InstaMesh Graph Theory Case Study

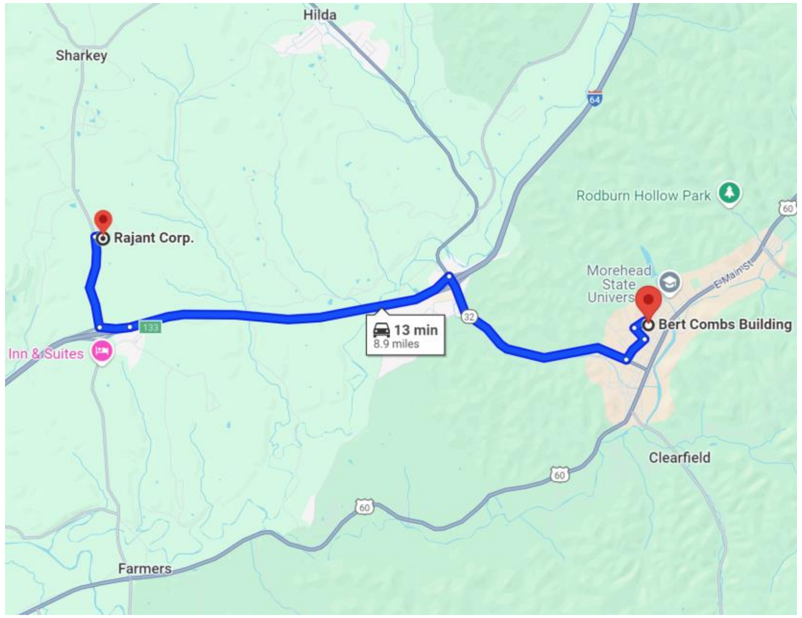
David Mayo, Software Engineer

Company Brief

- **Pioneer of Kinetic Mesh® wireless**
  BreadCrumb® wireless devices, the industry standard for reliable industrial mesh networking in the most challenging RF environments, using proprietary, patented InstaMesh® algorithm.

- **Based in the U.S., Rajant has successful deployments in over 80 countries**
  Almost all our manufacturing—and a lot of our engineering—is done in Morehead, Kentucky.

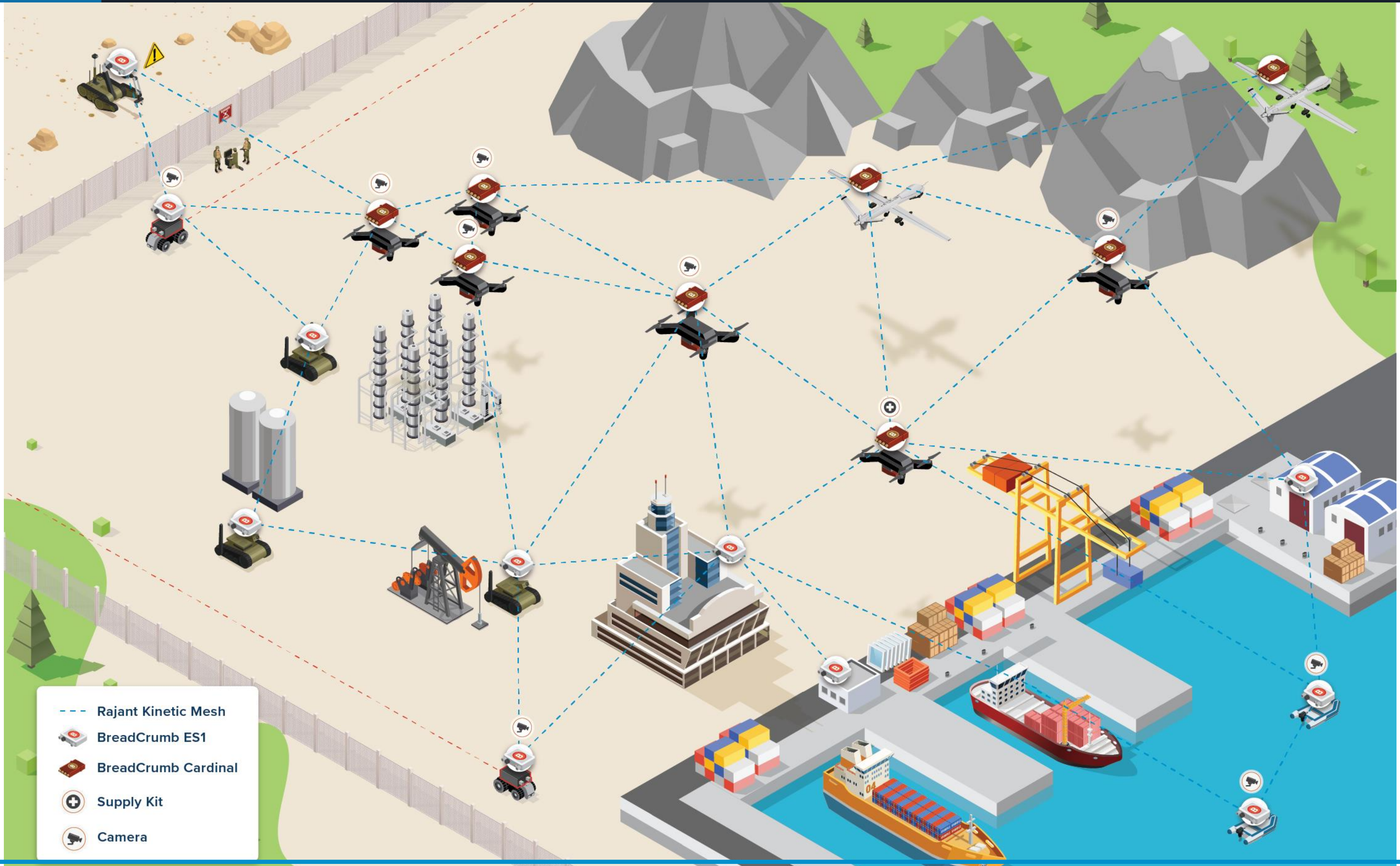
- **Powered by Morehead State University Graduates**
  ~30 MSU graduates on staff
  ~15 MSU engineering graduates on staff
  ~5 MSU computer science alumni on staff
  (including me!)



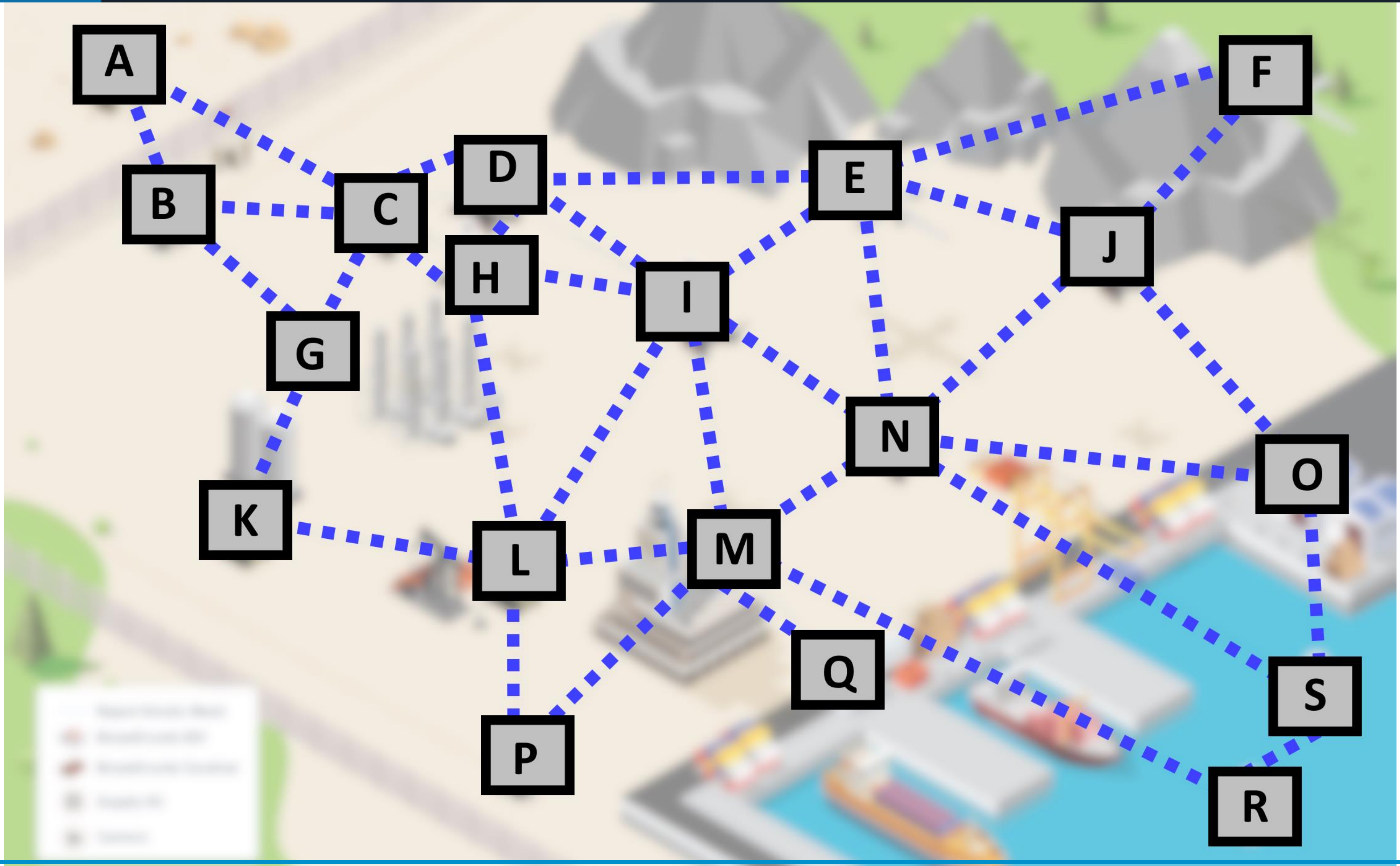
MOREHEAD STATE UNIVERSITY ™

Example Usage

Scenario


Rajant Kinetic Mesh
BreadCrumb ES1
BreadCrumb Cardinal
Supply Kit
Camera

Scenario



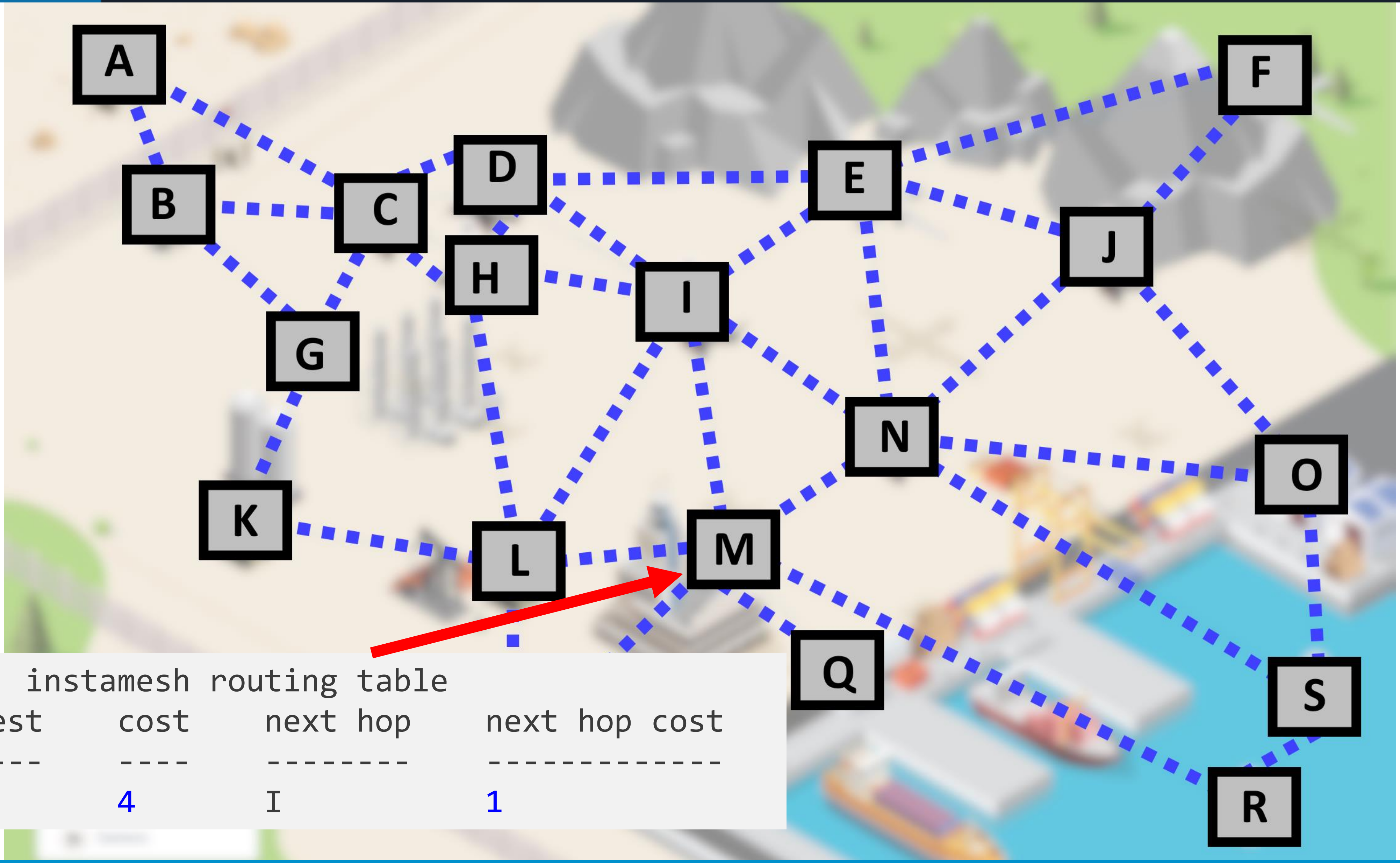
Log Files

```
#  breadcrumb serial number
M

#  instamesh neighbors
neighbor    cost
--------    ----
I           1
L           1
N           1
P           1
Q           1
R           1
```

```
#  instamesh routing table
Dest      cost      next hop      next hop cost
----      ----      --------      -------------
A         4         I             1
B         4         I             1
C         3         I             1
D         2         I             1
E         2         I             1
F         3         I             1
G         3         L             1
H         2         I             1
I         1         I             1
J         2         N             1
K         2         L             1
L         1         L             1
N         1         N             1
O         2         N             1
P         1         P             1
Q         1         Q             1
R         1         R             1
S         2         N             1
```

Scenario




```
#  instamesh routing table
Dest      cost      next hop      next hop cost
----      ----      ---------     --------------
A         4         I             1
```

RAJANT

Log Files

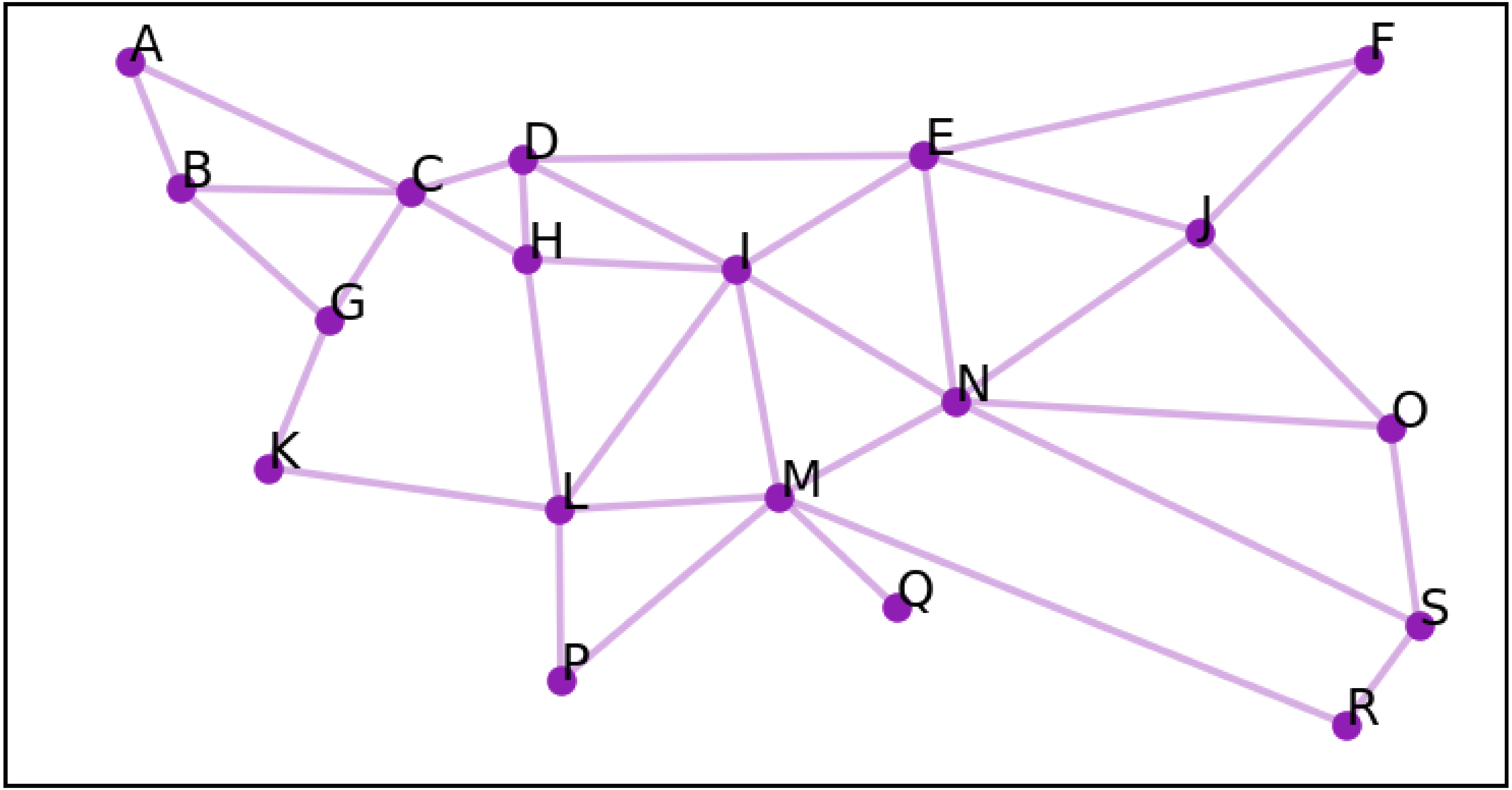
```
#  breadcrumb serial number
M

#  instamesh neighbors
neighbor    cost
--------    ----
I           1
L           1
N           1
P           1
Q           1
R           1
```

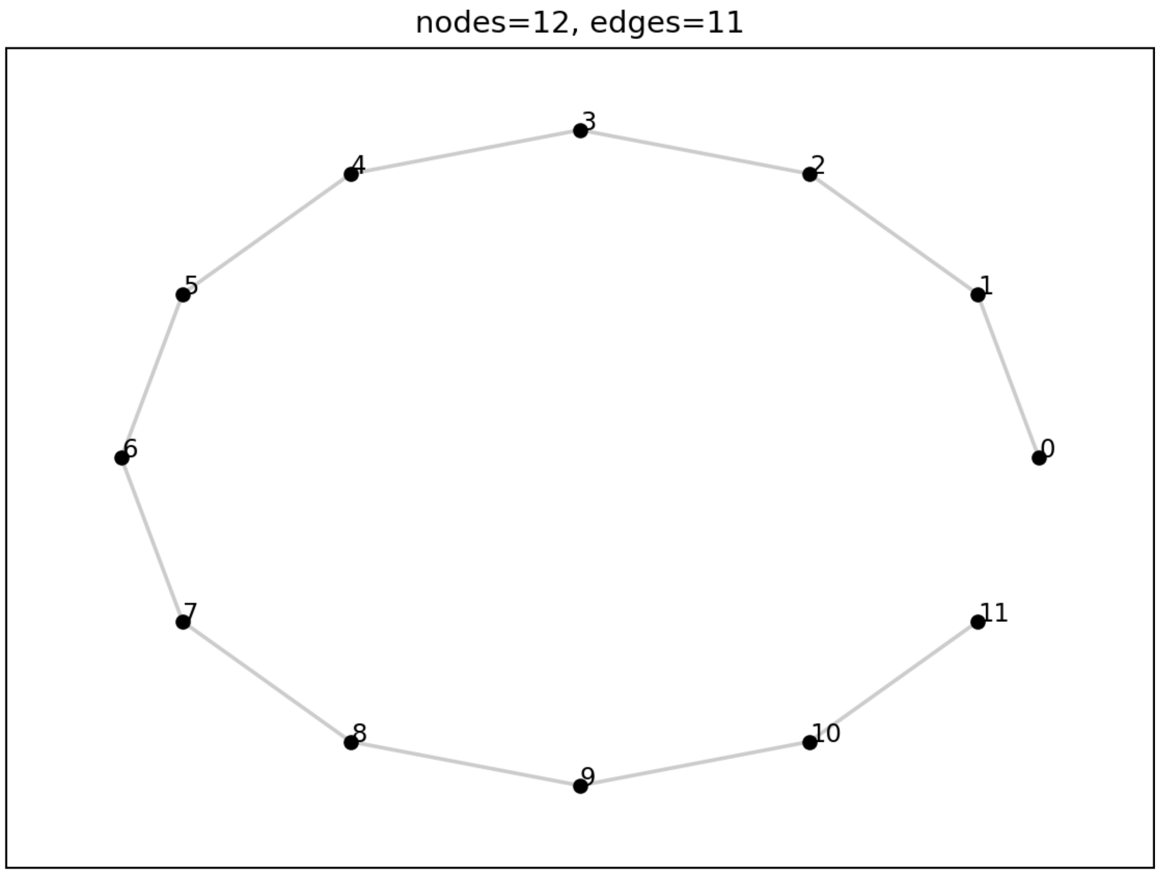
```
#  instamesh routing table
Dest    cost    next hop    next hop cost
----    ----    --------    -------------
A       4       I           1
B       4       I           1
C       3       I           1
D       2       I           1
E       2       I           1
F       3       I           1
                            1
                            1
                            1
                            1
                            1
L       1       L           1
N       1       N           1
O       2       N           1
P       1       P           1
Q       1       Q           1
R       1       R           1
S       2       N           1
```

My job is to make software that generates useful reports from all this stuff.

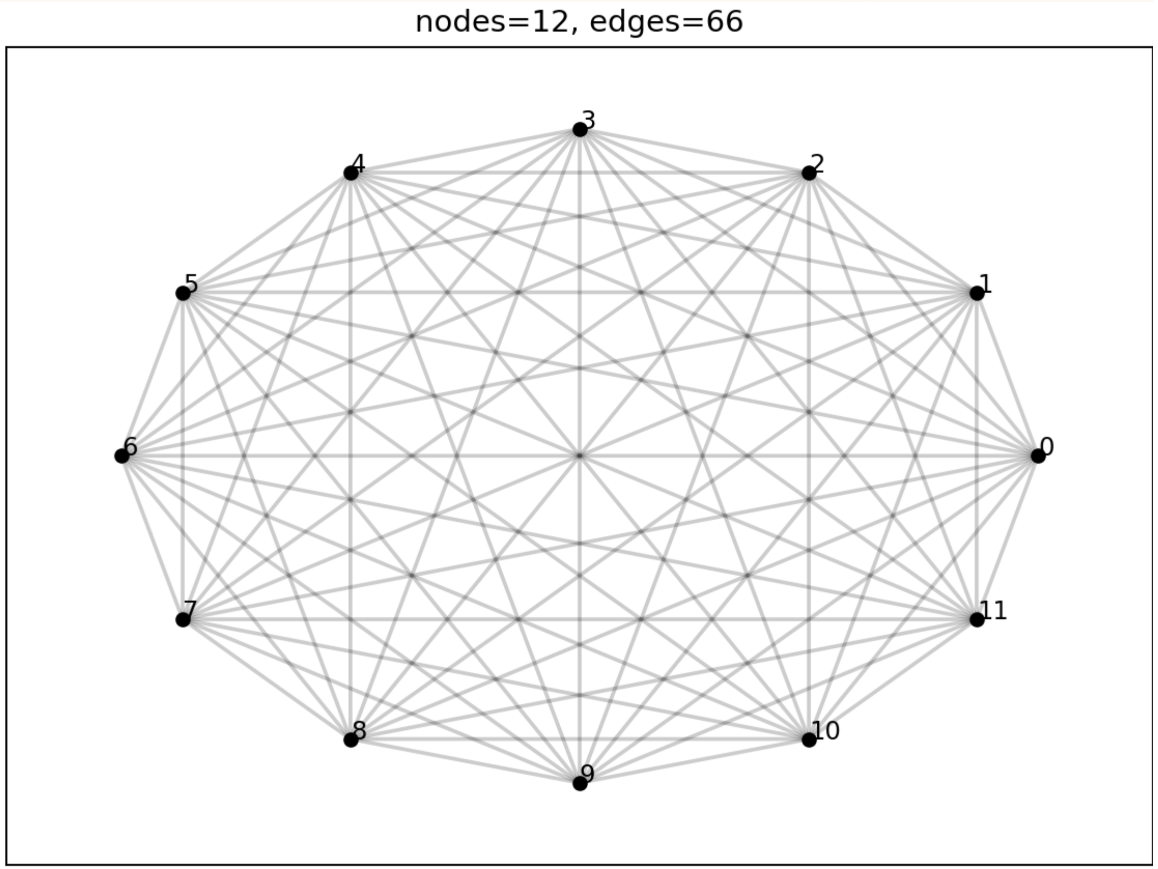
Scenario



How many edges?



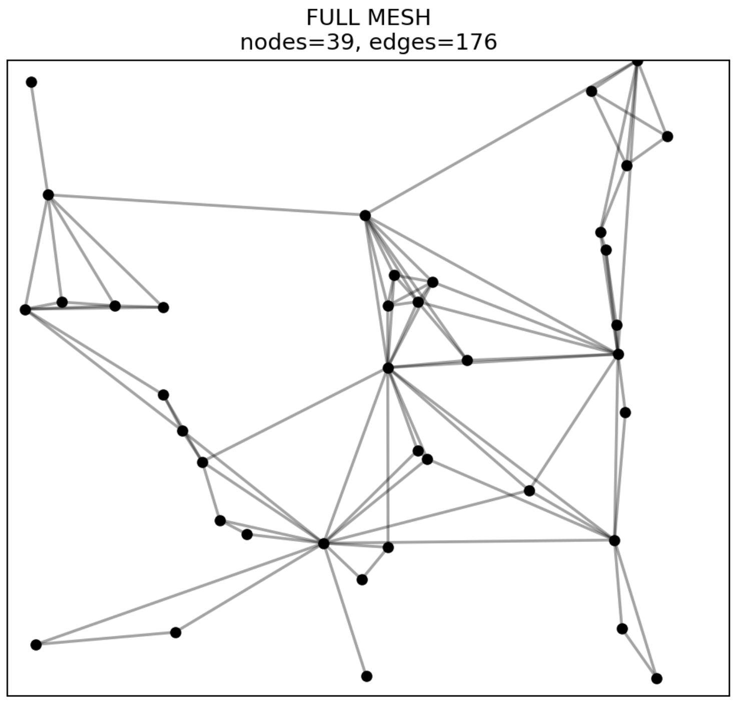
$$min = n - 1 = O(n)$$

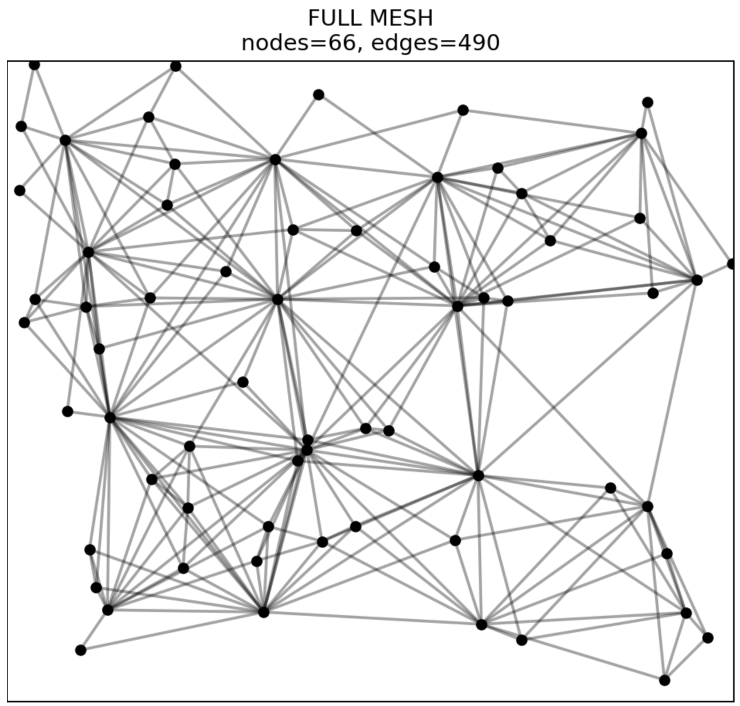


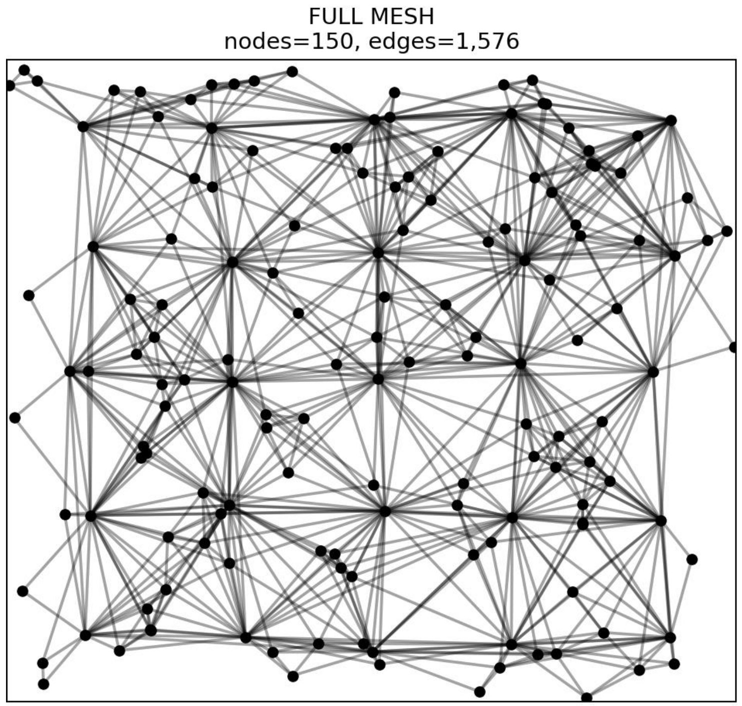
$$max = \frac{n^2 - n}{2} = O(n^2)$$

Example meshes

Rajant meshes are usually pretty dense.





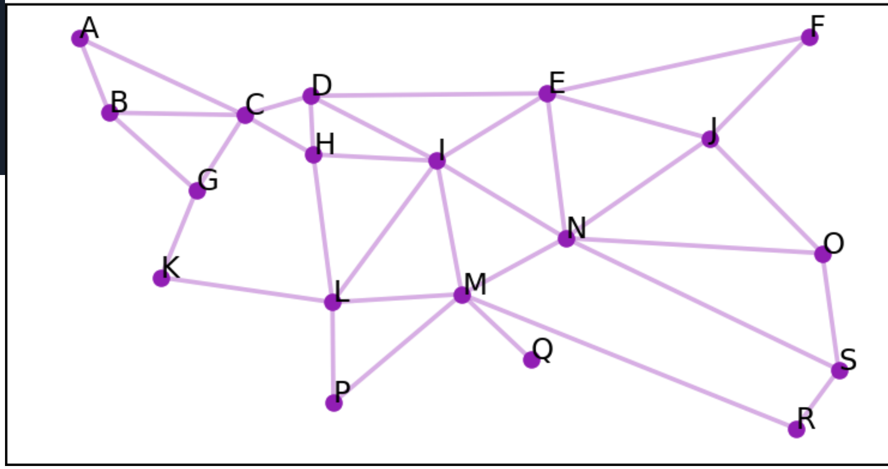


So what?

Things we can do with our graph representation:

- Find the longest routing path
- Find the vertex that is used the most commonly in routing
- Find the edge that is used the most commonly in routing
- Find nodes that would cause the network to be split if they failed.

Longest routing path



Problem: What is the longest active path?

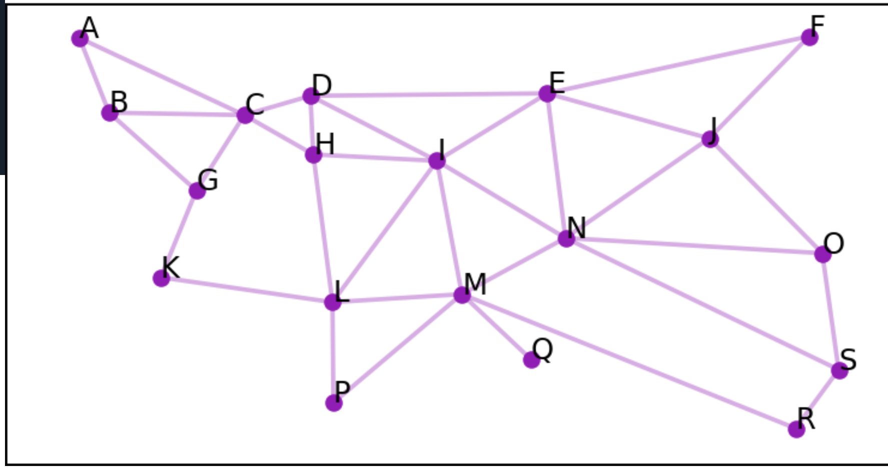
Strategy:

- Find all pairs of nodes (start, finish)
  - How many pairs?
  - O(n^2)
- Find length of each route
  - How long does this take?
  - O(n) in worst case, O(1) in most practical cases.

Overall running time:

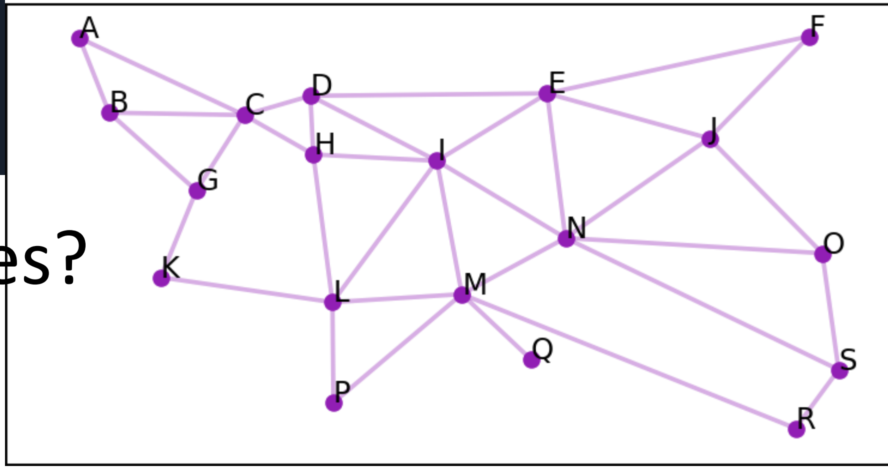
- O(n^2) pairs, each pair takes O(1), so O(n^2 * 1) = O(n^2)

Longest routing path




```
source='A' dest='O'
longest_path=
[
    InstameshRoutingTableEntry(next_hop_node='C'),
    InstameshRoutingTableEntry(next_hop_node='D'),
    InstameshRoutingTableEntry(next_hop_node='E'),
    InstameshRoutingTableEntry(next_hop_node='J'),
    InstameshRoutingTableEntry(next_hop_node='O')
]
```

Most common vertex



Problem: Which vertex is used most often in routes?

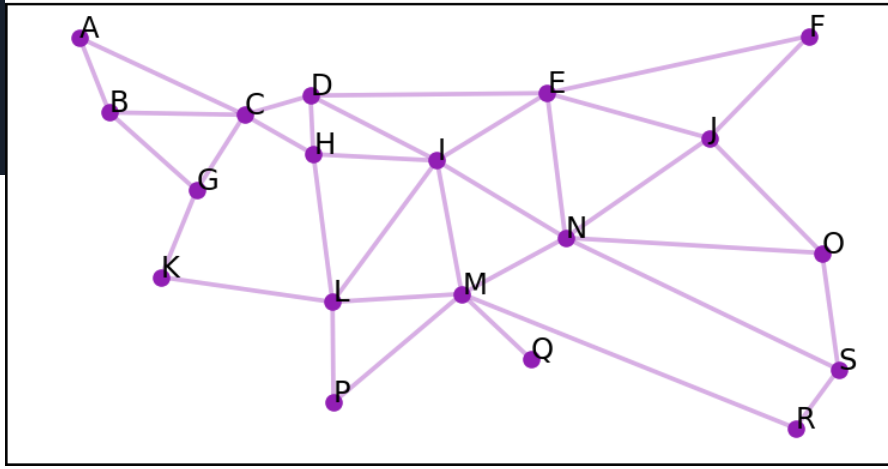
Strategy:

- Find all pairs of nodes (start, finish)
  - How many pairs?
  - O(n^2)
- Traverse each route and add 1 for each vertex we find
  - How long does this take?
  - O(n) in worst case, O(1) in most practical cases.

Overall running time:

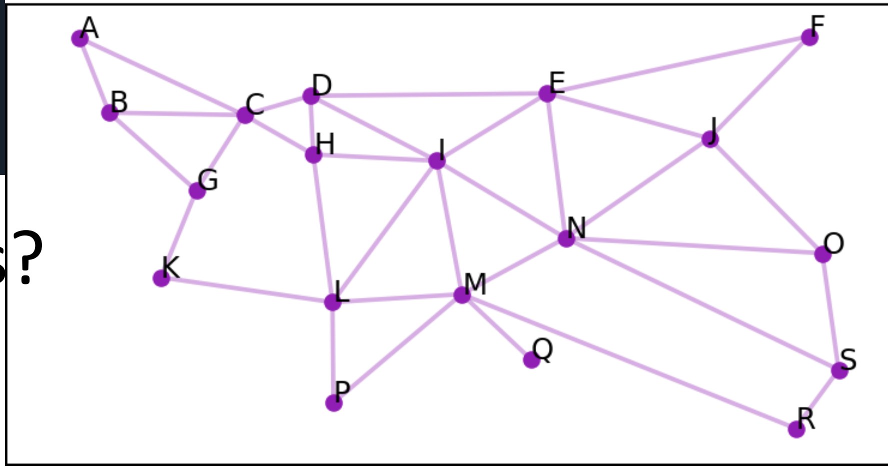
- O(n^2) pairs, each pair takes O(1), so O(n^2 * 1) = O(n^2)

Most common vertex




```
waypoints.most_common()=
[
    ('D', 80),
    ('E', 78),
    ('C', 76),
    ('I', 76),
    ('M', 70),
    ('N', 52),
    ('L', 48),
    ('H', 16),
    ('J', 14),
    ('K', 10),
    ('G', 8),
    ('B', 4),
    ('S', 2),
]
```

Most common edge



Problem: Which edge is used most often in routes?

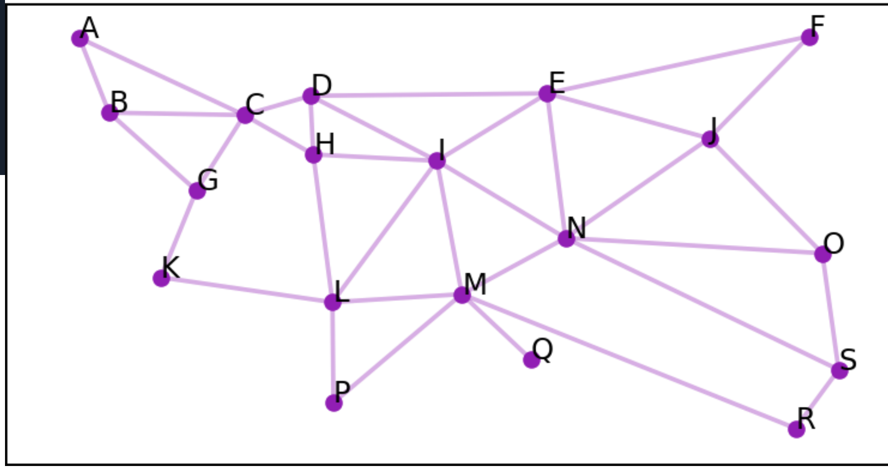
Strategy:

- Find all pairs of nodes (start, finish)
  - How many pairs?
  - O(n^2)
- Traverse each route and add 1 for each (vertex1, vertex2) pair we find
  - How long does this take?
  - O(n) in worst case, O(1) in most practical cases.

Overall running time:

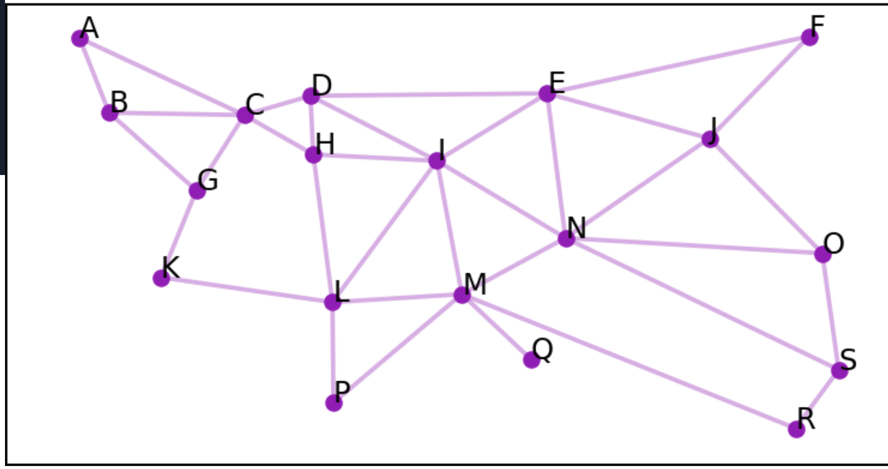
- O(n^2) pairs, each pair takes O(1), so O(n^2 * 1) = O(n^2)

Most common edge




```
waypoints.most_common()=
[
    (('C', 'D'), 42),
    (('D', 'C'), 42),
    (('D', 'E'), 33),
    (('E', 'D'), 33),
    (('I', 'M'), 24),
    (('M', 'I'), 24),
    (('M', 'Q'), 18),
    (('L', 'K'), 18),
    (('K', 'L'), 18),
    (('Q', 'M'), 18),
    (('D', 'I'), 17),
    (('E', 'I'), 17),
    (('I', 'L'), 17),
    (('I', 'D'), 17),
    (('I', 'E'), 17),
    (('L', 'I'), 17),
    # Plus a bunch more
]
```

Critical Nodes



Problem: What nodes will cause a disconnection if they fail?

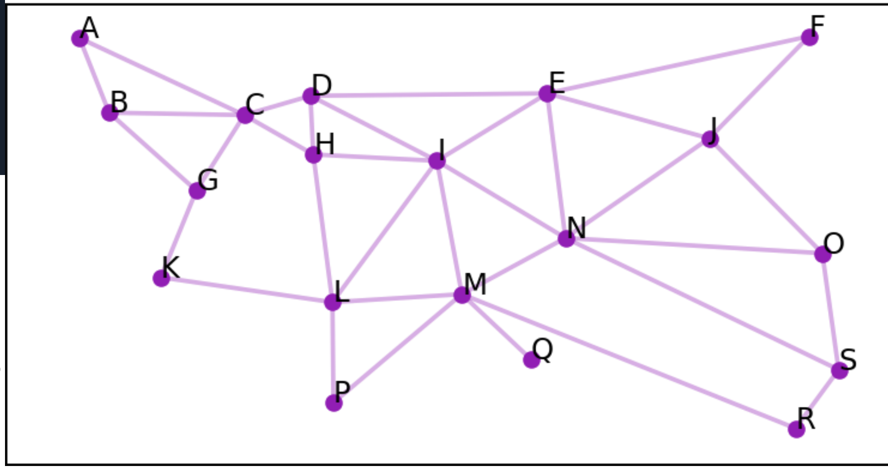
Example: If node M fails, node Q will have no connection to the rest of the mesh, so we say that M is a critical node. ("Articulation Point" is the fancy term in graph theory.)

Are there other critical nodes?

Strategy:

- ???

Connected Components

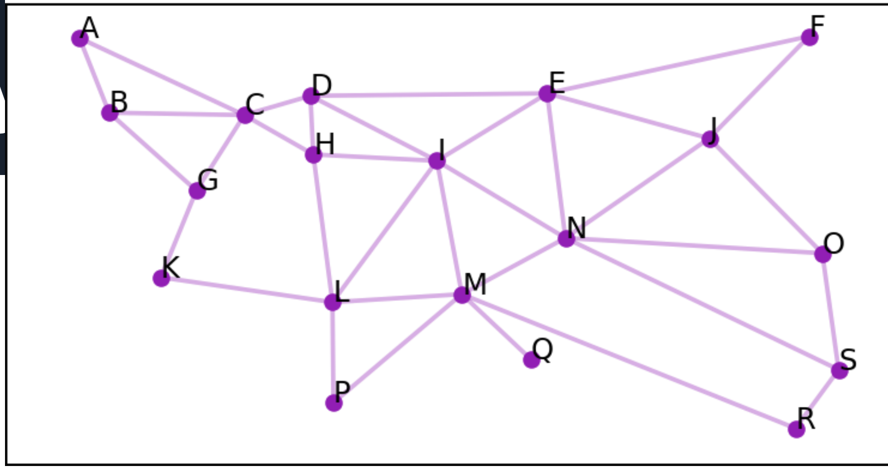


In a graph, a "connected component" is a chunk where all vertices have a path to all other vertices in the chunk.

Connected components algorithm:

- Make a list "remaining" of all vertices.
- Pick first vertex from list.
- Run depth-first search on graph. Remove each vertex from "remaining" as you visit it.
- Repeat until "remaining" is empty. The number of loop iterations is the number of connected components.
- Running time is just running time of depth-first search, which is O(V+E)

Critical Nodes-naïve strategy

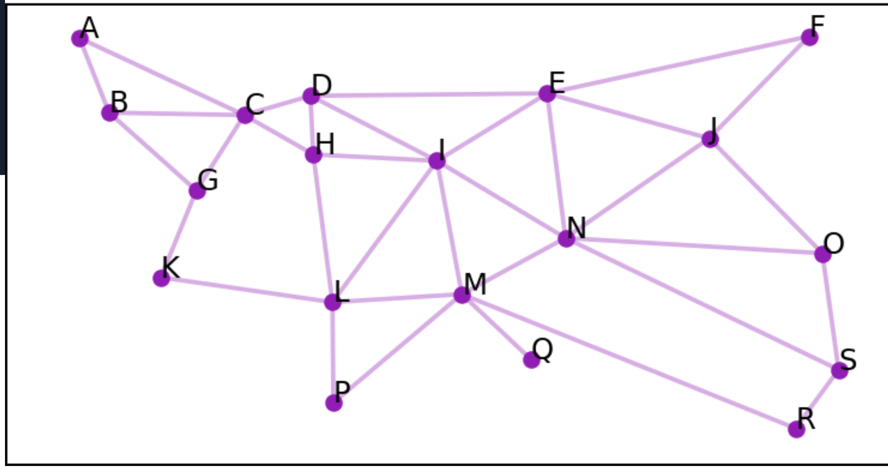


Strategy:

- Remove vertex V
- Run a “connected components” algorithm
  to see if there are more connected components afterwards than before.
- If yes, then V is a critical node. If no, then V is not a critical node.
- Add vertex V back to the graph
- Repeat for all vertices

- Running time?
- Connected components algorithm is O(V+E),
  which = O(V^2) for dense meshes.
- Have to run it V times, so run time is O(V^2 * V) = O(V^3)

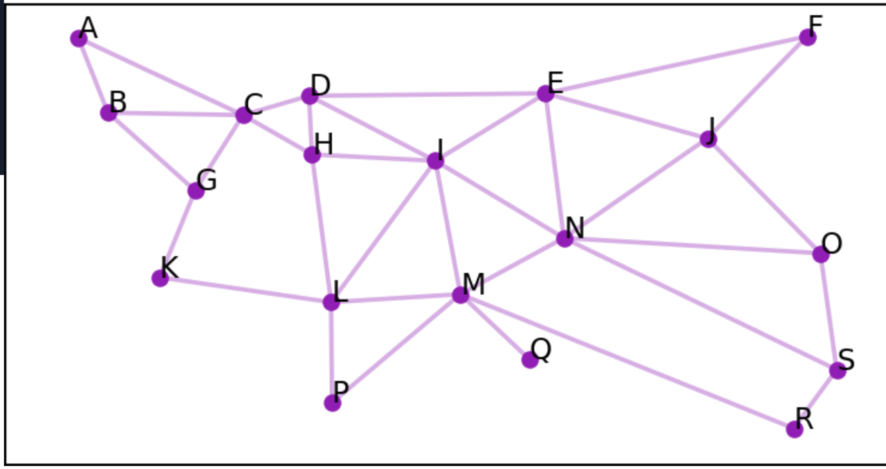
Critical Nodes-fancy strat



Strategy:

- Google “critical points in network graph”
- Find a StackOverflow post that says “These are called ‘Articulation Points.’”
- Google “Articulation Points”
- Find “Tarjan’s Algorithm” Wikipedia page
- Read it. Get confused. But learn that it’s based on DFS and takes O(V+E) time (same as DFS).
- Ask GitHub Copilot to write a function that does it.
- It works!
- Time complexity: O(V+E) = O(V^2) for dense meshes.
- Money complexity: O($10/month) for GitHub Copilot

Critical Nodes-the code



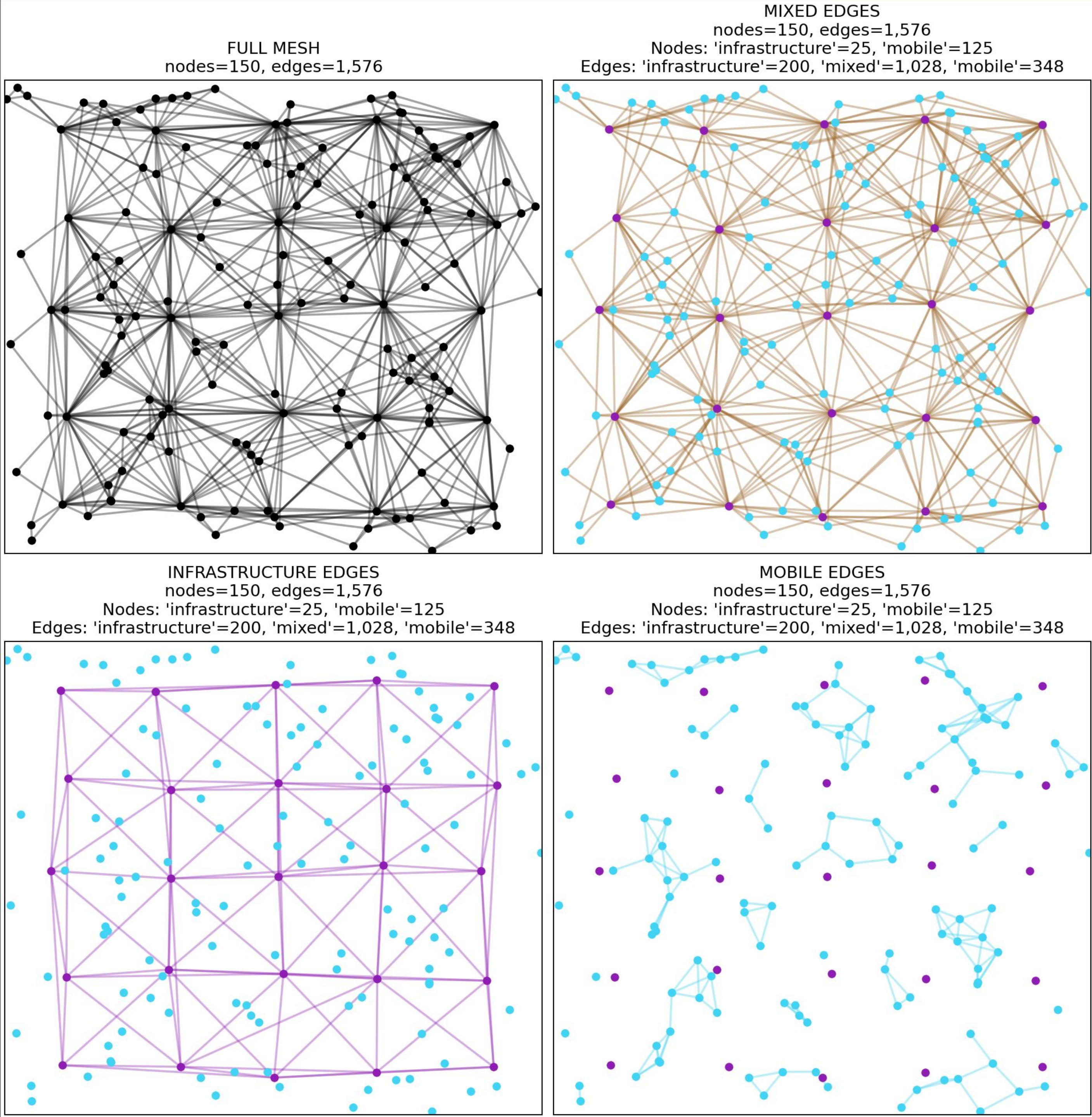

```python
# Entirely generated by GitHub Copilot.
def find_articulation_points(graph: Graph) -> set[Node]:
    def dfs_articulation(node: Node, parent: Node, visited: set[Node],
        discovery: dict[Node, int], low: dict[Node, int], time: int,
        articulation_points: set[Node],
    ):
        visited.add(node)
        discovery[node] = low[node] = time
        children = 0
        for neighbor in node.neighbors:
            if neighbor == parent:
                continue
            if neighbor not in visited:
                children += 1
                dfs_articulation(neighbor, node, visited, discovery, low, time + 1,articulation_points)
                low[node] = min(low[node], low[neighbor])
                if parent is None and children > 1:
                    articulation_points.add(node)
                if parent is not None and low[neighbor] >= discovery[node]:
                    articulation_points.add(node)
            else:
                low[node] = min(low[node], discovery[neighbor])
    visited = set()
    discovery = {}
    low = {}
    articulation_points = set()
    for node in graph.nodes:
        if node not in visited:
            dfs_articulation(node, None, visited, discovery, low, 0, articulation_points)
    return articulation_points
```

And now for something completely different!

Node types

Most of our meshes have permanent/infrastructure nodes (breadcrumbs mounted on poles) and temporary/mobile ones (breadcrumbs mounted on vehicles).

We care about the infrastructure ones a lot more.

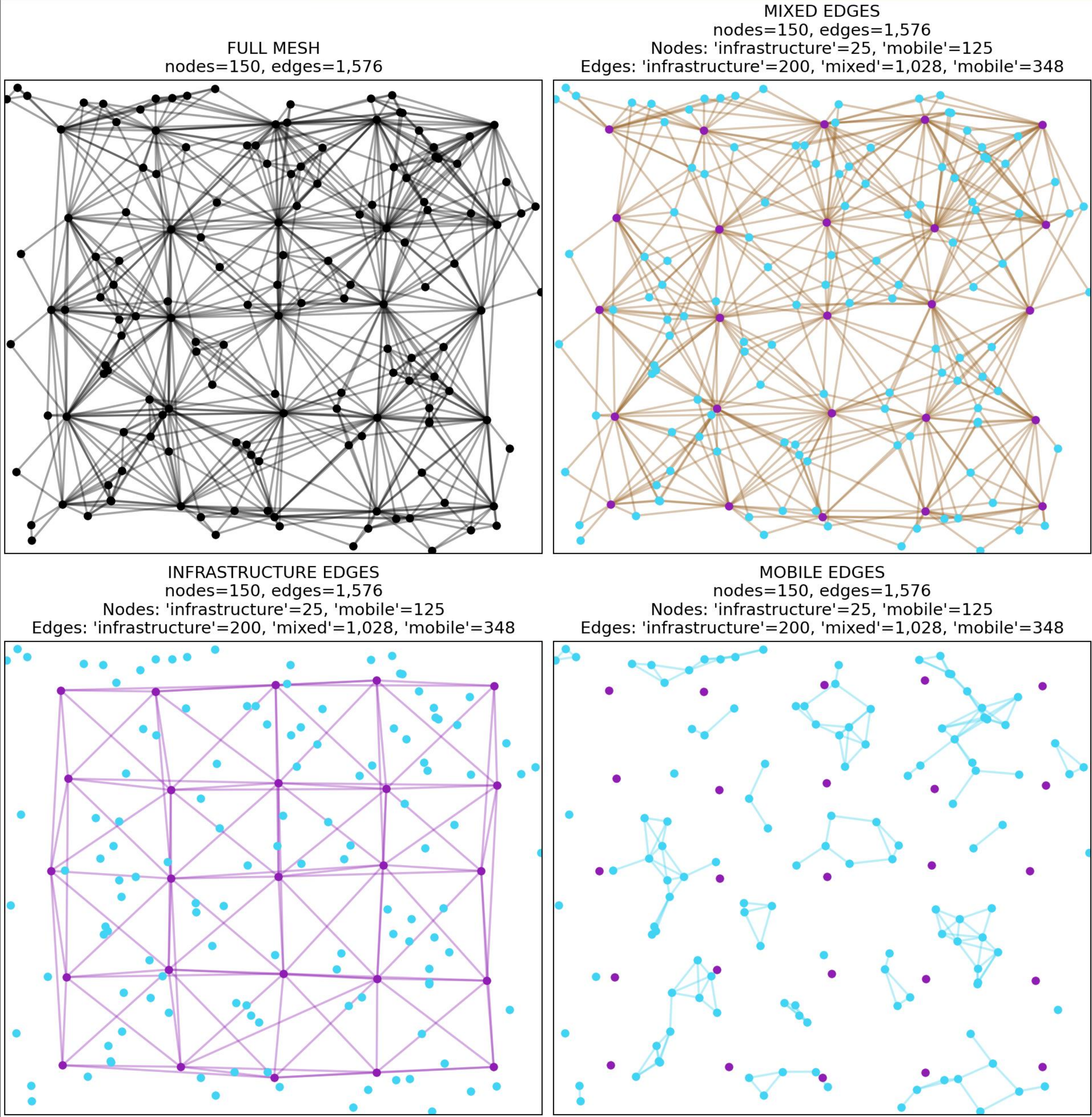


Node types

What we get from the logs is the upper left graph.

How can we separate out the infrastructure (lower left) from mobile (lower right) nodes?

If you can think of a good answer, email me your resume.



thank you!

(Horrible) code and this presentation available at
github.com/davidmayo/acm-presentation

David Mayo
Rajant Corporation, Morehead, KY
dmayo@rajant.com