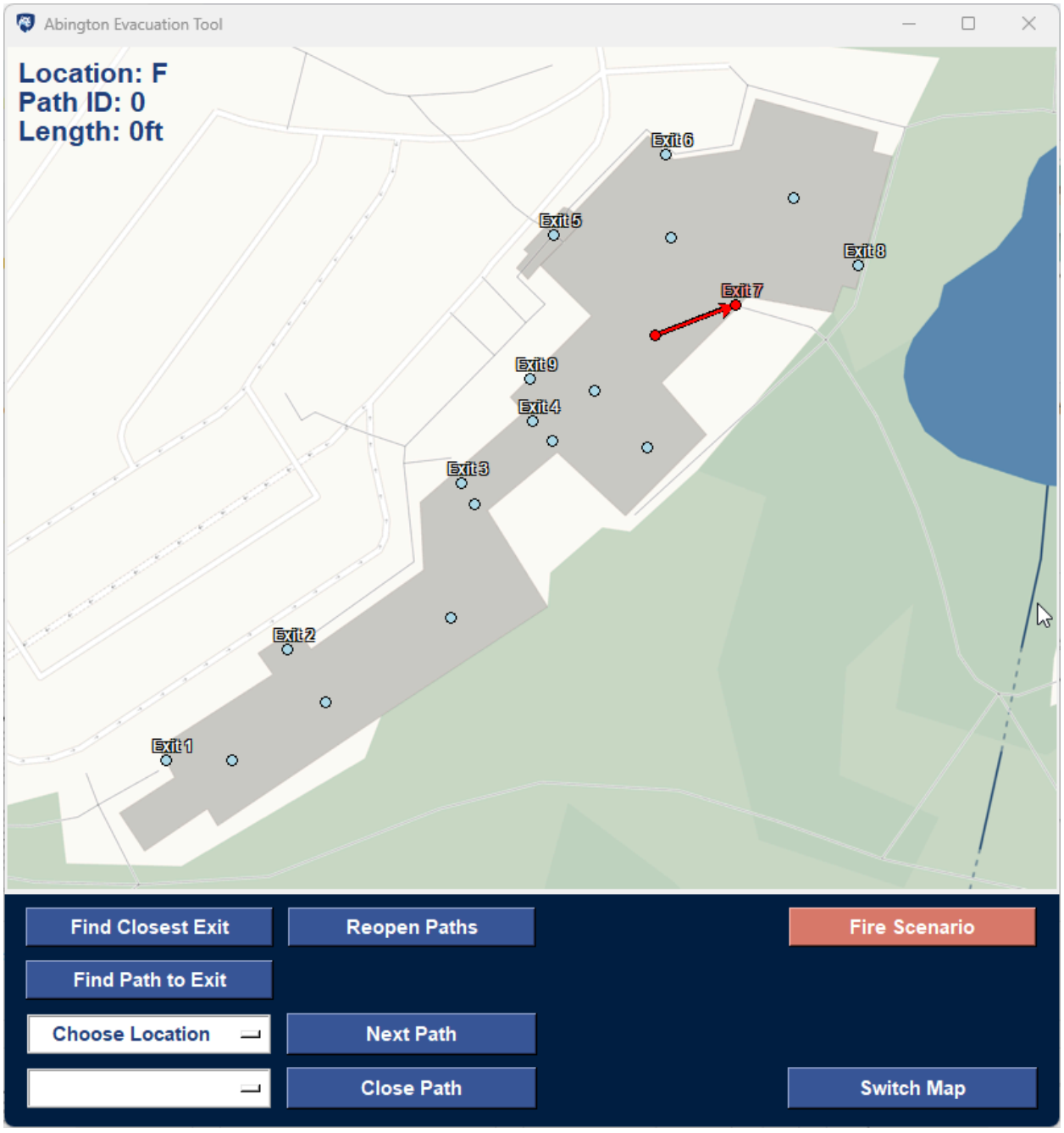
1. Find Closest Exit
    - Select (click) a node/location.



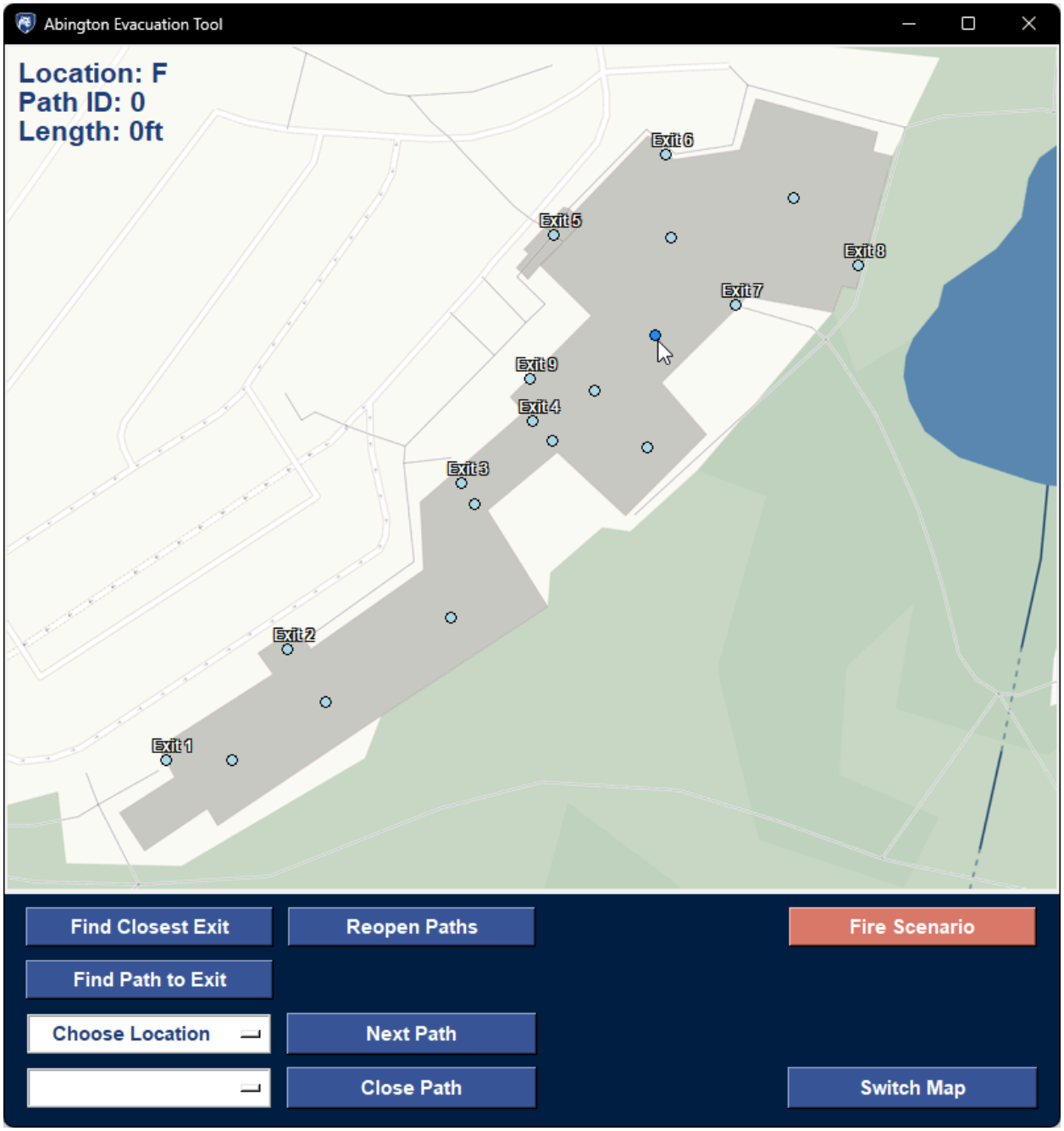
- Press the “Find Closest Exit” button



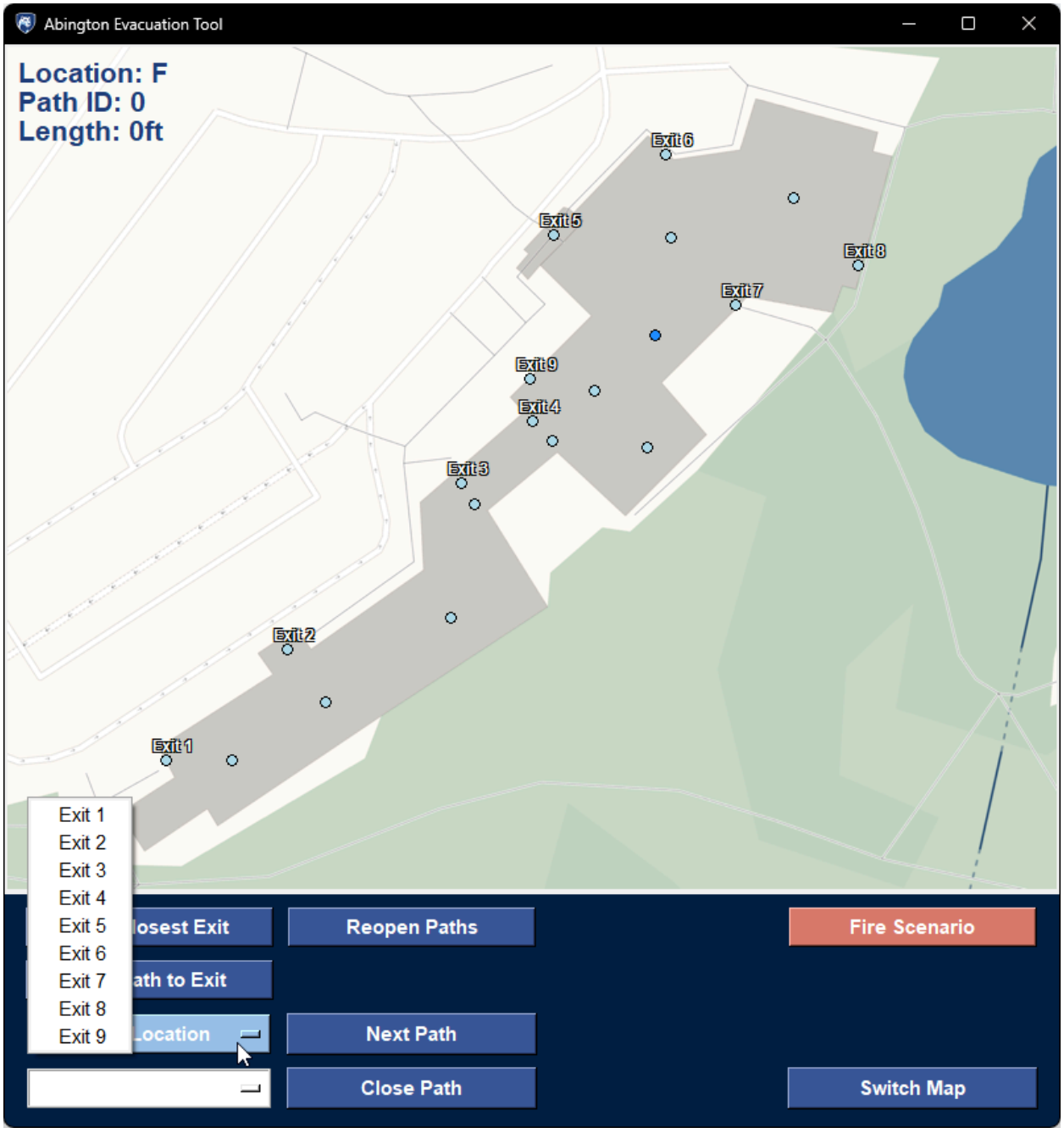
- The shortest path to the closest exit is displayed.



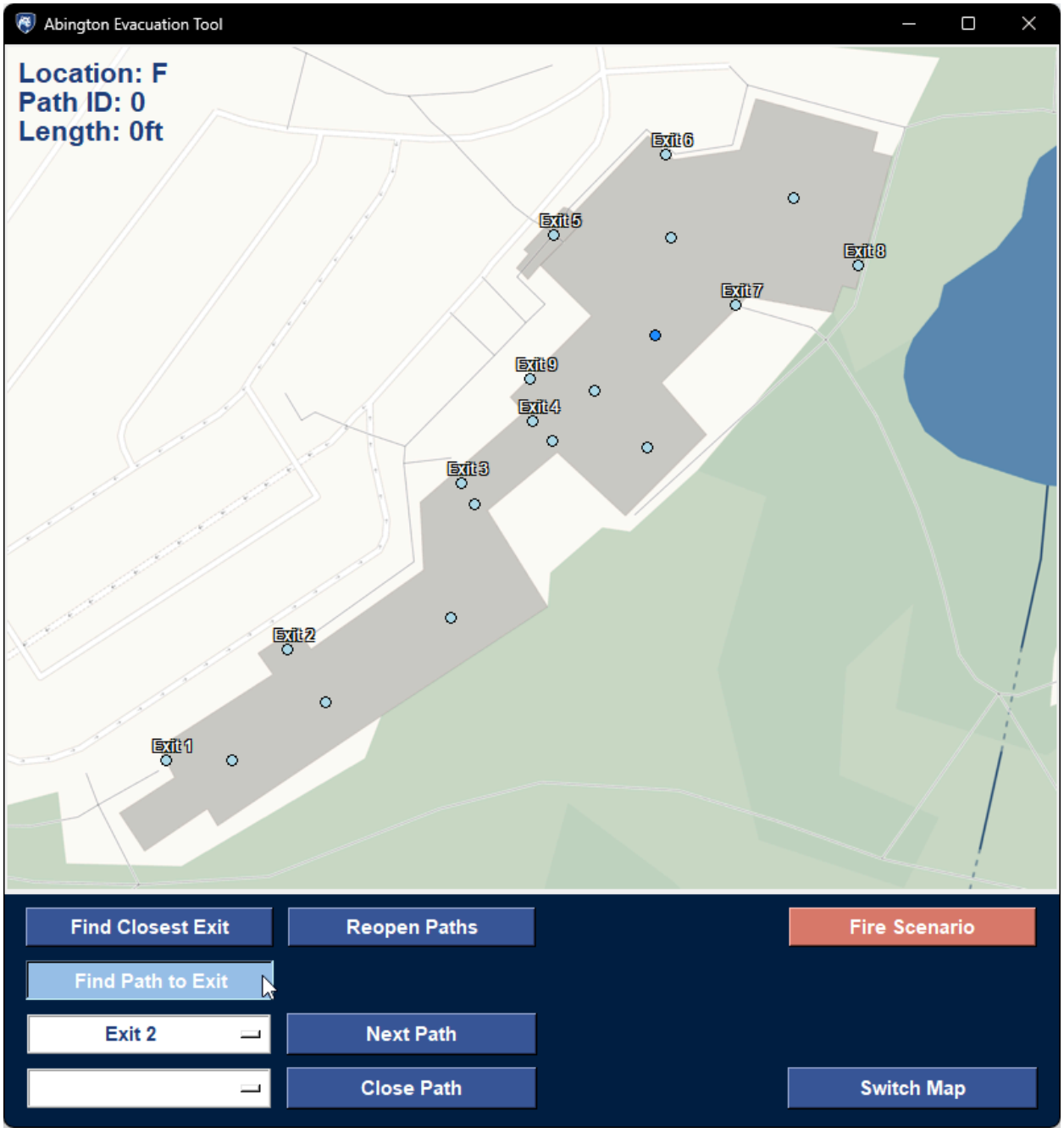
2. Find Path to Exit
    - Select (click) a node/location on the map



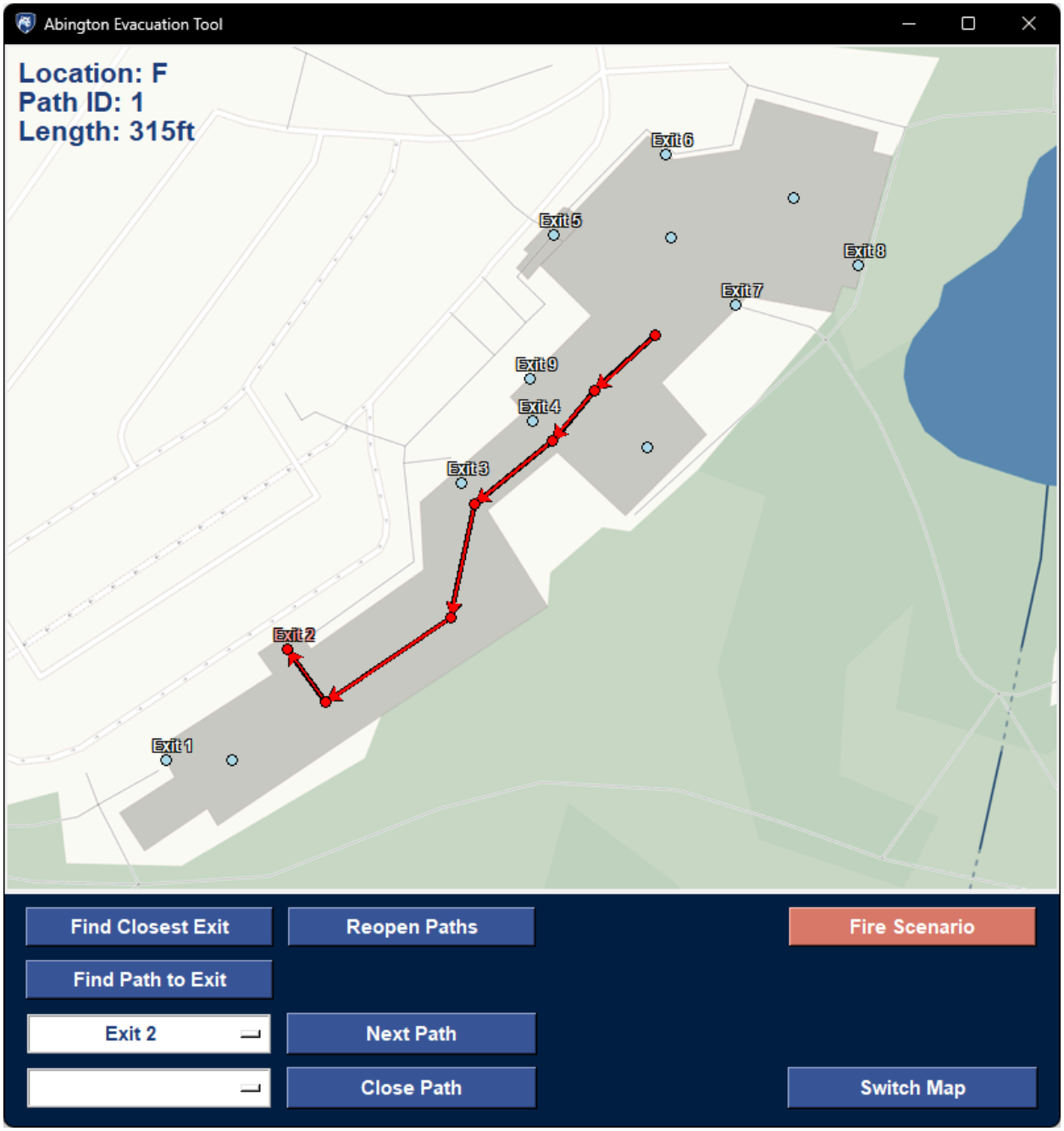
- Select an exit under the “Choose Location” drop down menu.



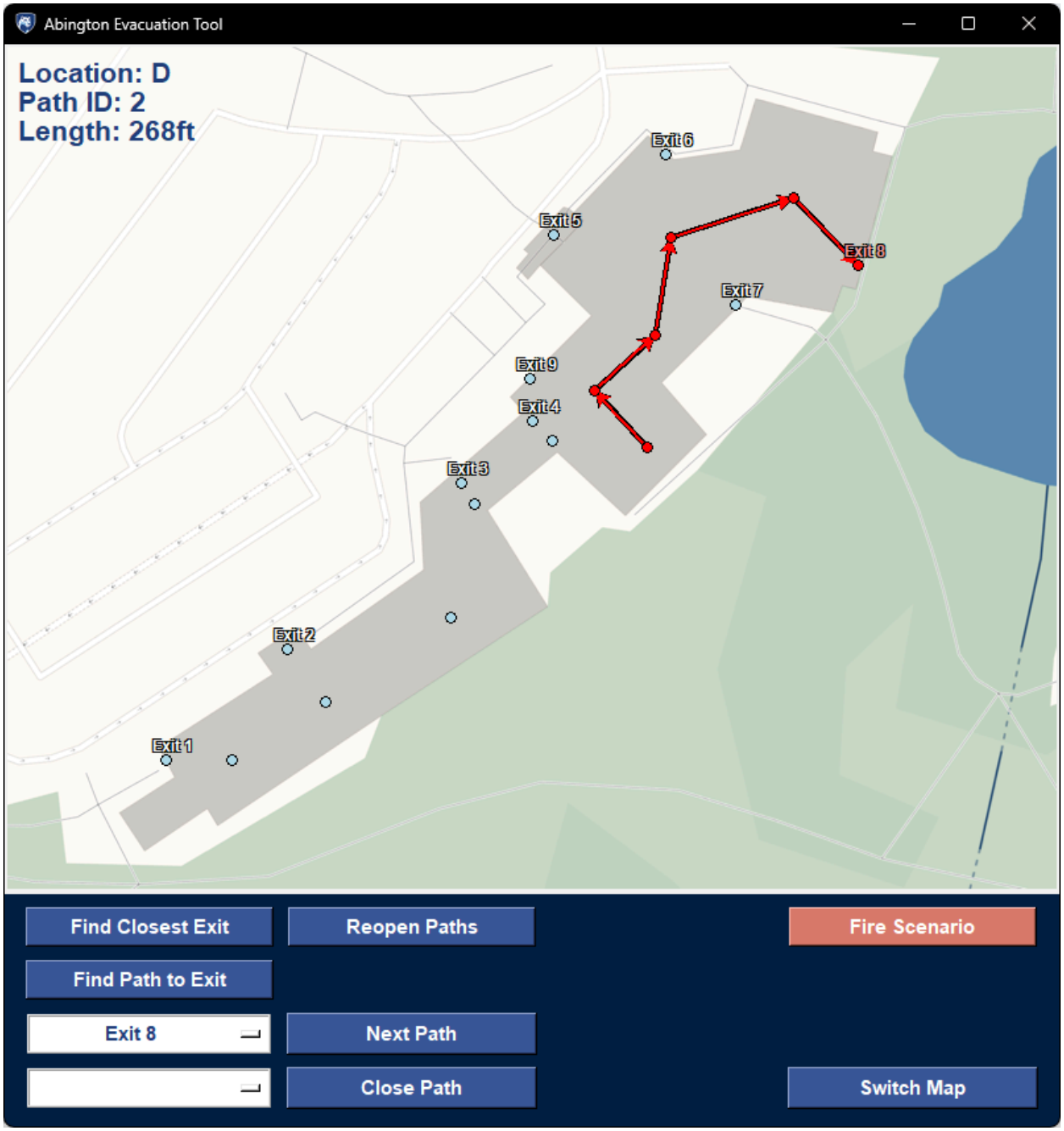
- Press the “Find Path to Exit” button.



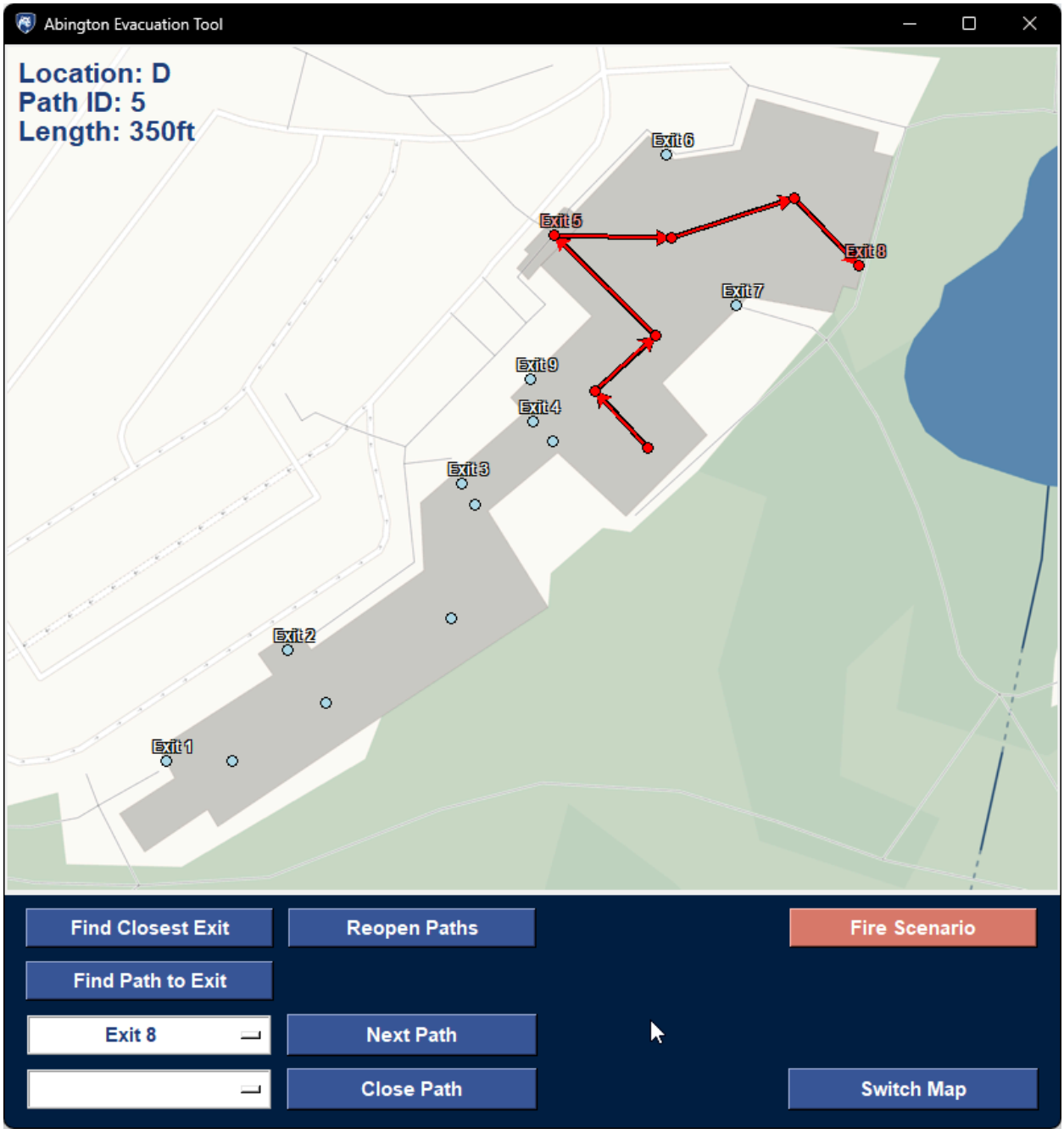
- The shortest path from the selected node to the selected exit is then displayed



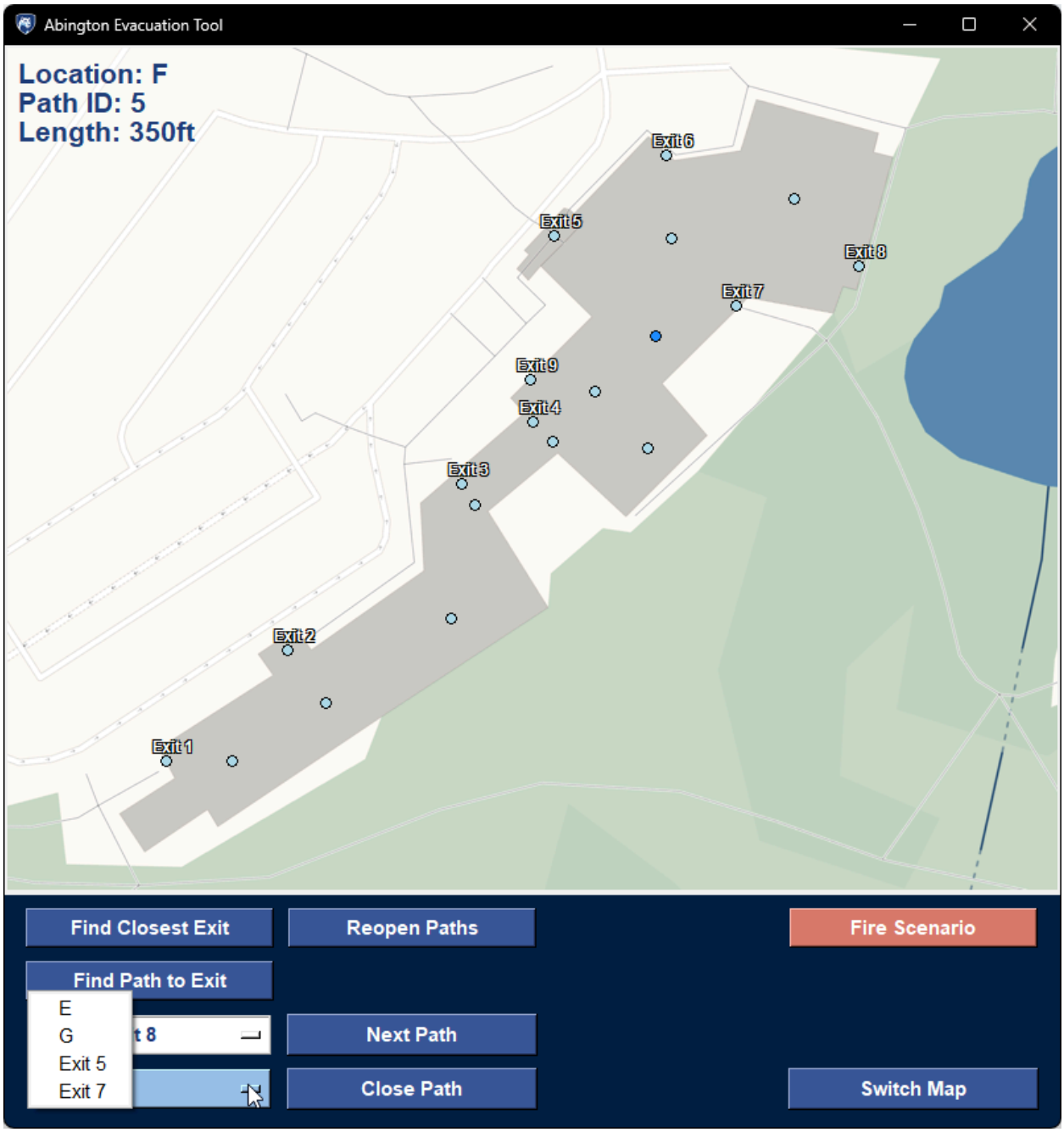
3. Next Path
    - Path already displayed on the map, press the “Next Path” button.



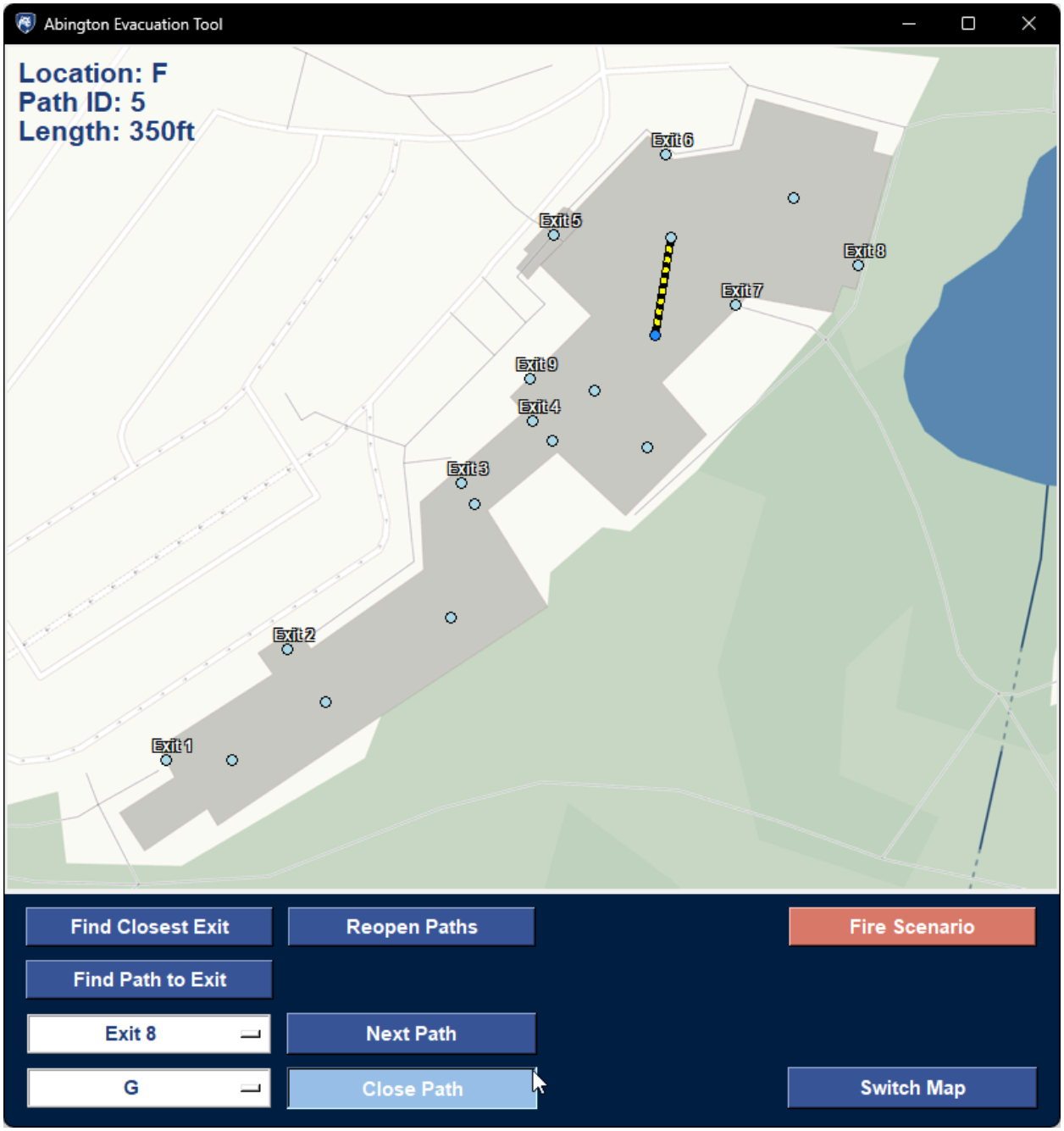
- An next shortest path is then displayed to that location



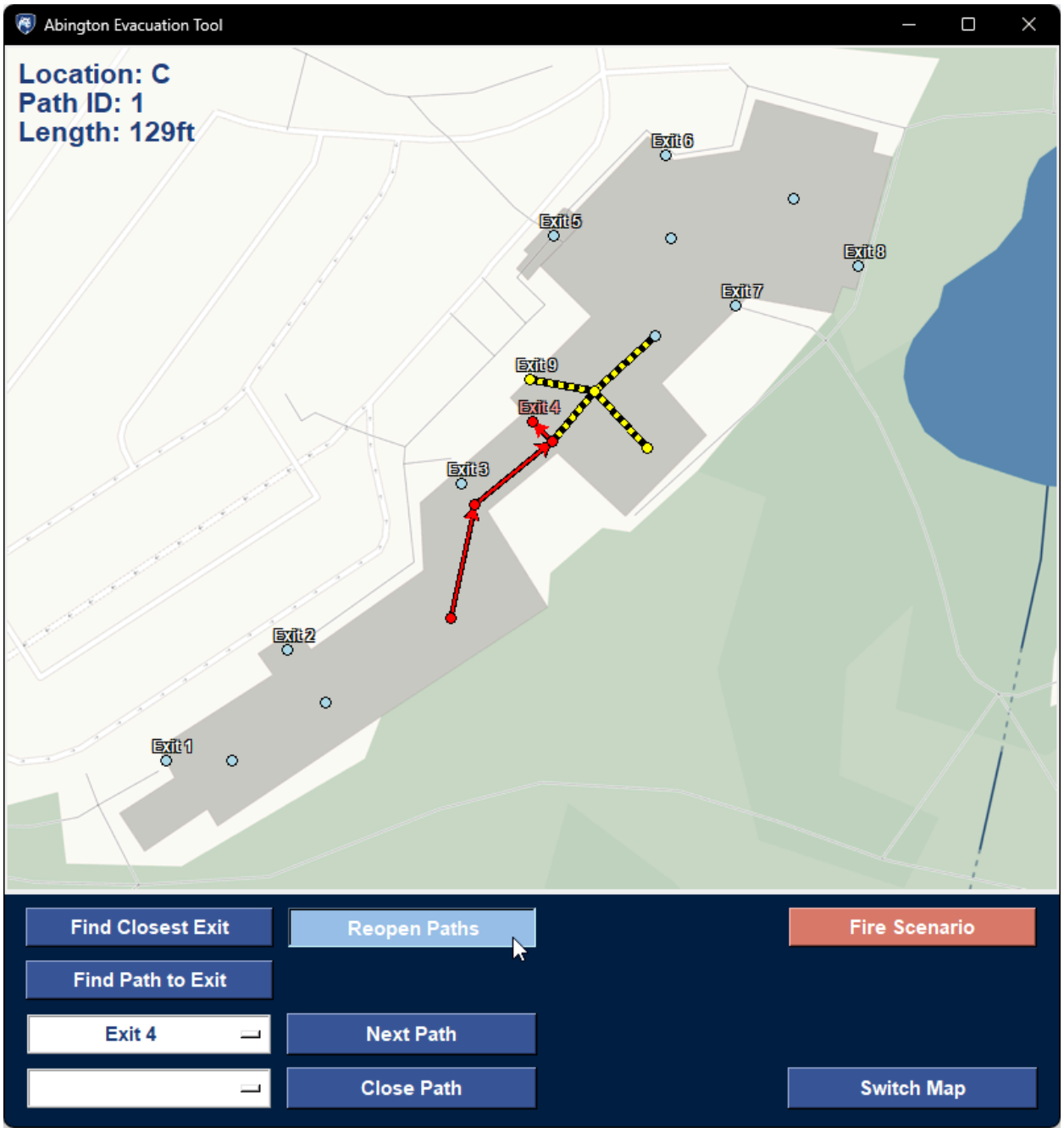
4. Close Path
    - Select (click) a node/location on the map and press the lower left neighbor drop down and select a neighboring node to that location.



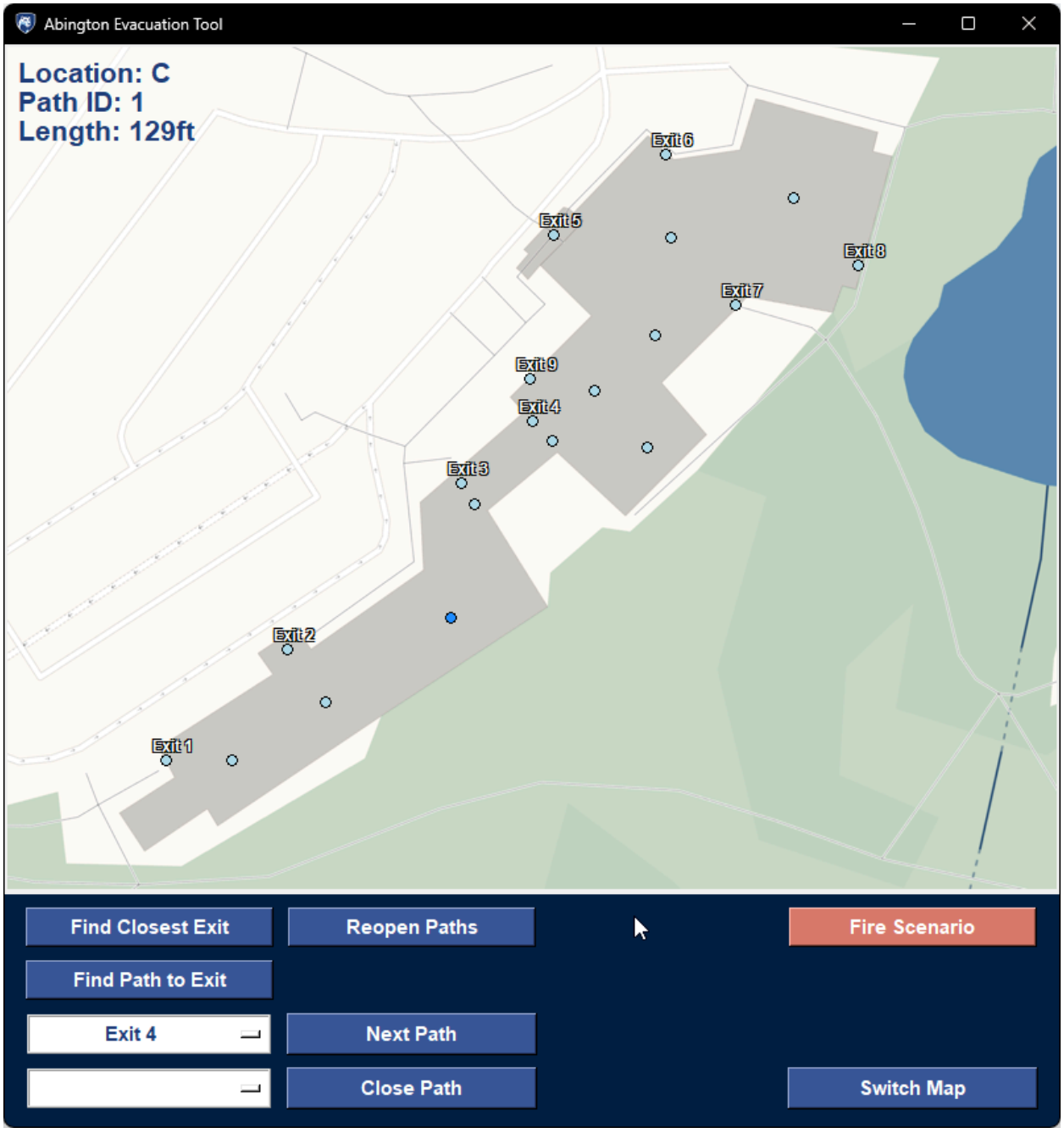
- Press the “Close Path” button and that path will be marked as closed on the map with a yellow hazard line.



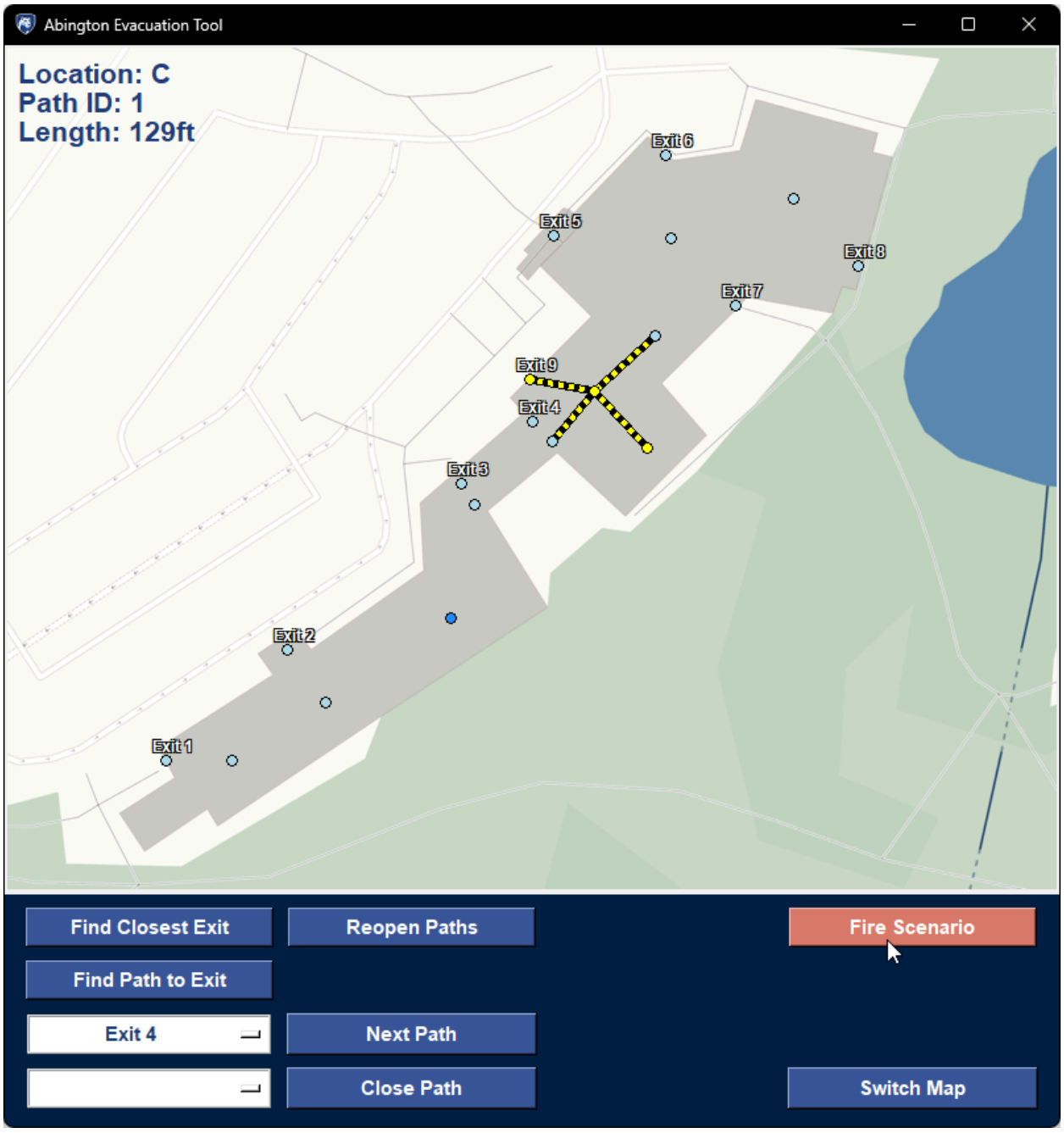
5. Reopen Paths
    - With a map that contains closed paths and a route, press the “Reopen Paths” button.



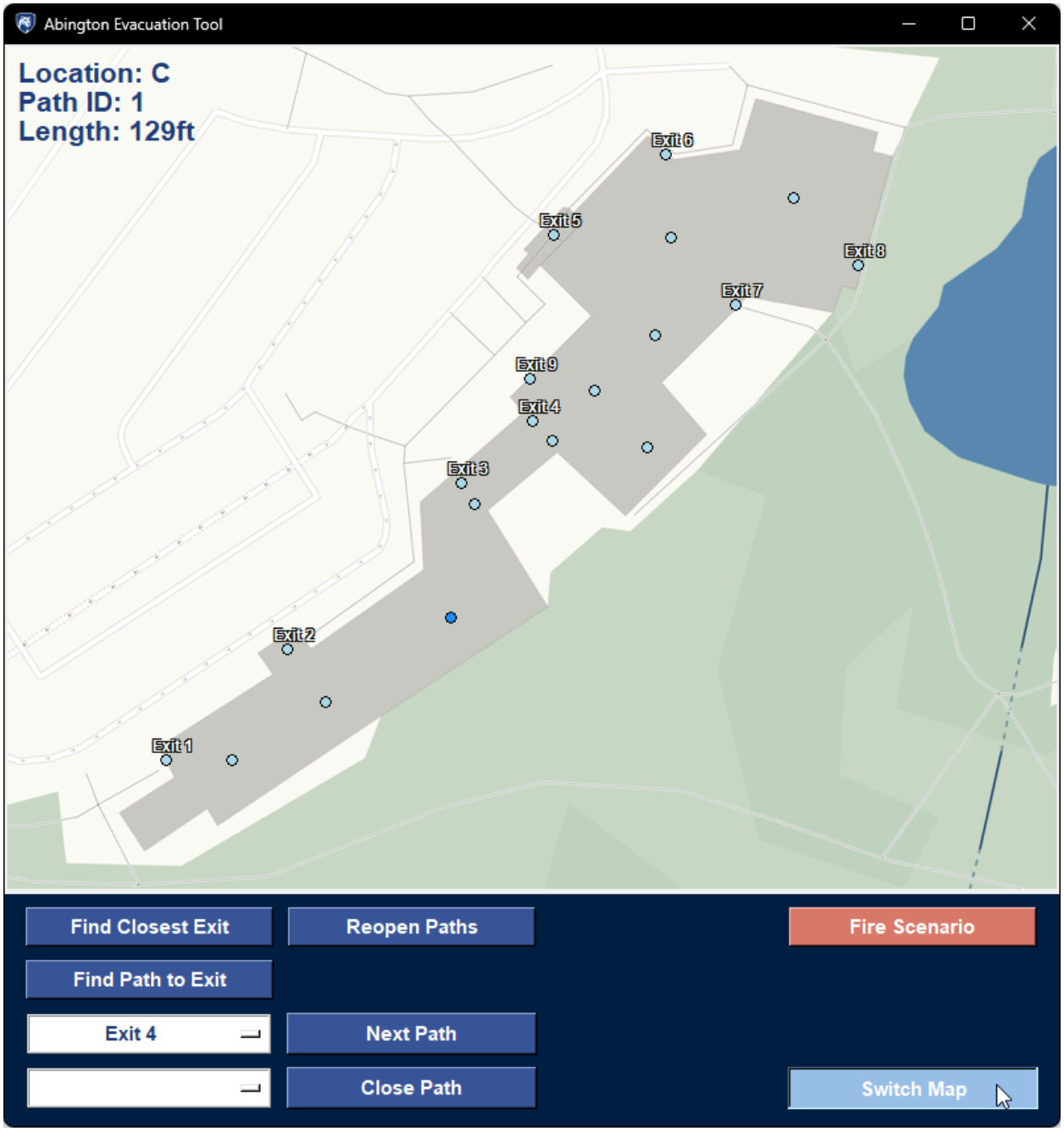
- All closed paths and routes are reopened on the map.



6. Fire Scenario
    - Press the “Fire Scenario” button to get path closures simulating a fire scenario in the Woodland building



7. Switch Map
    - Press the “Switch Map” button in the lower right corner.



- The map is then switched from the Woodland building map back to the Abington campus map and the according interface.

- Path finding with closed paths present
    - Path finding without closures

- Path finding with closures

Conclusions

This project successfully used dictionaries and graph strategies to correctly create an evacuation tool of Abington campus for extreme weather events and the Woodland building. Using Dijkstra's algorithm and an adapted version for finding alternative paths, the program is able to draw an evacuation route for users from a selected node. The initial route is the shortest route to a selected building, exit, or the closest of either marked by red arrows and nodes along the map. The program also allows for closures of certain paths or edges which can be used to represent scenarios such as fallen trees or flooding. It also includes present closures for flooding or snow as well as a fire scenario for the Woodland building. This program can be used to help make sure students get to safety in extreme weather situations and prevent injuries from confusion. Overall, this project came together rather smoothly and demonstrates one of the vast usages of graphs.