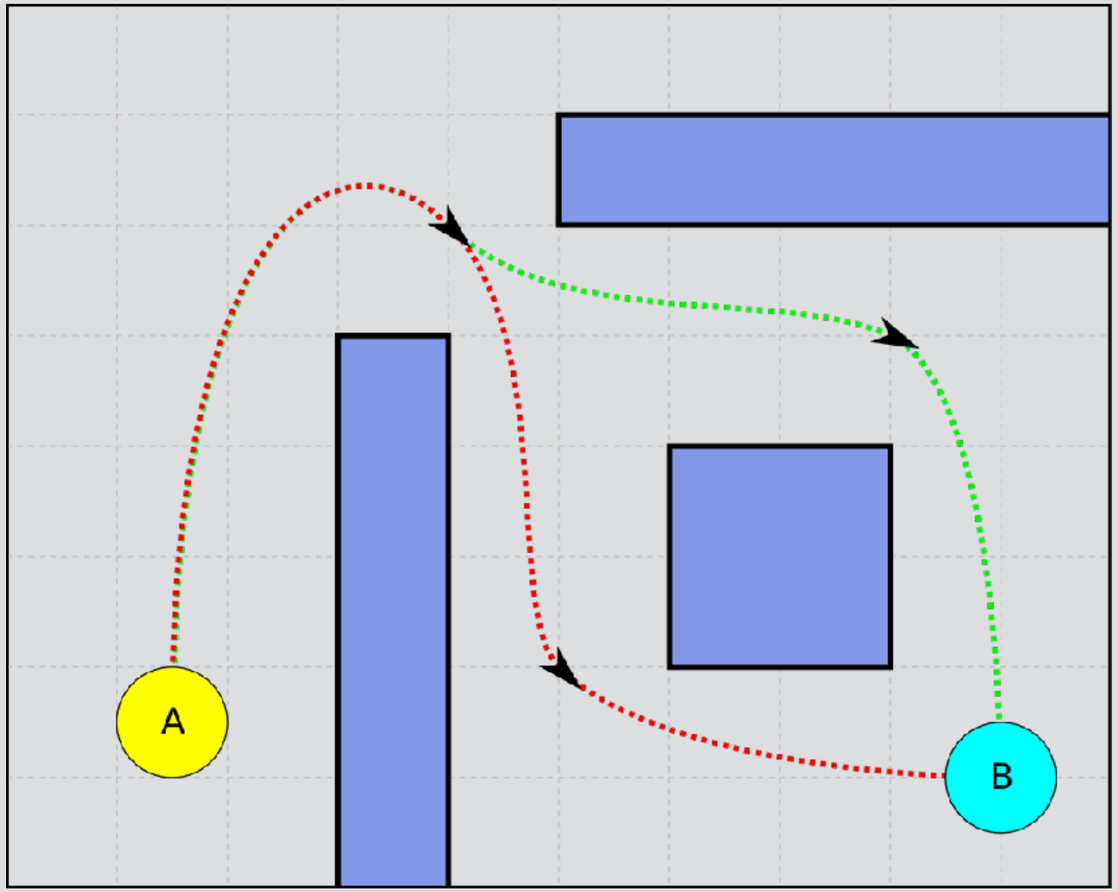
Game AI.

Part 2

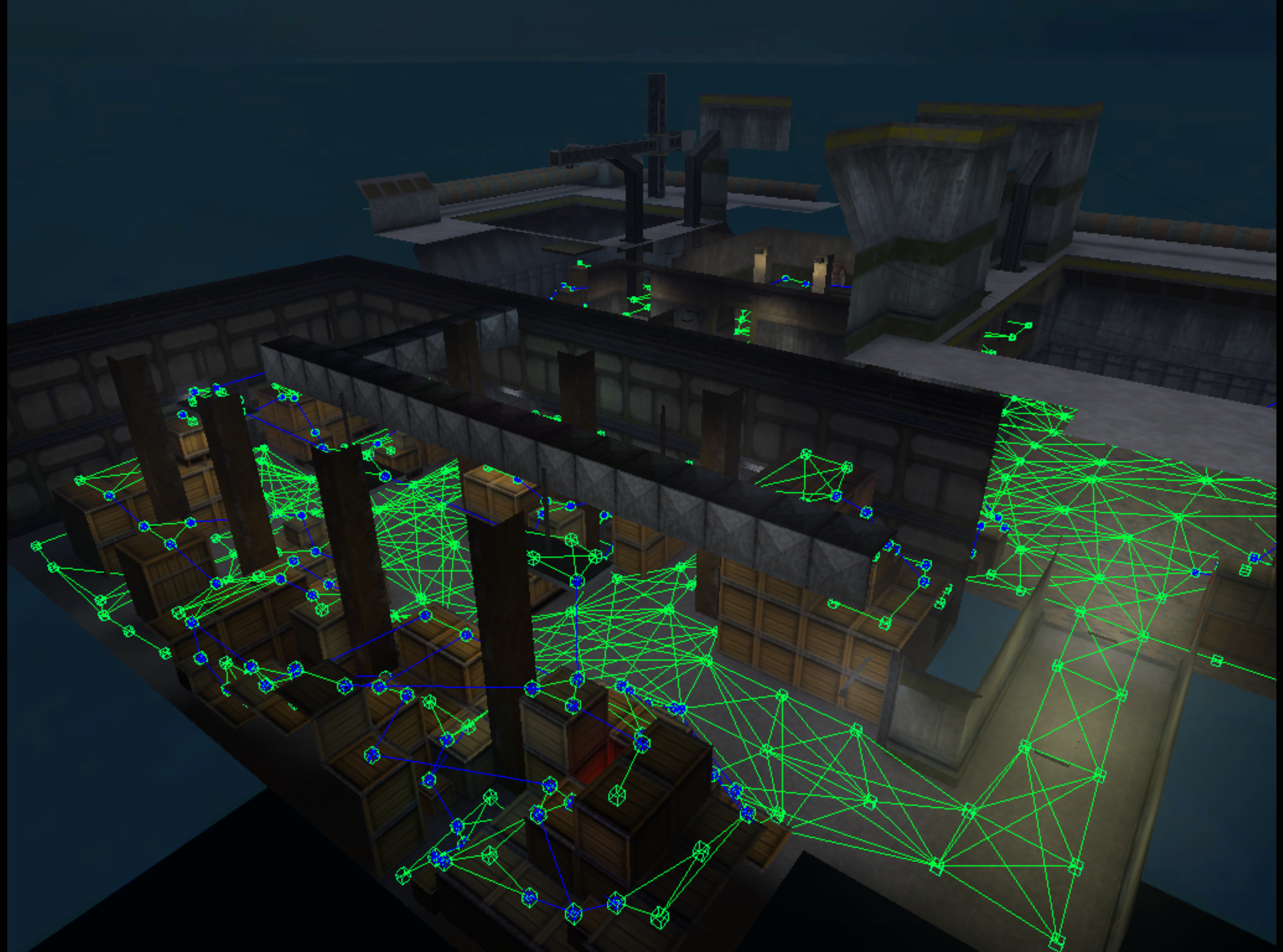


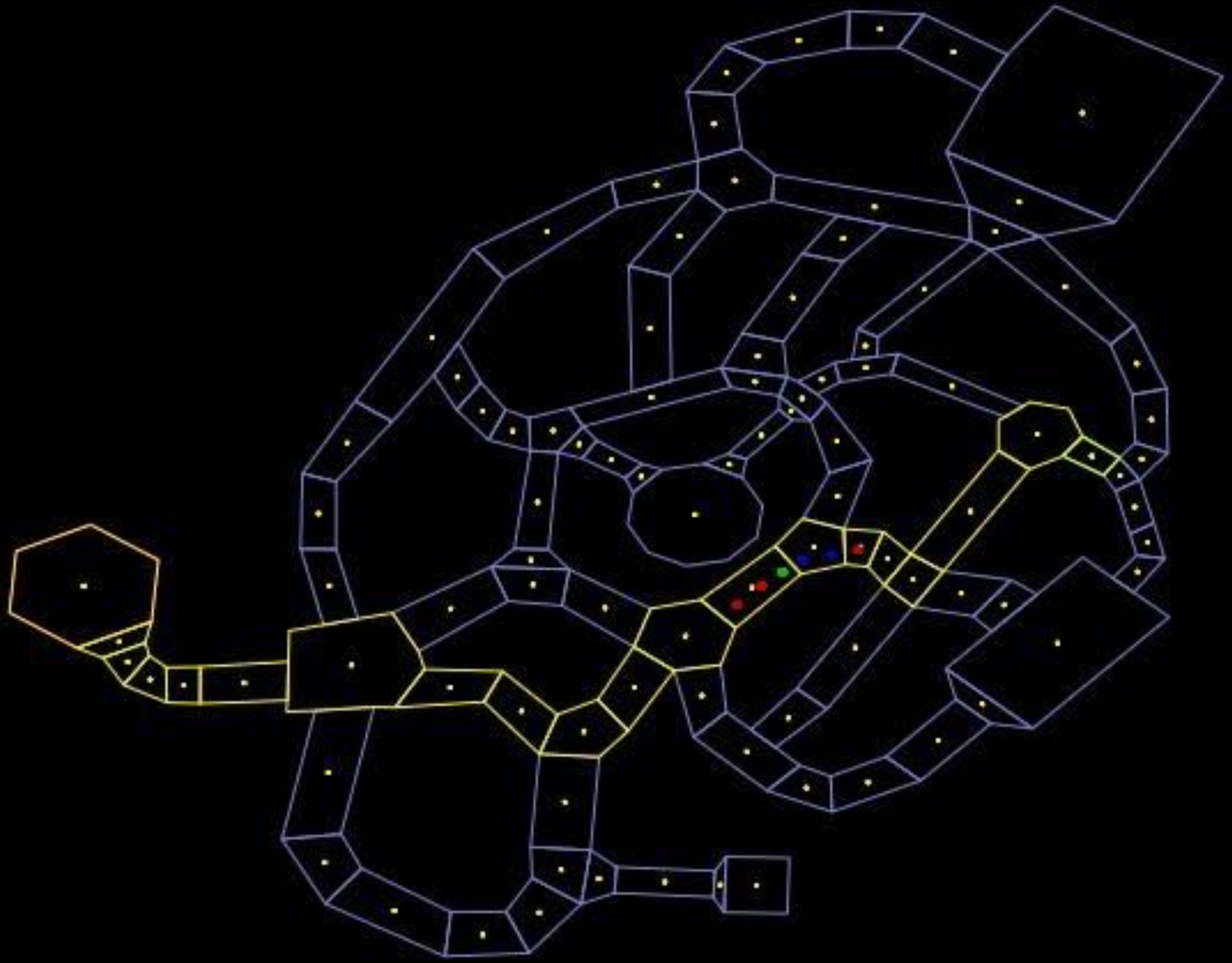
Pathfinding

Navigating game maps.

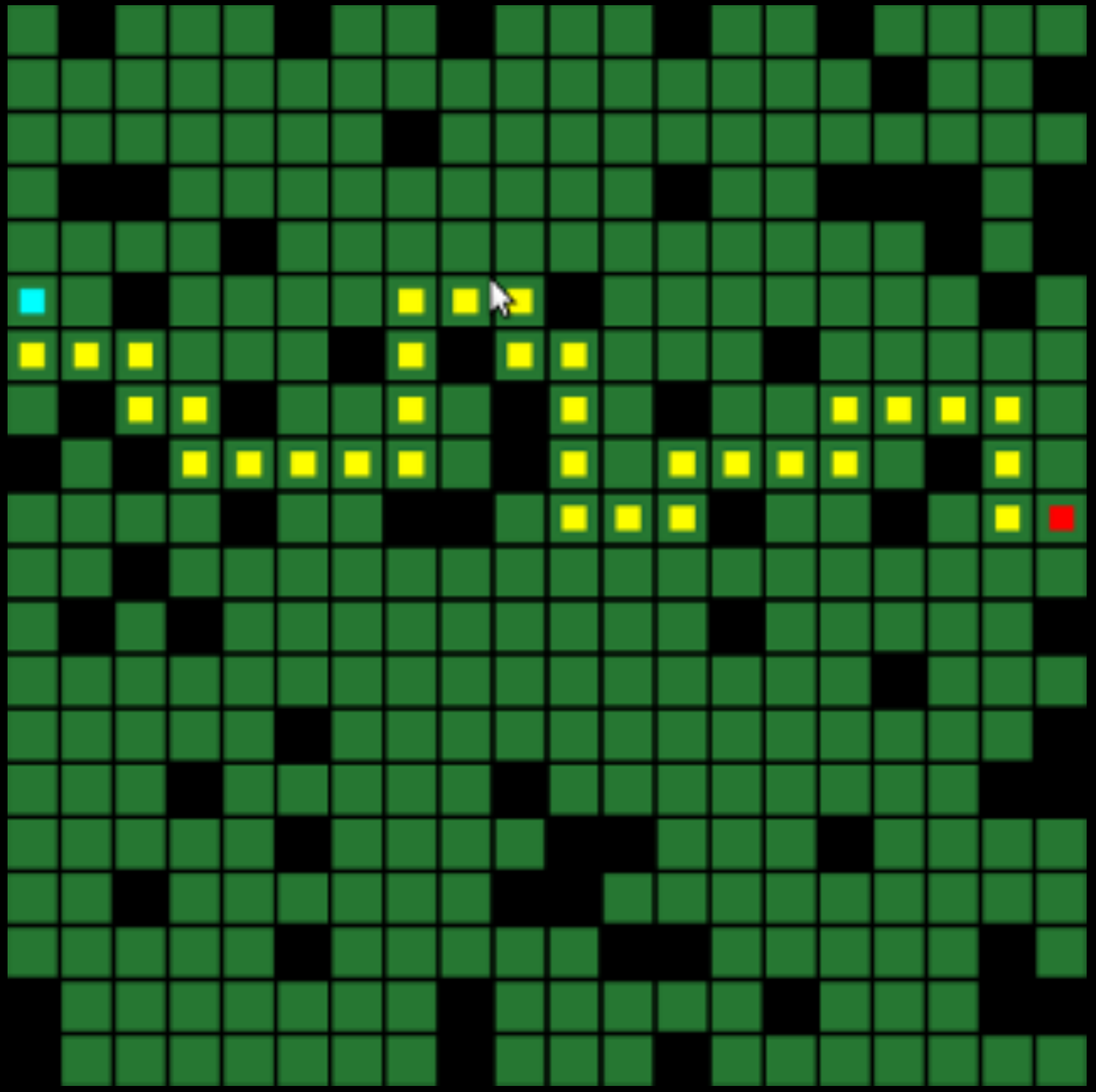

2
905

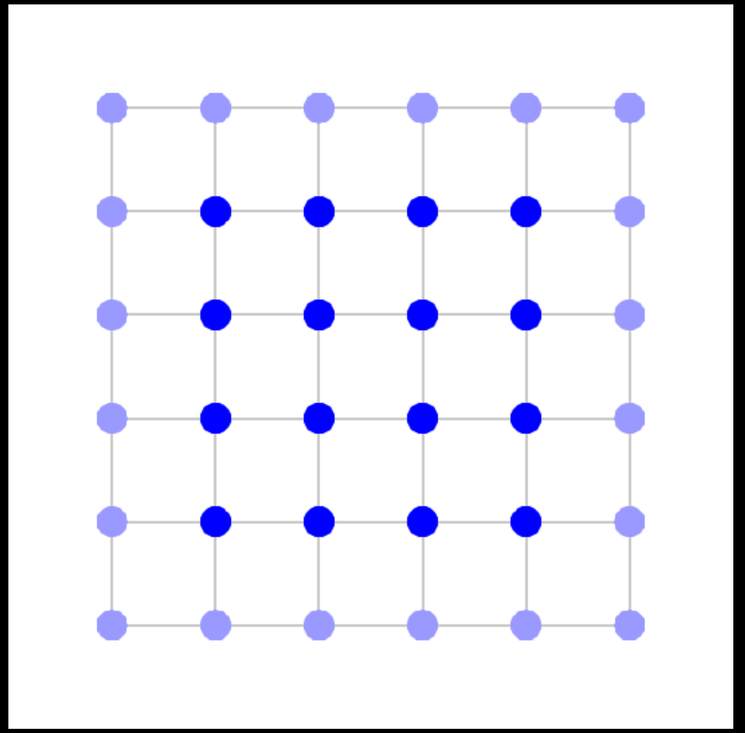
Navigation nodes.



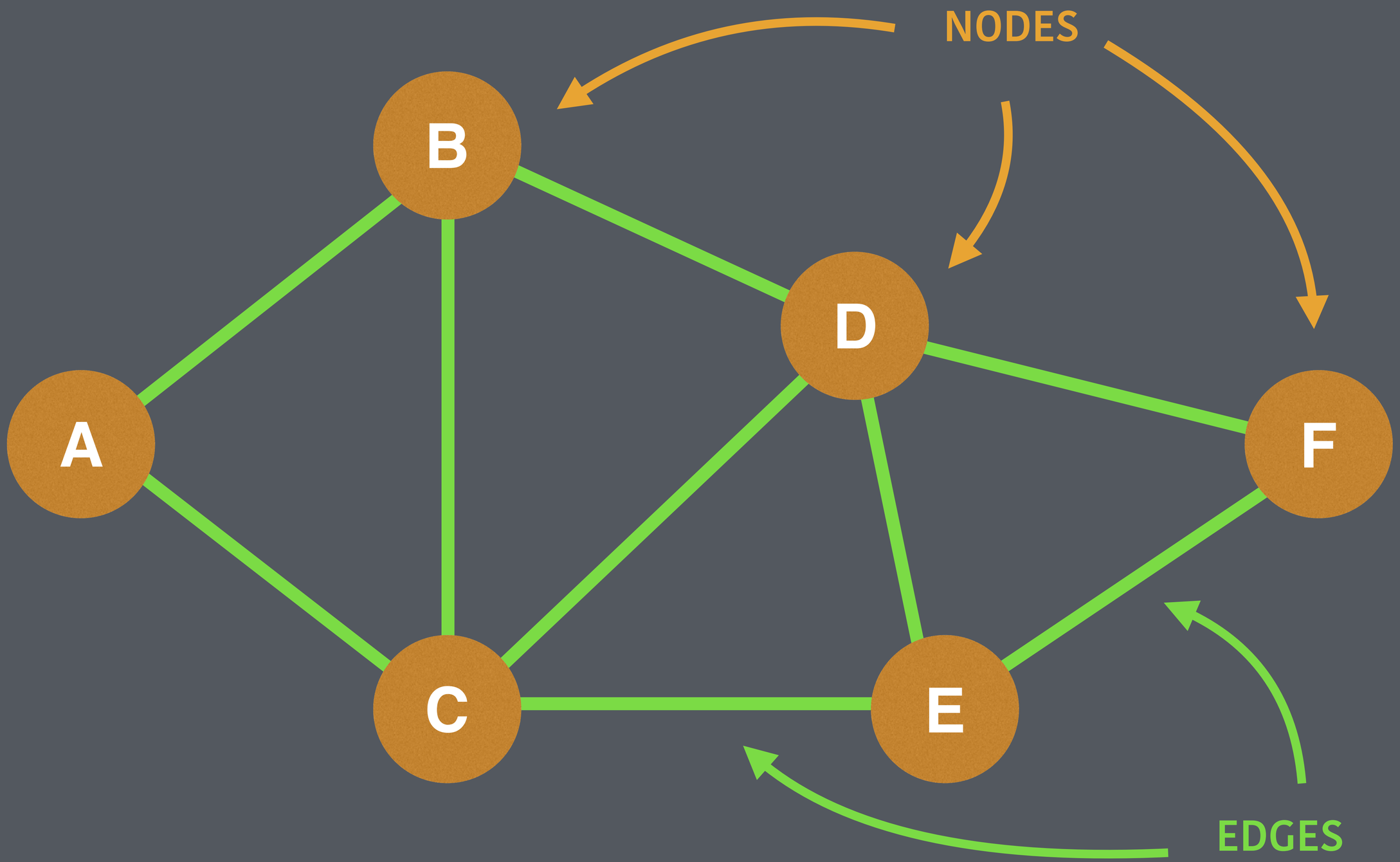



Pathfind
Distance to Wall
Raycast
Find Polys Around
Region
Min Region Size 50
Merged Region Size 20
Polygonization
Max Edge Length 12
Max Edge Error 1.3
Verts Per Poly 6
Tiling
TileSize 32
Tiles 37 x 50
Keep Itermediate Results
Measure Per Tile Timings
Build
Debug Mode
Draw
Input Mesh
Navmesh
Navmesh Invis
Navmesh Trans
Navmesh BVTree
Poly Mesh
Tick 'Keep Itermediate Results' to see more debug mode options.
s 732
196
5
10
15
20
25
169





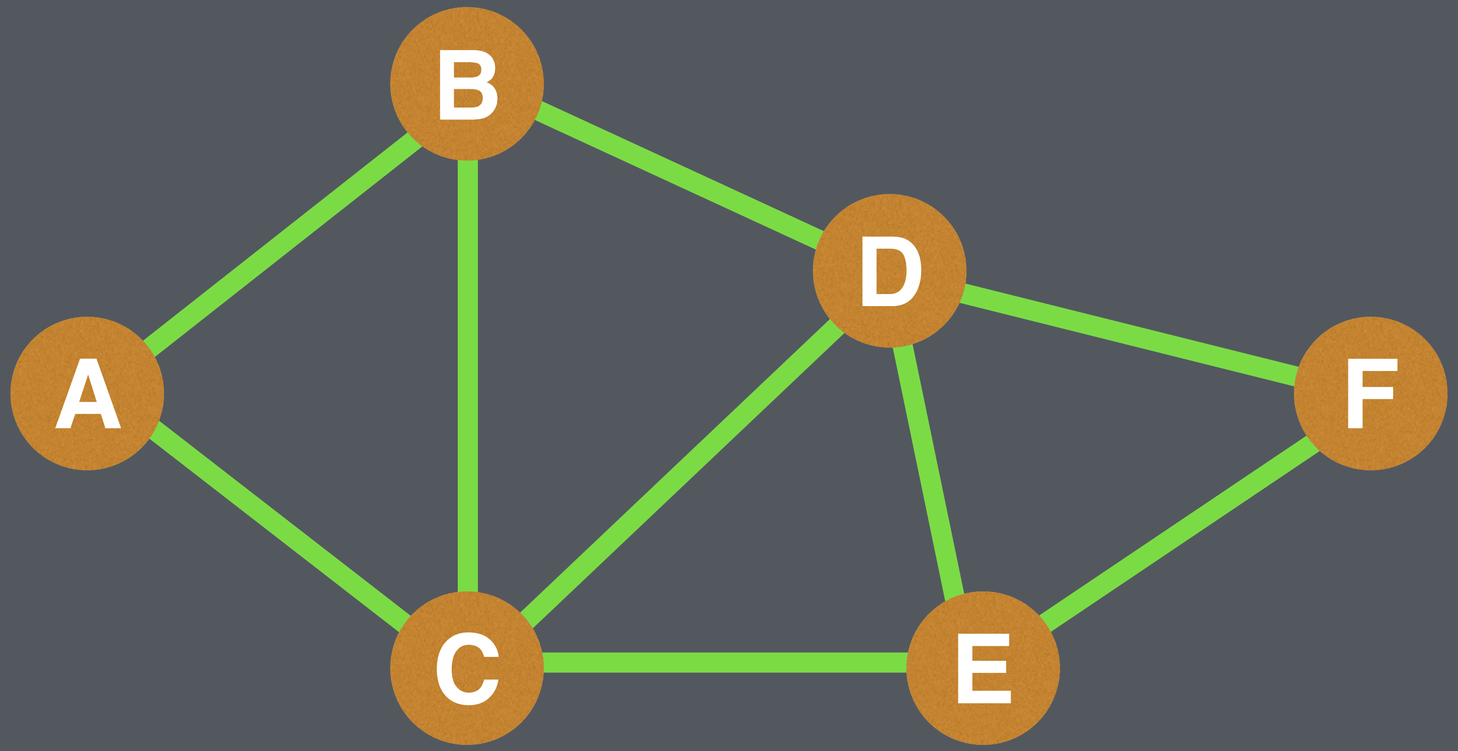
Graphs


NODES
B
D
A
F
C
E
EDGES

Finding the path between two graph nodes.

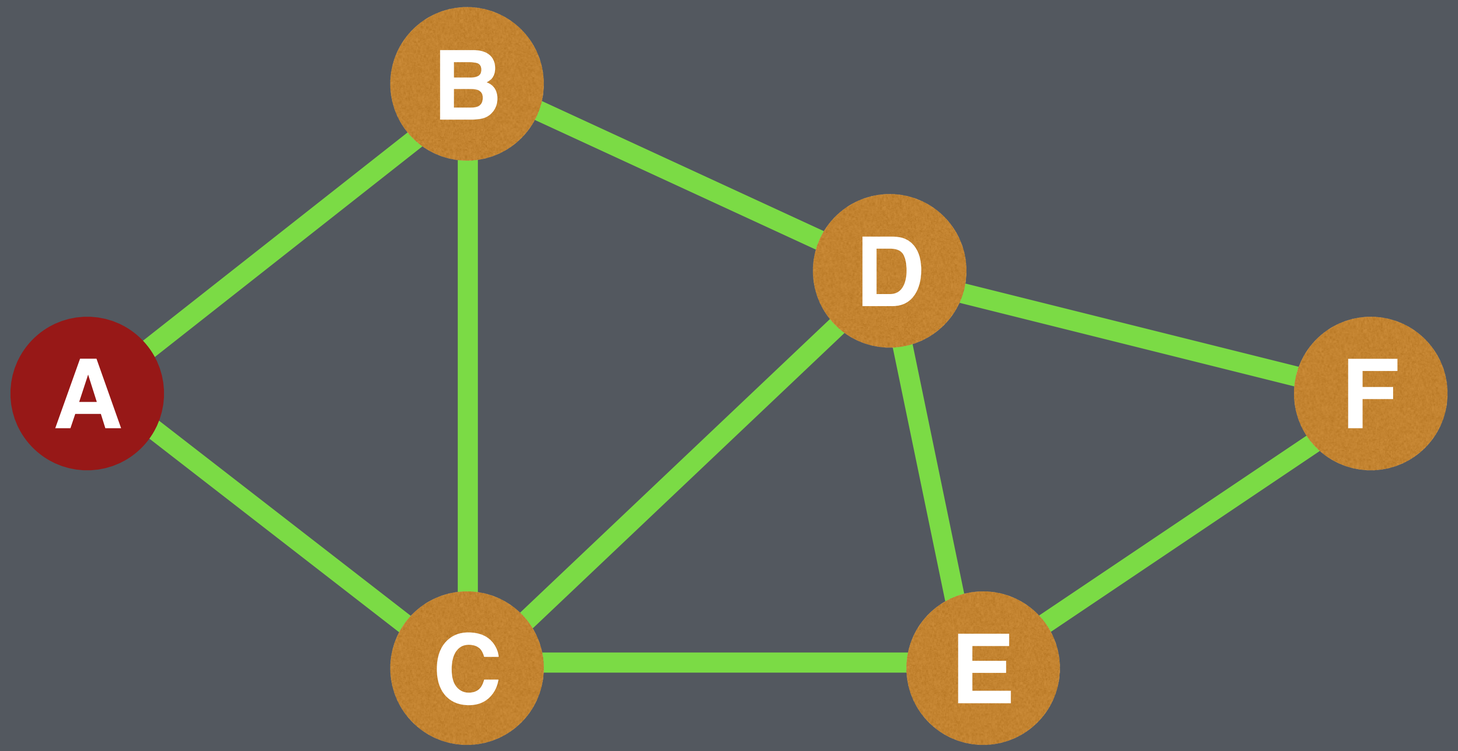
Breadth First Search algorithm.

```
class Node {
    public:
        Node() {}
        bool visited;
        Node *cameFrom;
        std::vector<Node*> connected;
};
```



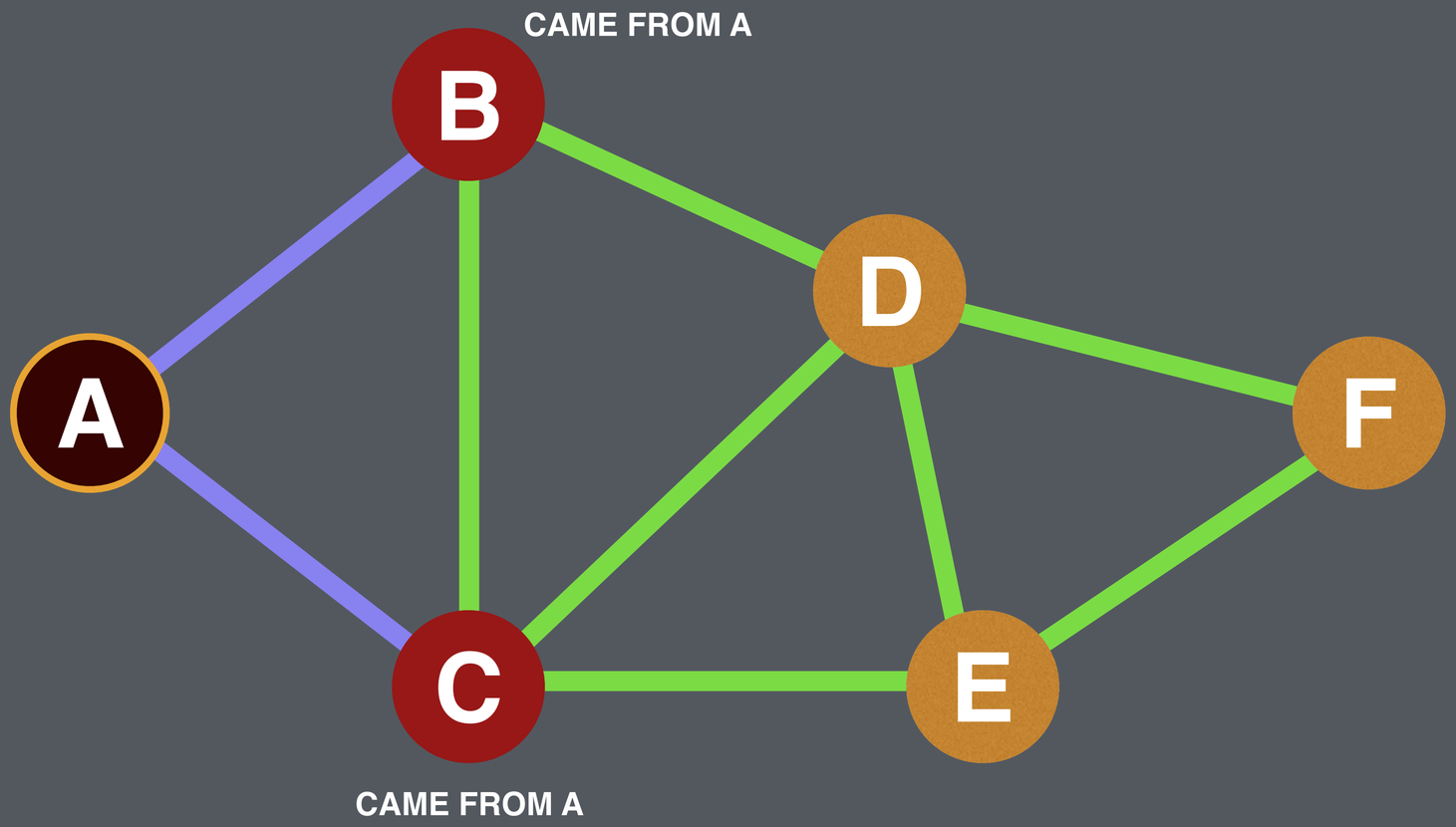
Finding the path from A to F using Breadth First Search

QUEUE:



Add our starting node to the queue.

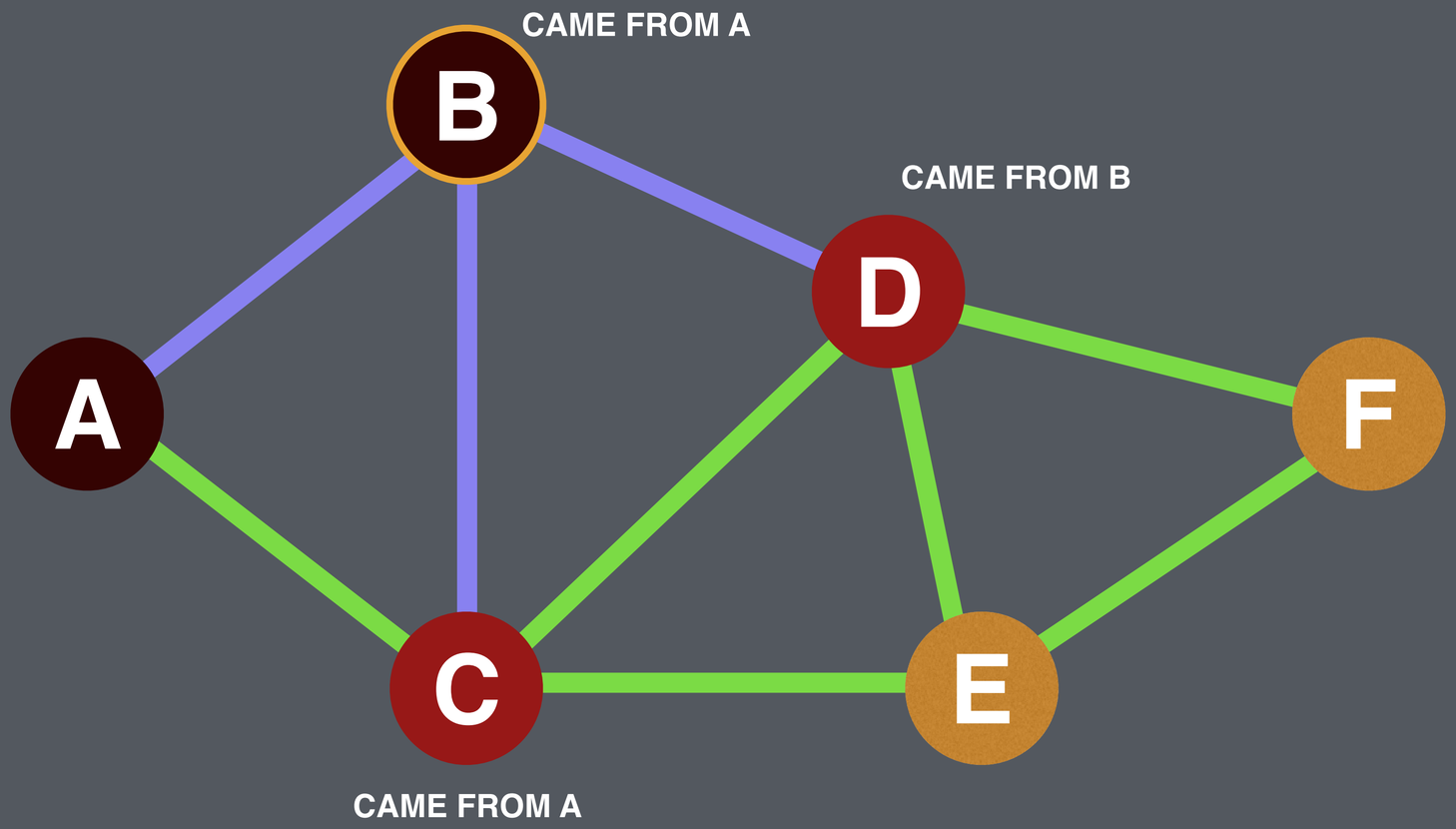
QUEUE: A



While queue has items in it, pop the front queue item, add its connected nodes to the queue if they haven’t been visited, mark them visited and set their “cameFrom” node to the node we popped from the queue.

CURENT NODE: A

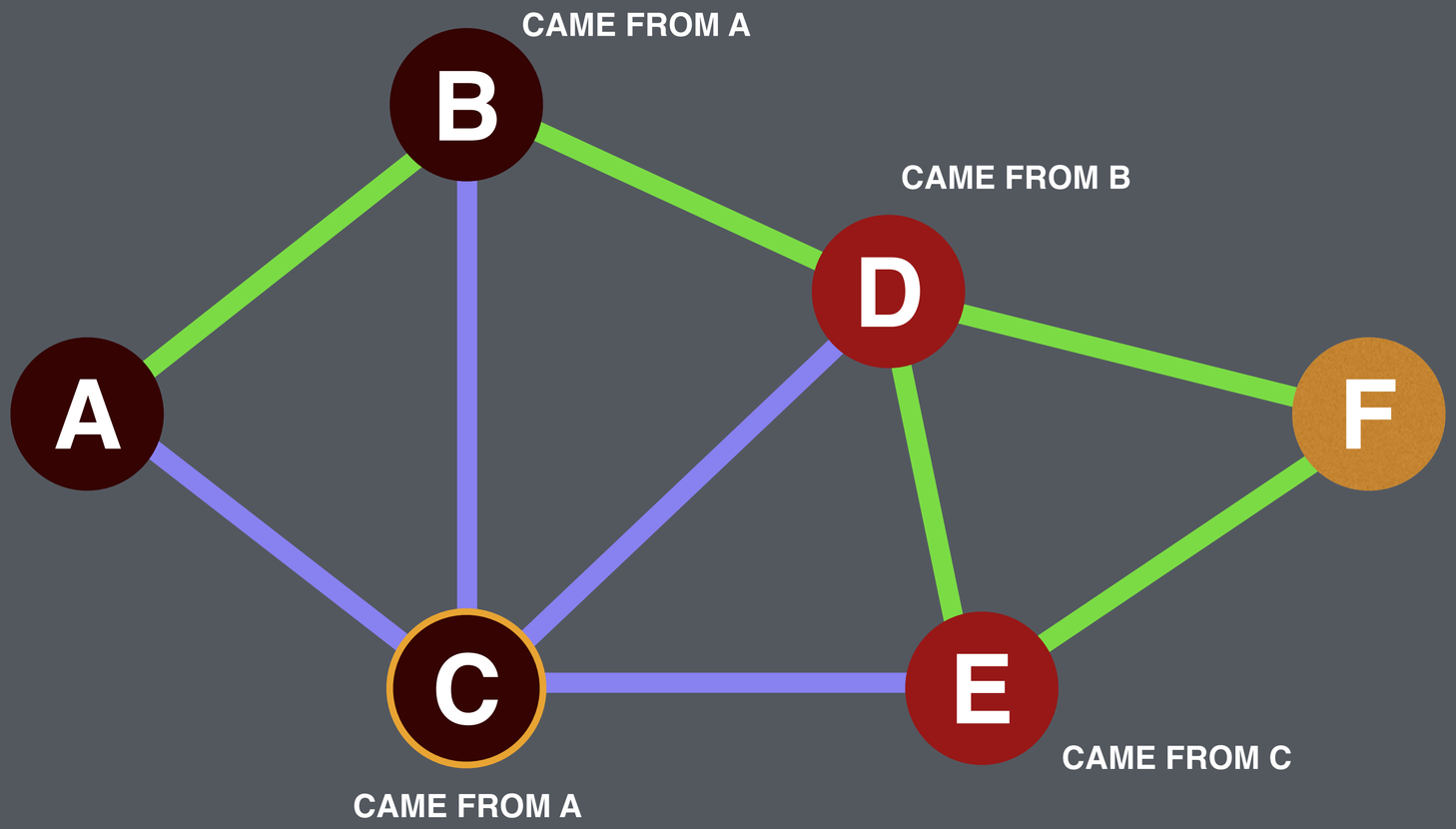
QUEUE: B,C



While queue has items in it, pop the front queue item, add its connected nodes to the queue if they haven’t been visited, mark them visited and set their “cameFrom” node to the node we popped from the queue.

CURENT NODE: B

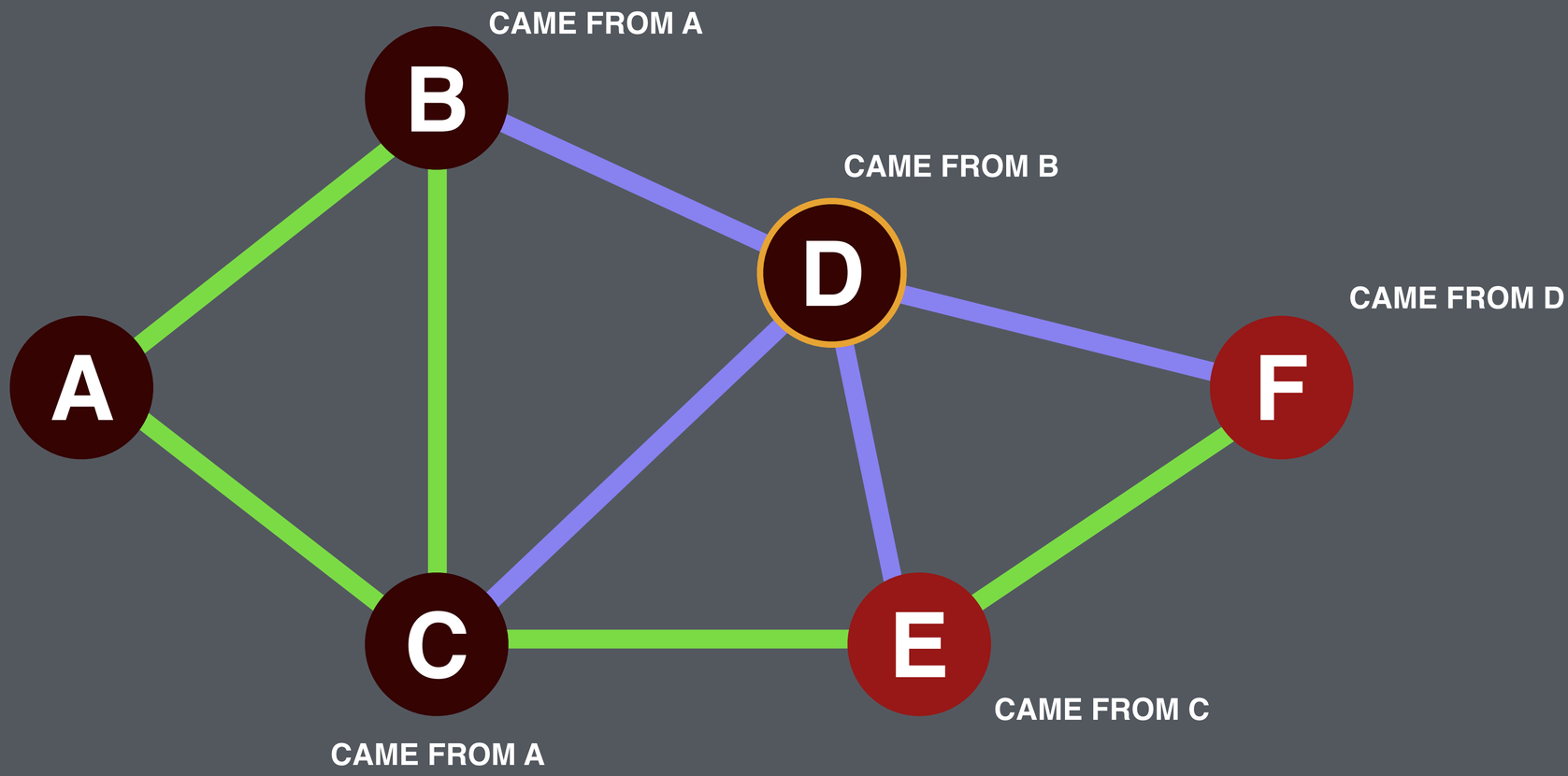
QUEUE: C,D



While queue has items in it, pop the front queue item, add its connected nodes to the queue if they haven’t been visited, mark them visited and set their “cameFrom” node to the node we popped from the queue.

CURENT NODE: C

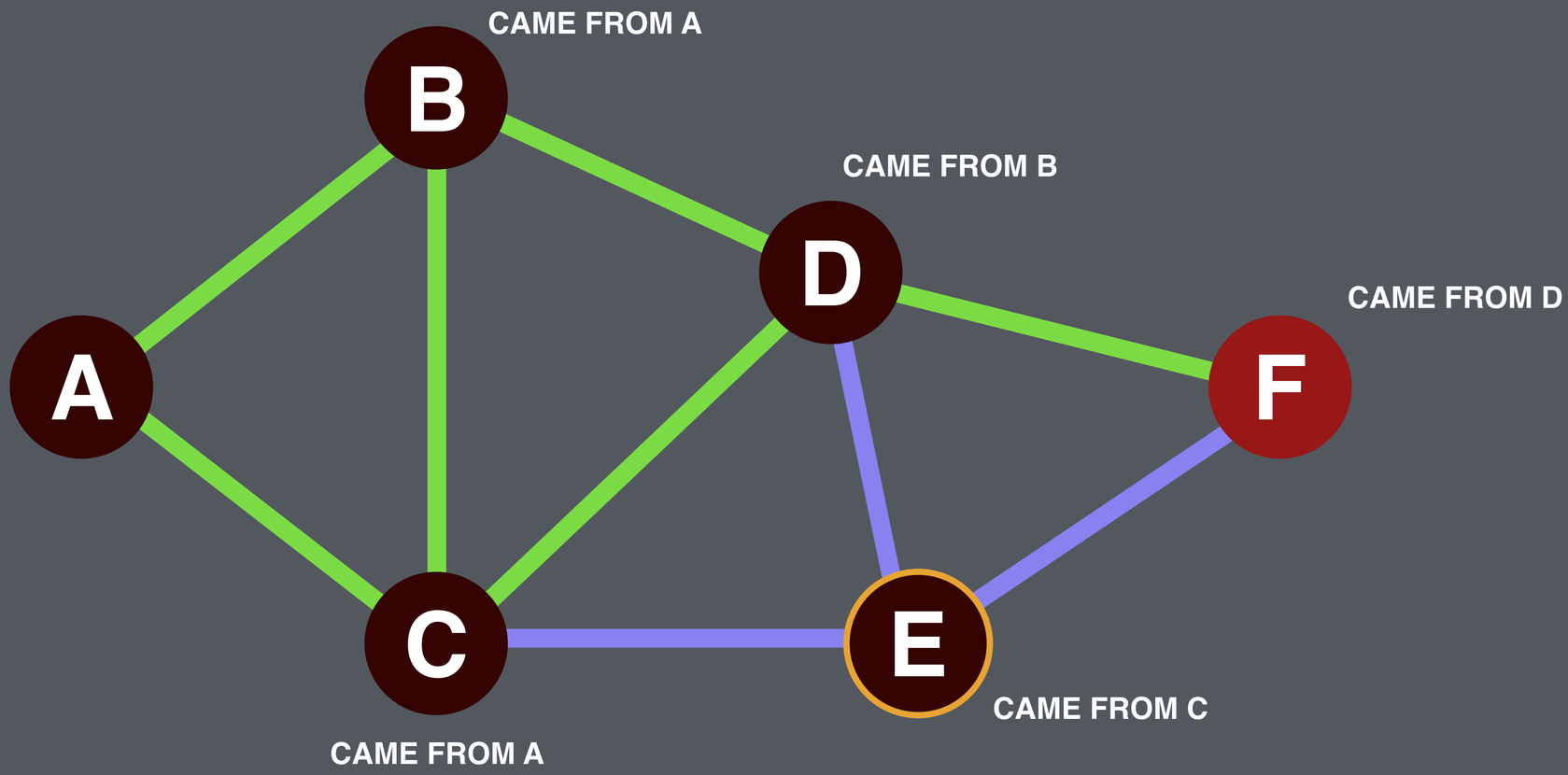
QUEUE: D,E



While queue has items in it, pop the front queue item, add its connected nodes to the queue if they haven’t been visited, mark them visited and set their “cameFrom” node to the node we popped from the queue.

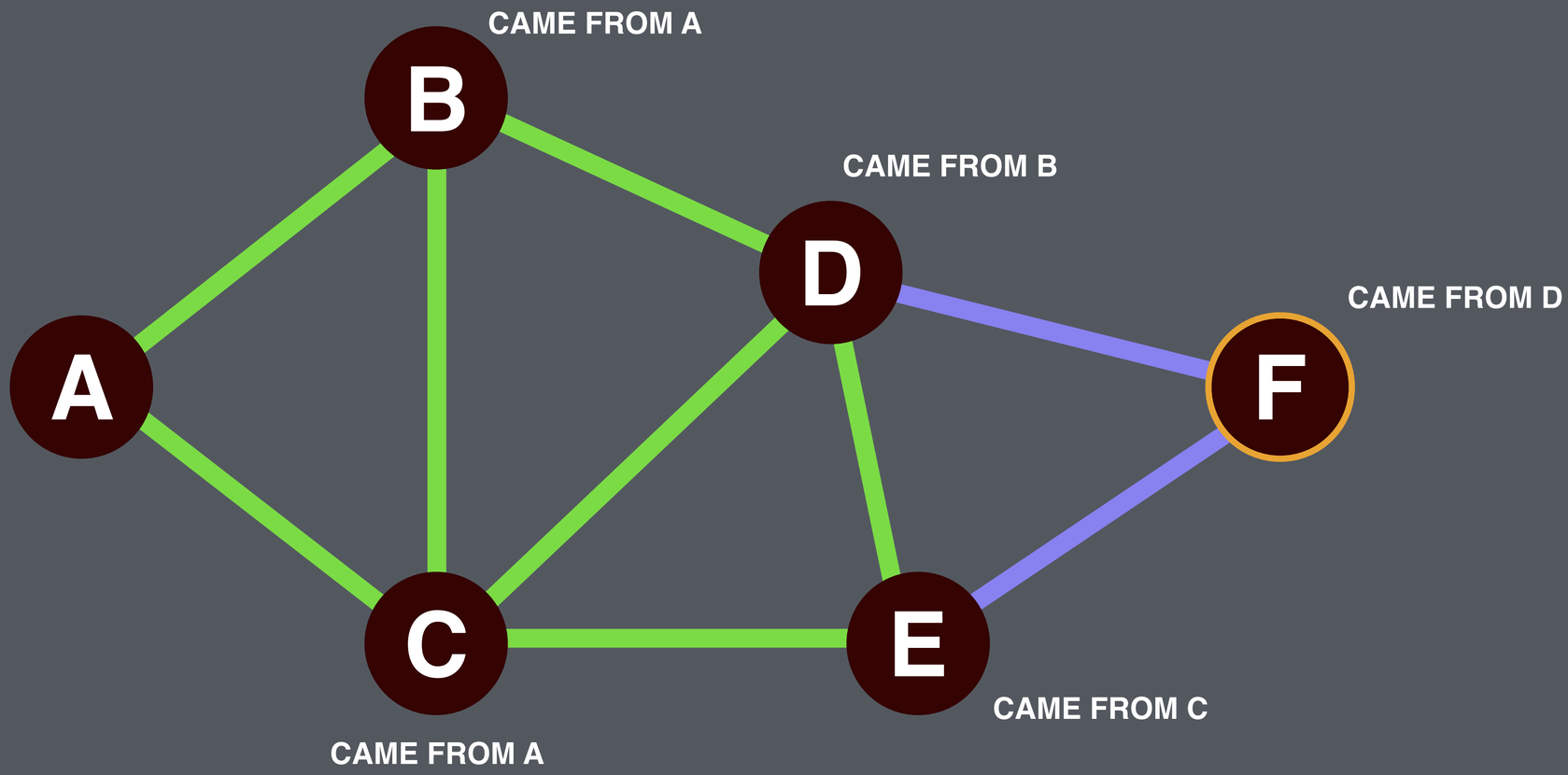
CURENT NODE: D

QUEUE: E,F



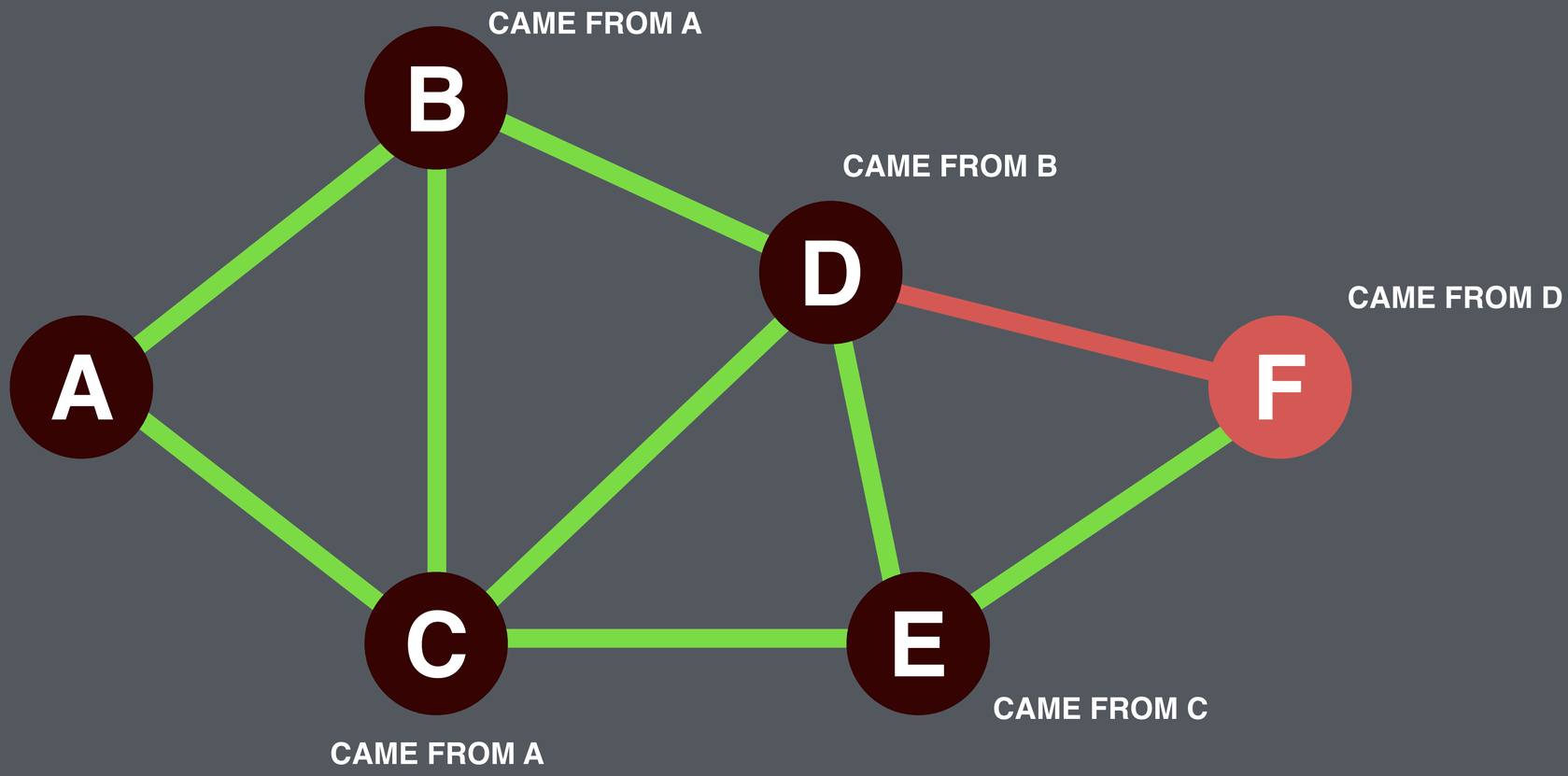
While queue has items in it, pop the front queue item, add its connected nodes to the queue if they haven’t been visited, mark them visited and set their “cameFrom” node to the node we popped from the queue.

QUEUE: F



While queue has items in it, pop the front queue item, add its connected nodes to the queue if they haven’t been visited, mark them visited and set their “cameFrom” node to the node we popped from the queue.

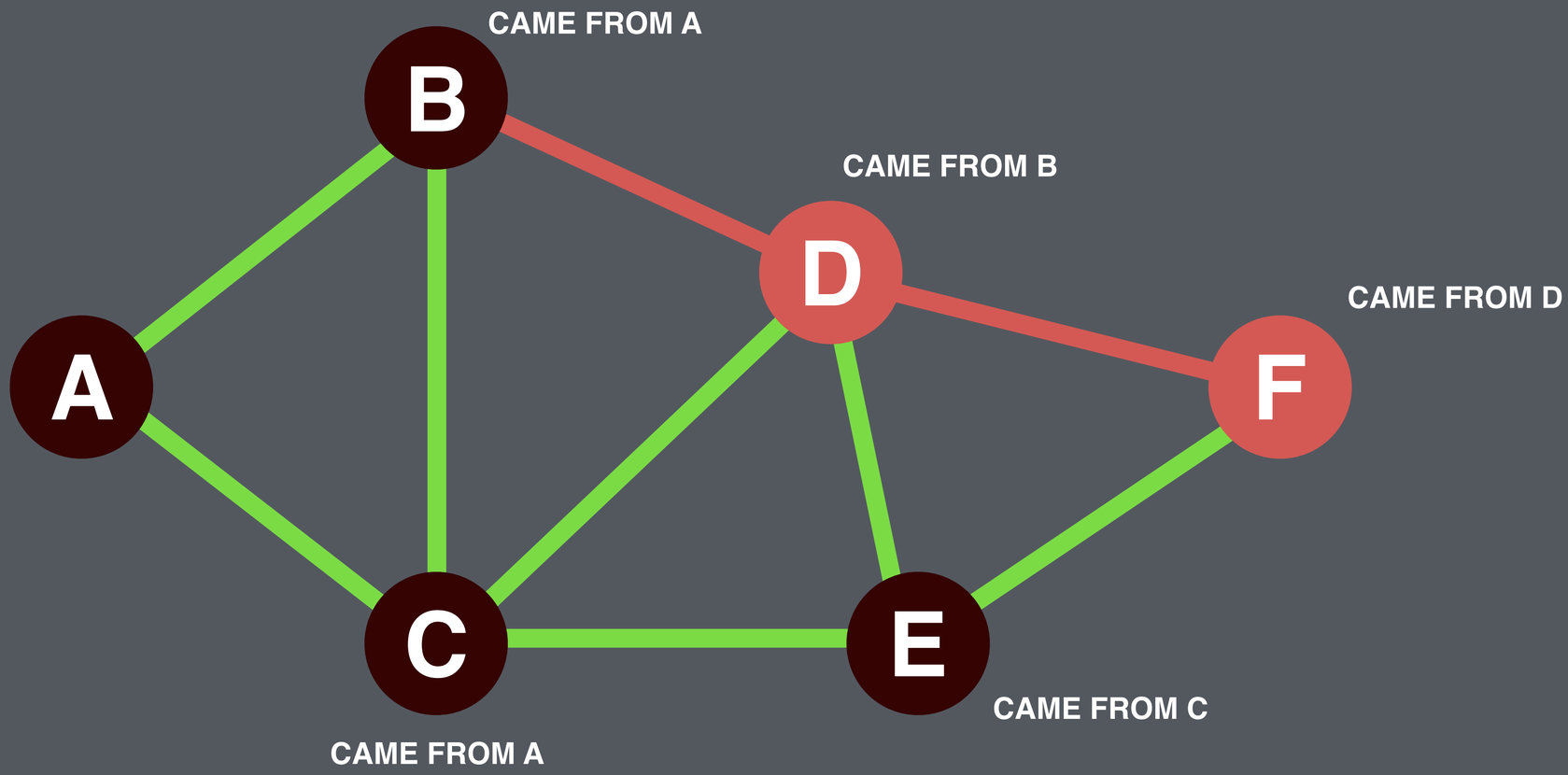
QUEUE:



Now, starting at the goal node (F), look at the cameFrom nodes, adding them to a list of nodes until we hit the starting node.

CURENT NODE: F CAME FROM: D

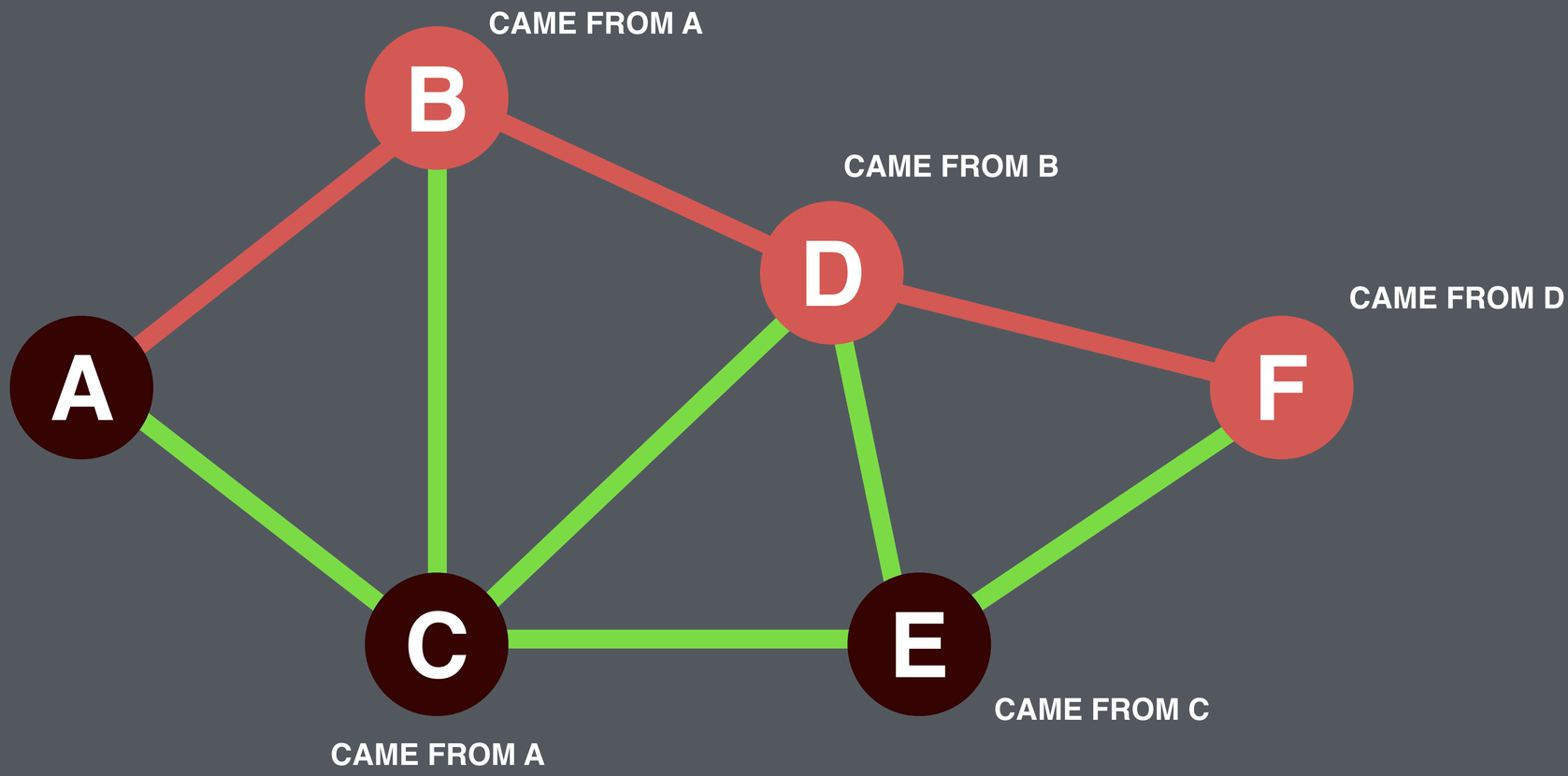
PATH: F



Now, starting at the goal node (F), look at the cameFrom nodes, adding them to a list of nodes until we hit the starting node.

CURENT NODE: D CAME FROM: B

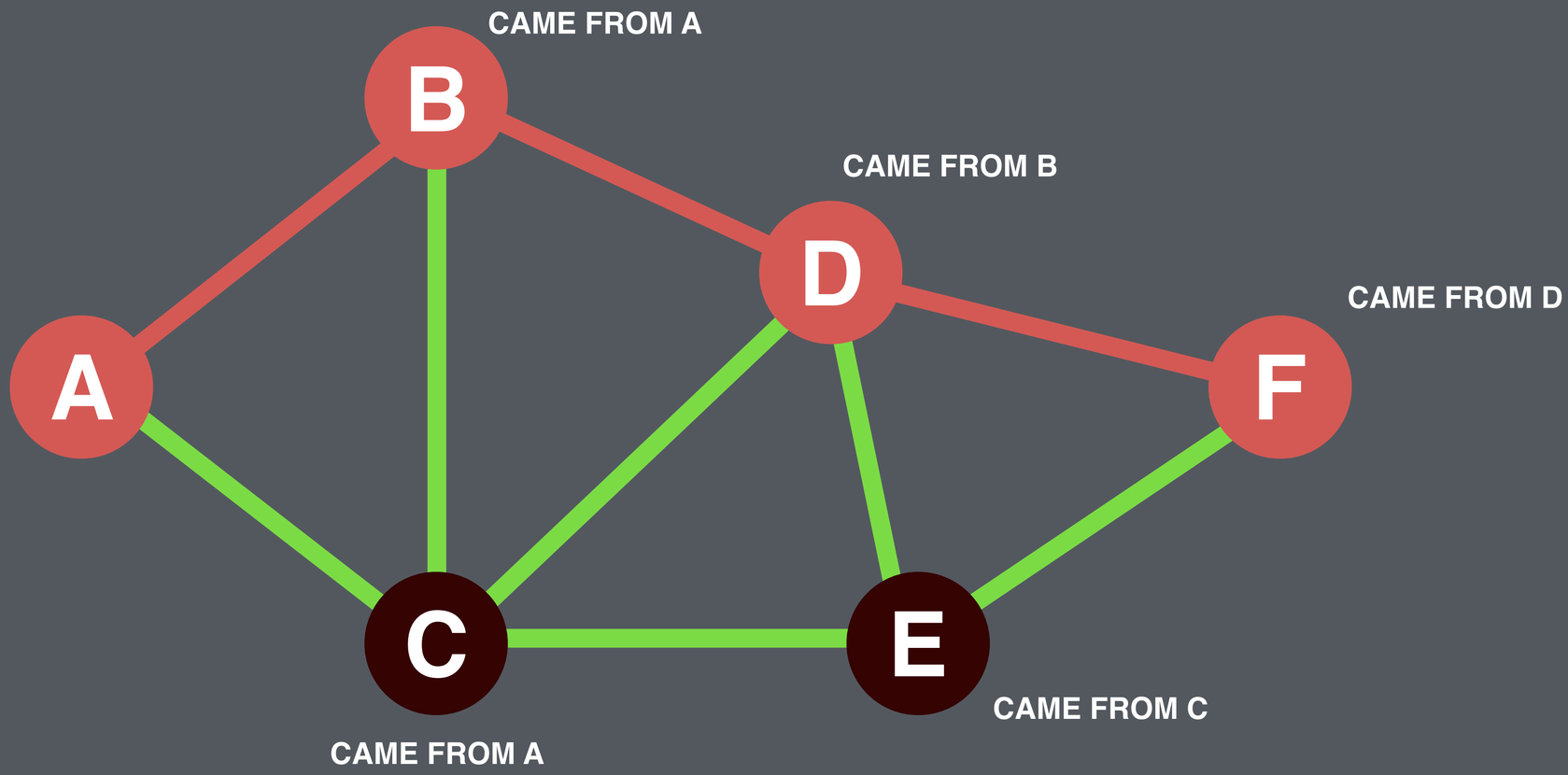
PATH: F,D



Now, starting at the goal node (F), look at the cameFrom nodes, adding them to a list of nodes until we hit the starting node.

CURENT NODE: B CAME FROM: A

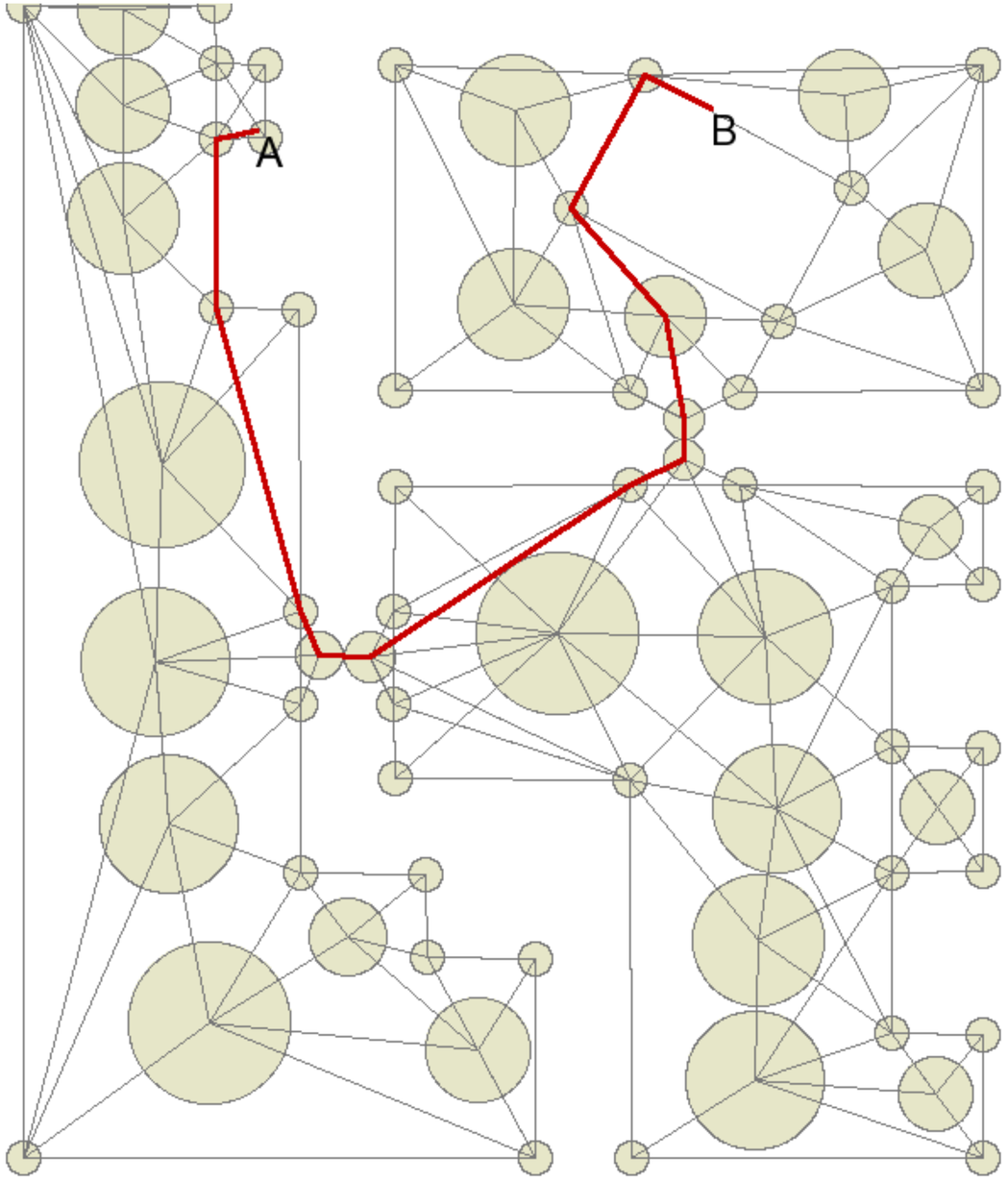
PATH: F,D,B

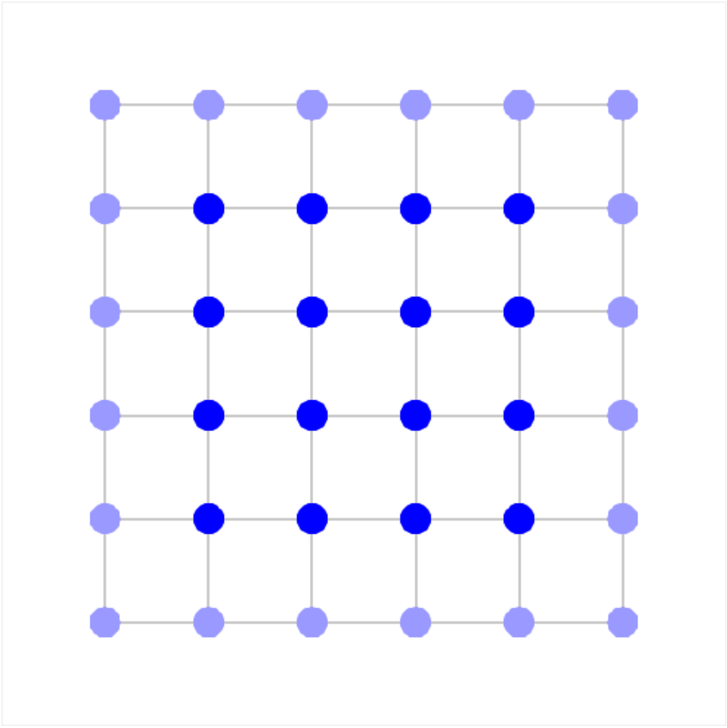


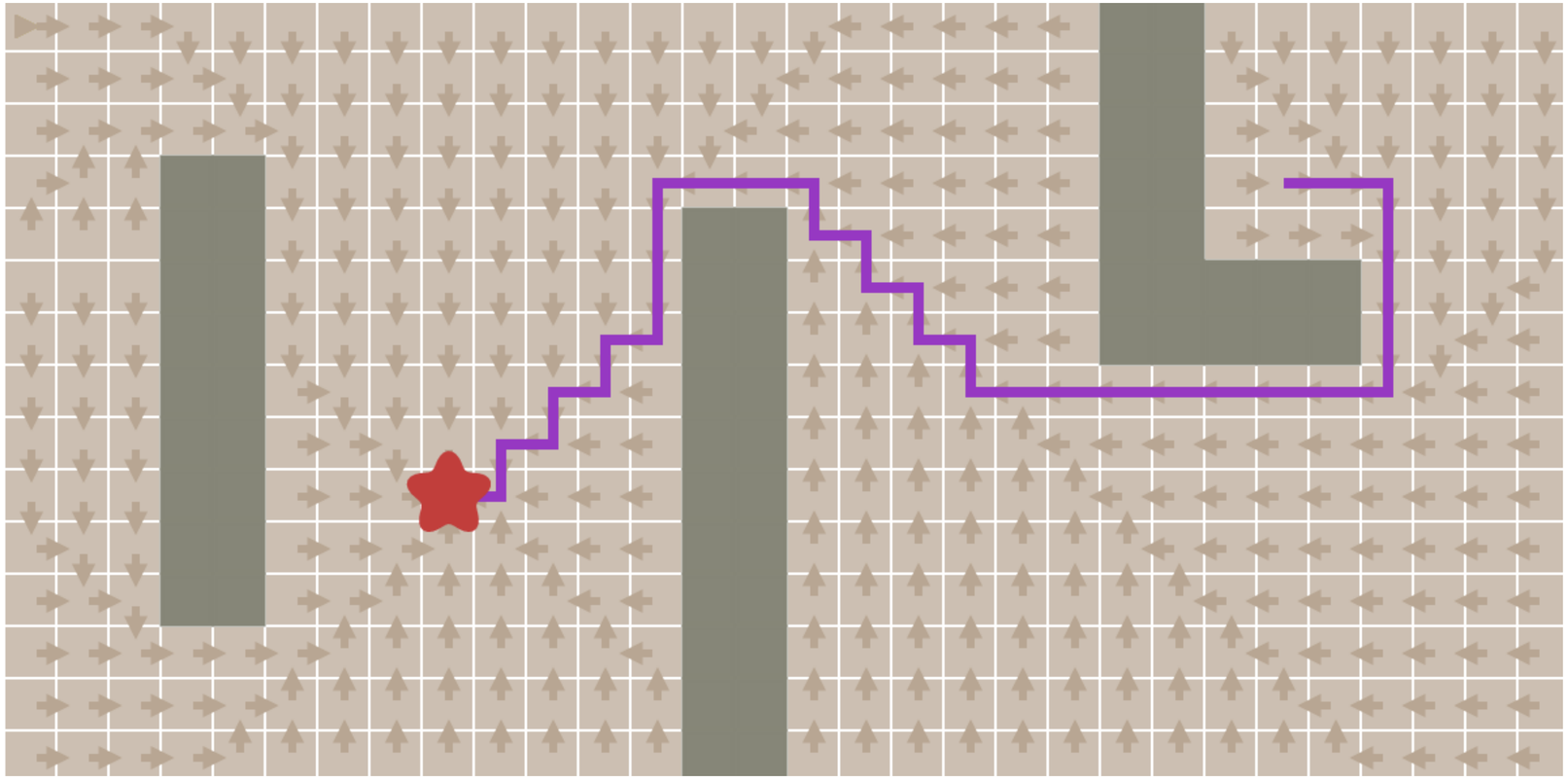
Now, starting at the goal node (F), look at the cameFrom nodes, adding them to a list of nodes until we hit the starting node.

CURENT NODE: A A IS OUR STARTING NODE!

PATH: F,D,B,A

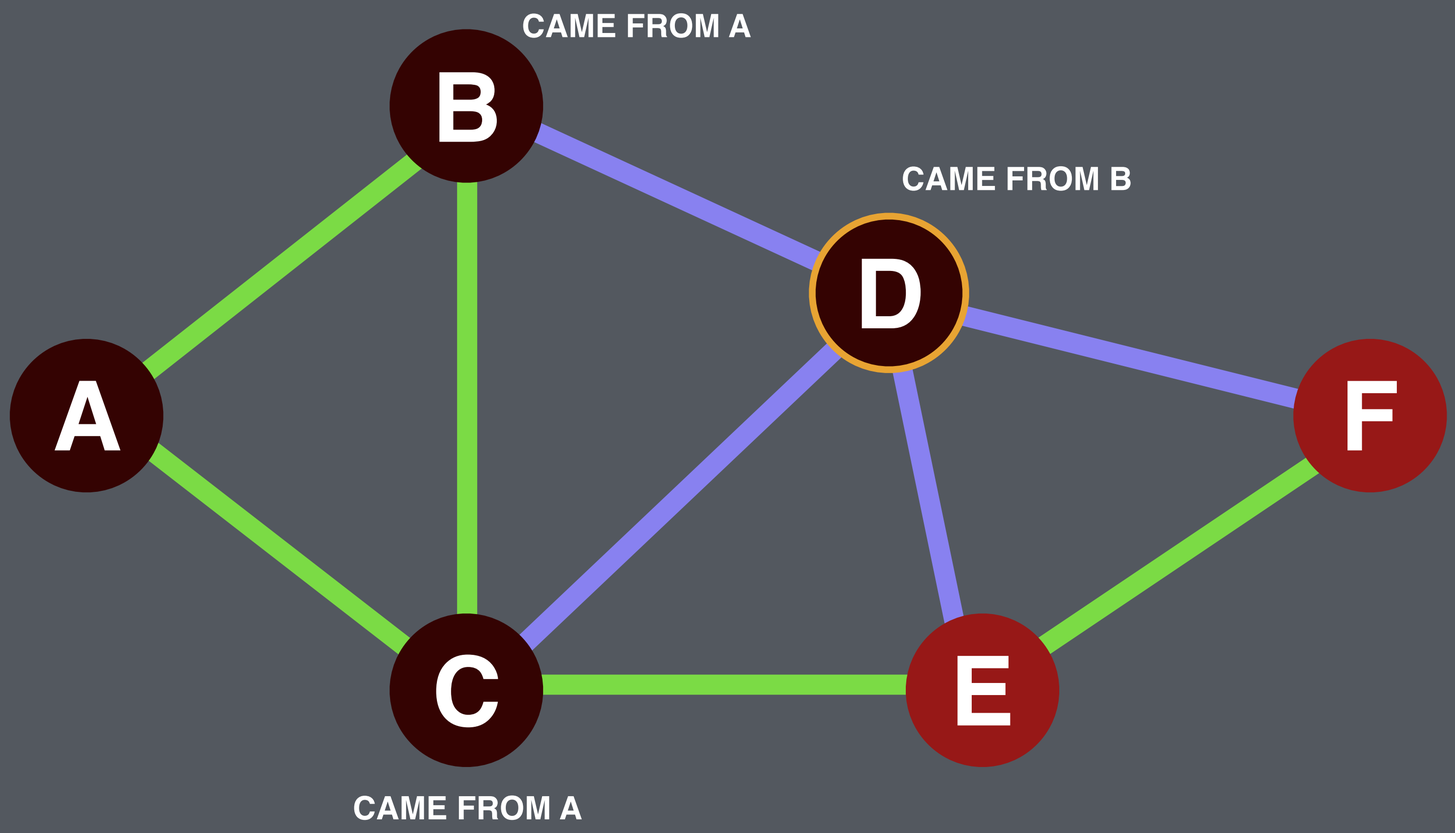

A
B



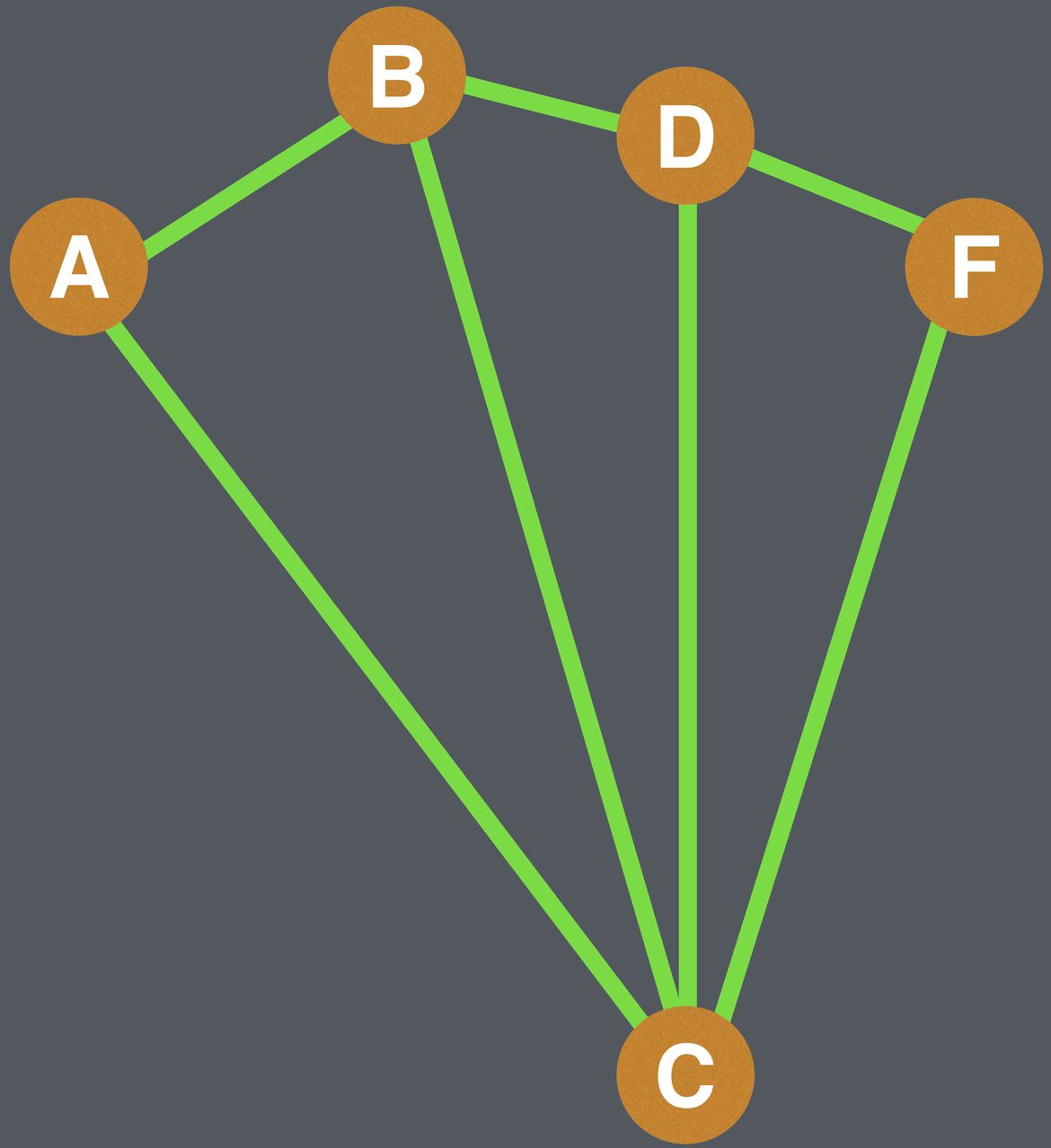


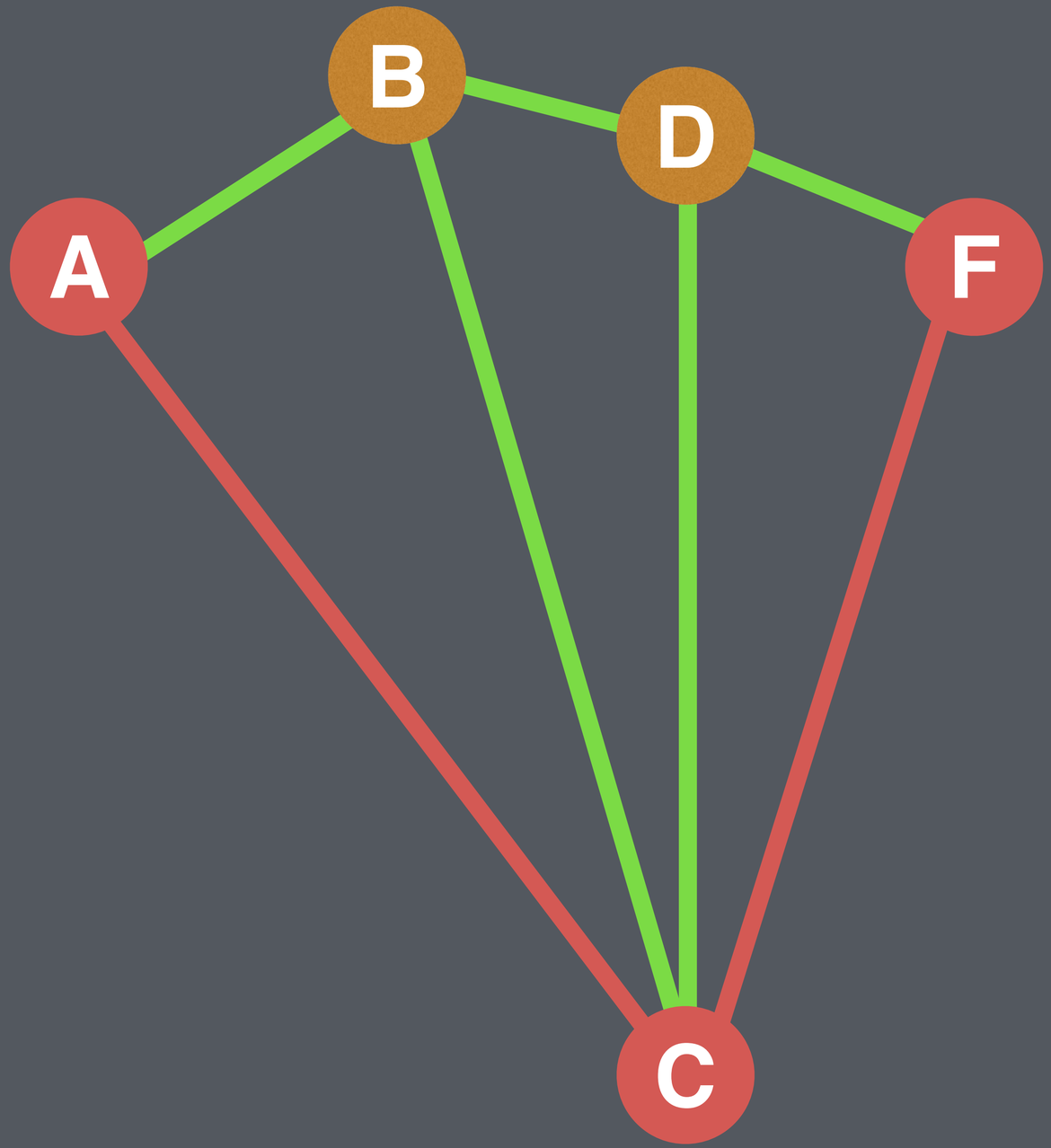
We can optimize our search by stopping cameFrom generation when we hit our goal node.

Path from A to D:

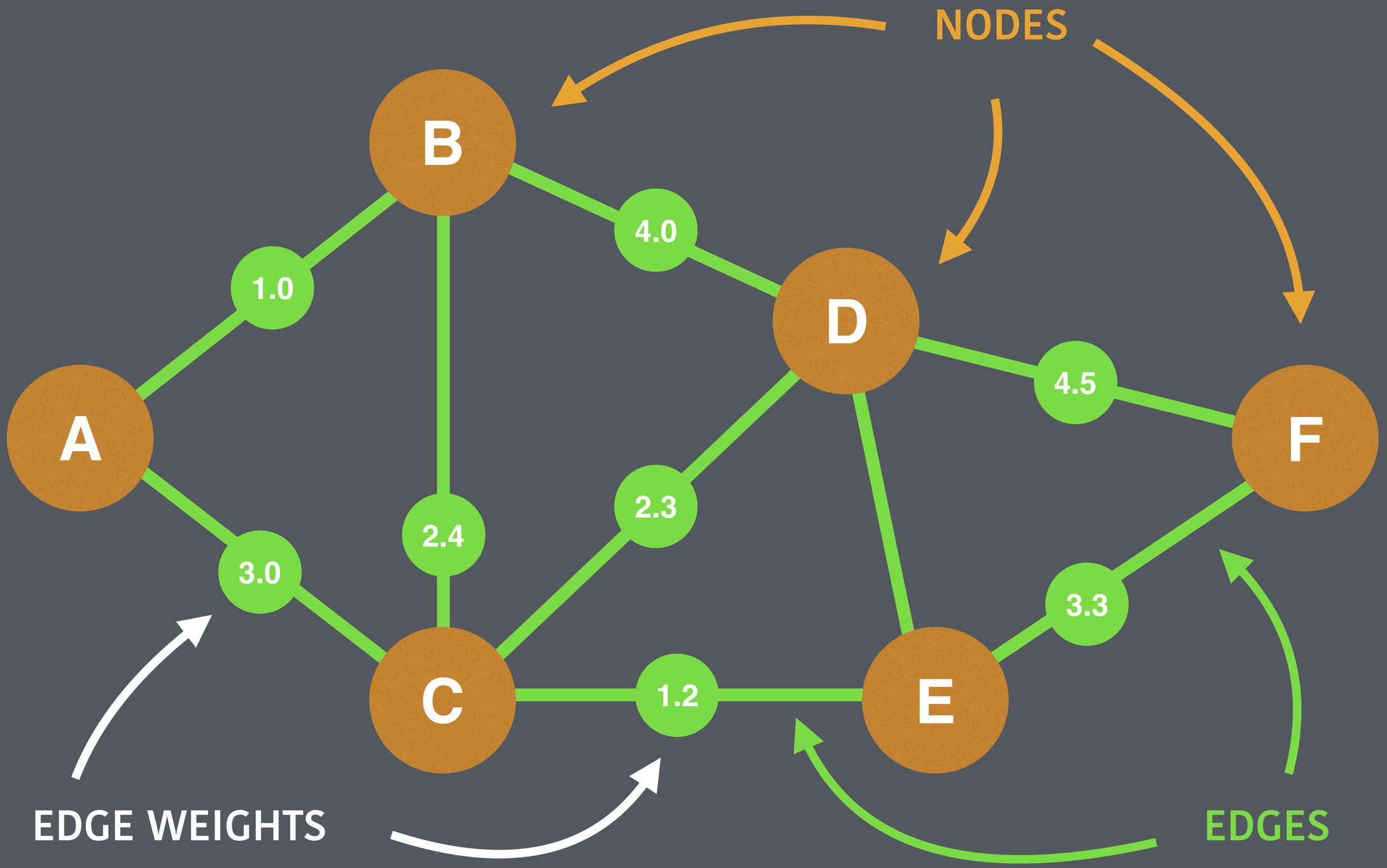


Distance

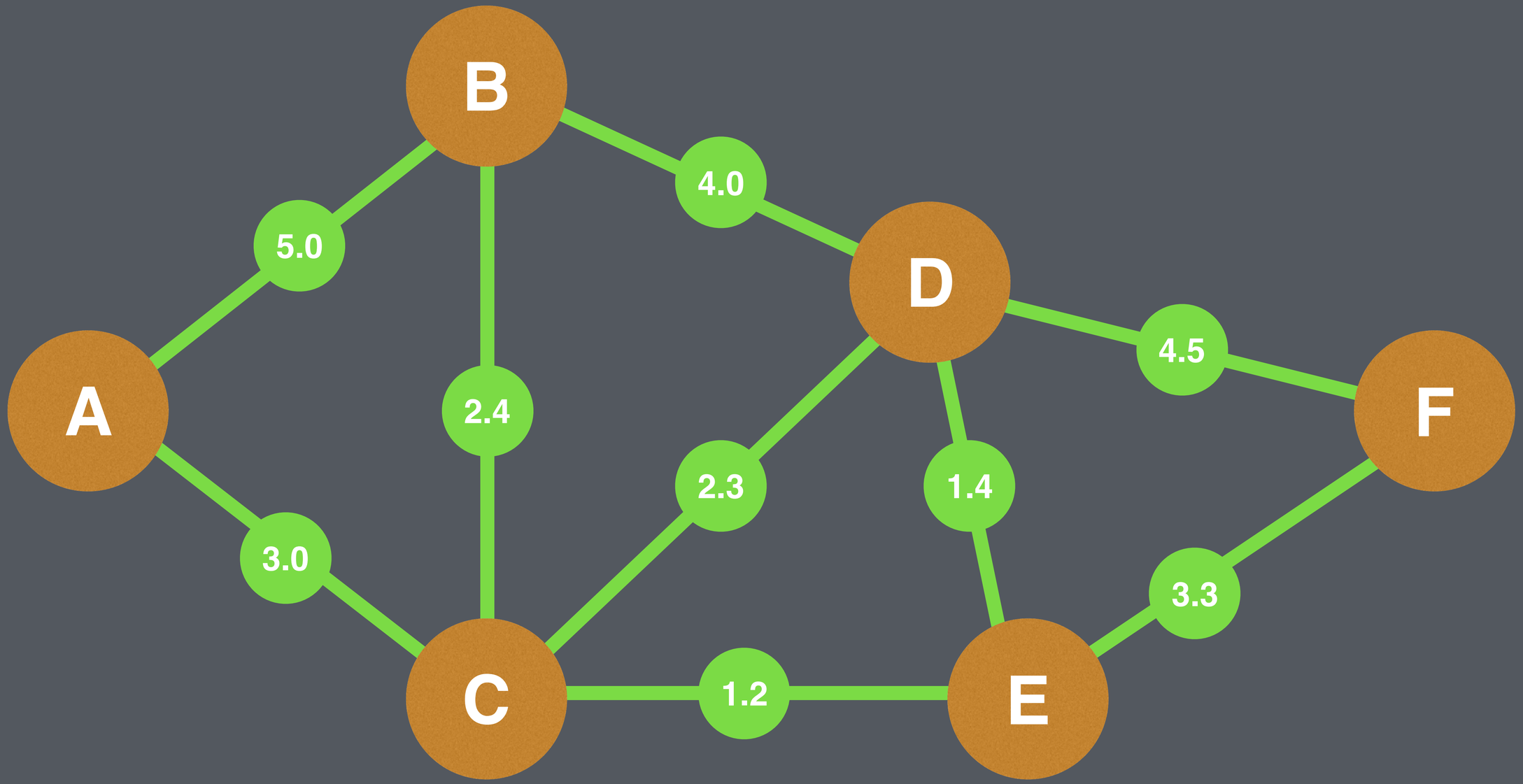

B
D
A
F
C


B
D
A
F
C

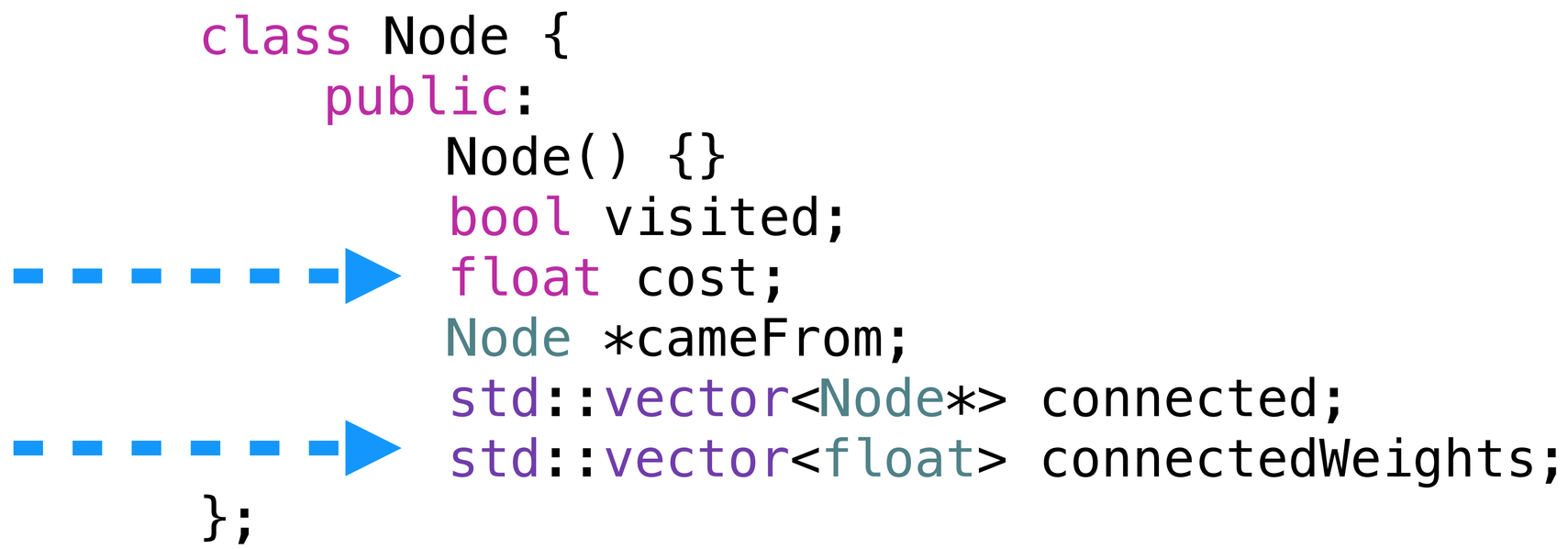
Weighted graphs


NODES
B
A
D
F
C
E
1.0
4.0
4.5
2.3
2.4
3.0
3.3
1.2
EDGE WEIGHTS
EDGES

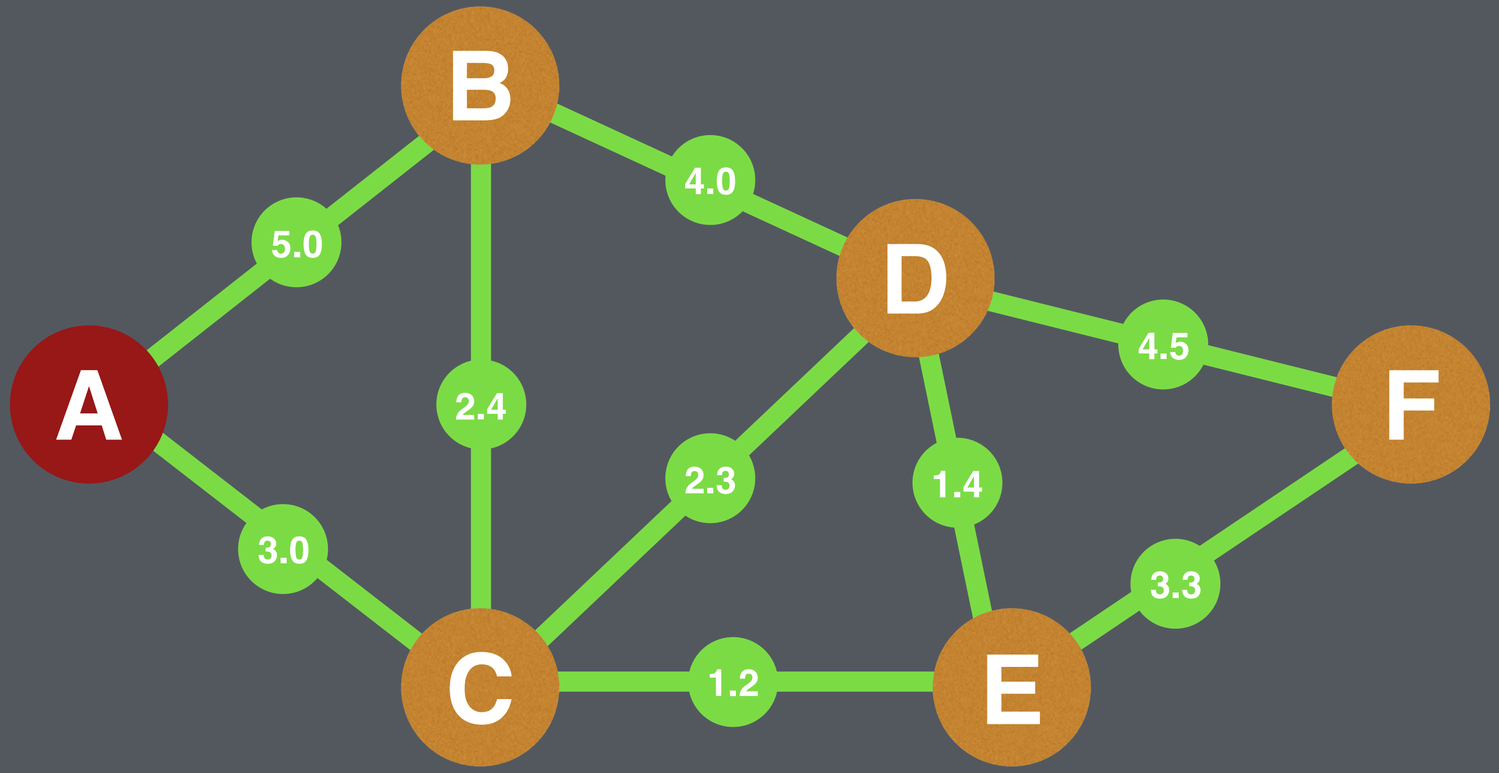
AI path nodes as weighted graphs.


B
4.0
5.0
D
4.5
A
2.4
F
2.3
1.4
3.0
3.3
C
1.2
E

Dijkstra’s algorithm.

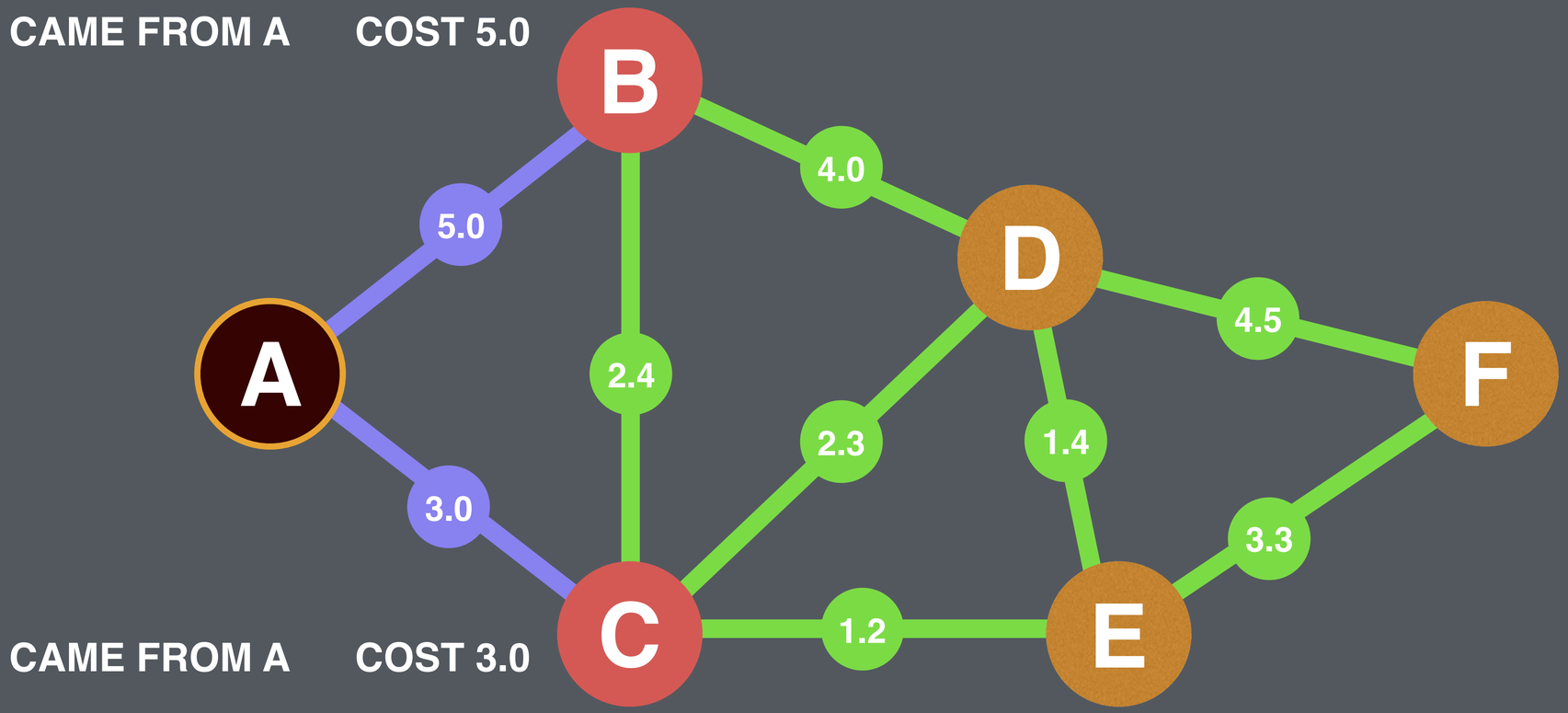

class Node {
public:
Node() {}
bool visited;
float cost;
Node *cameFrom;
std::vector<Node*> connected;
std::vector<float> connectedWeights;
};

Priority queue



Add our starting node to the queue with cost 0

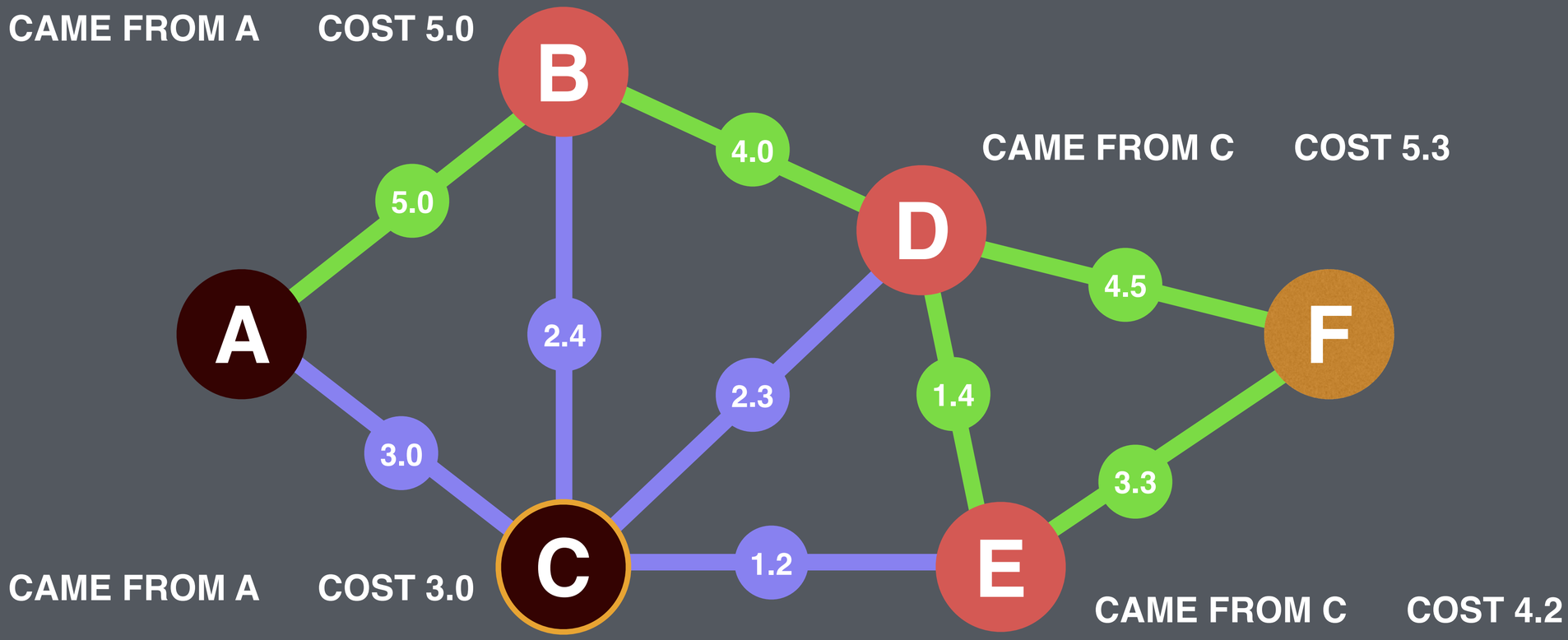
PRIORITY_QUEUE: A(0)



While queue has items in it, pop the front queue item, add its connected nodes to the queue if they haven't been visited OR IF THE COST OF THE CURRENT NODE PLUS THE CONNECTION COST IS LESS THAN THE COST OF THAT NODE, mark them visited, set their "cameFrom" node to the node we popped from the queue and SET THEIR COST TO THE CURRENT NODE COST + CONNECTION COST.

CURENT NODE: A

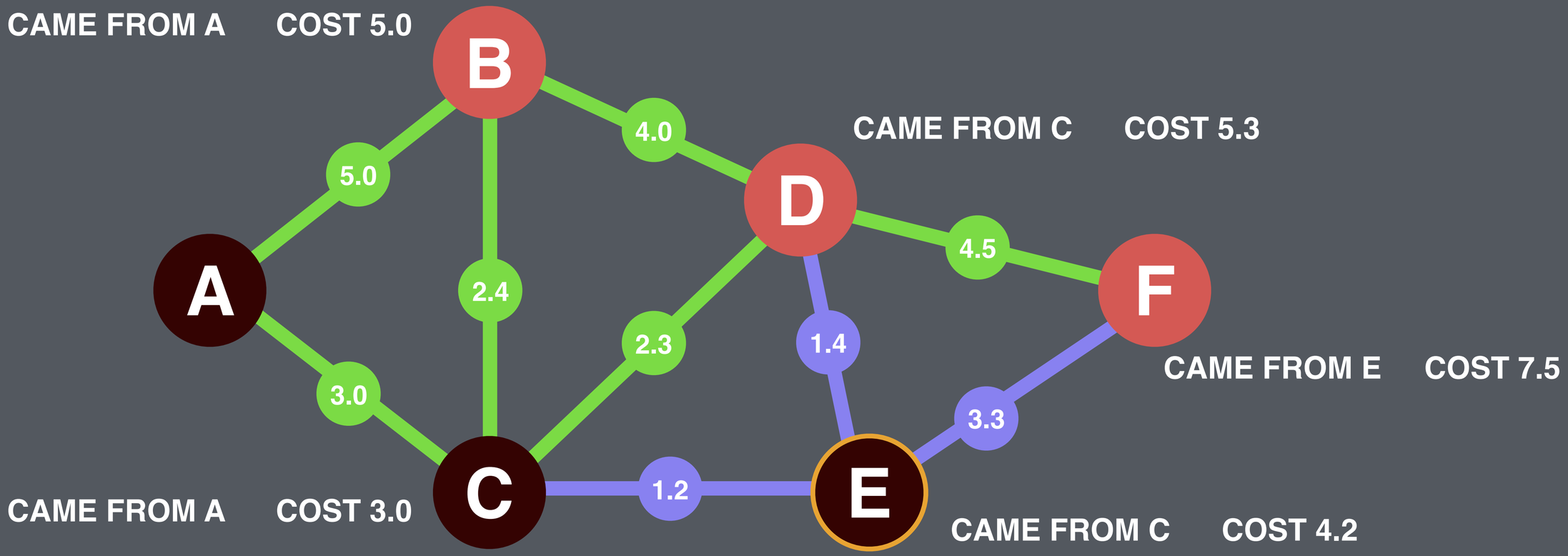
PRIORITY_QUEUE: C(3.0) B(5.0)



While queue has items in it, pop the front queue item, add its connected nodes to the queue if they haven't been visited OR IF THE COST OF THE CURRENT NODE PLUS THE CONNECTION COST IS LESS THAN THE COST OF THAT NODE, mark them visited, set their "cameFrom" node to the node we popped from the queue and SET THEIR COST TO THE CURRENT NODE COST + CONNECTION COST.

CURENT NODE: C

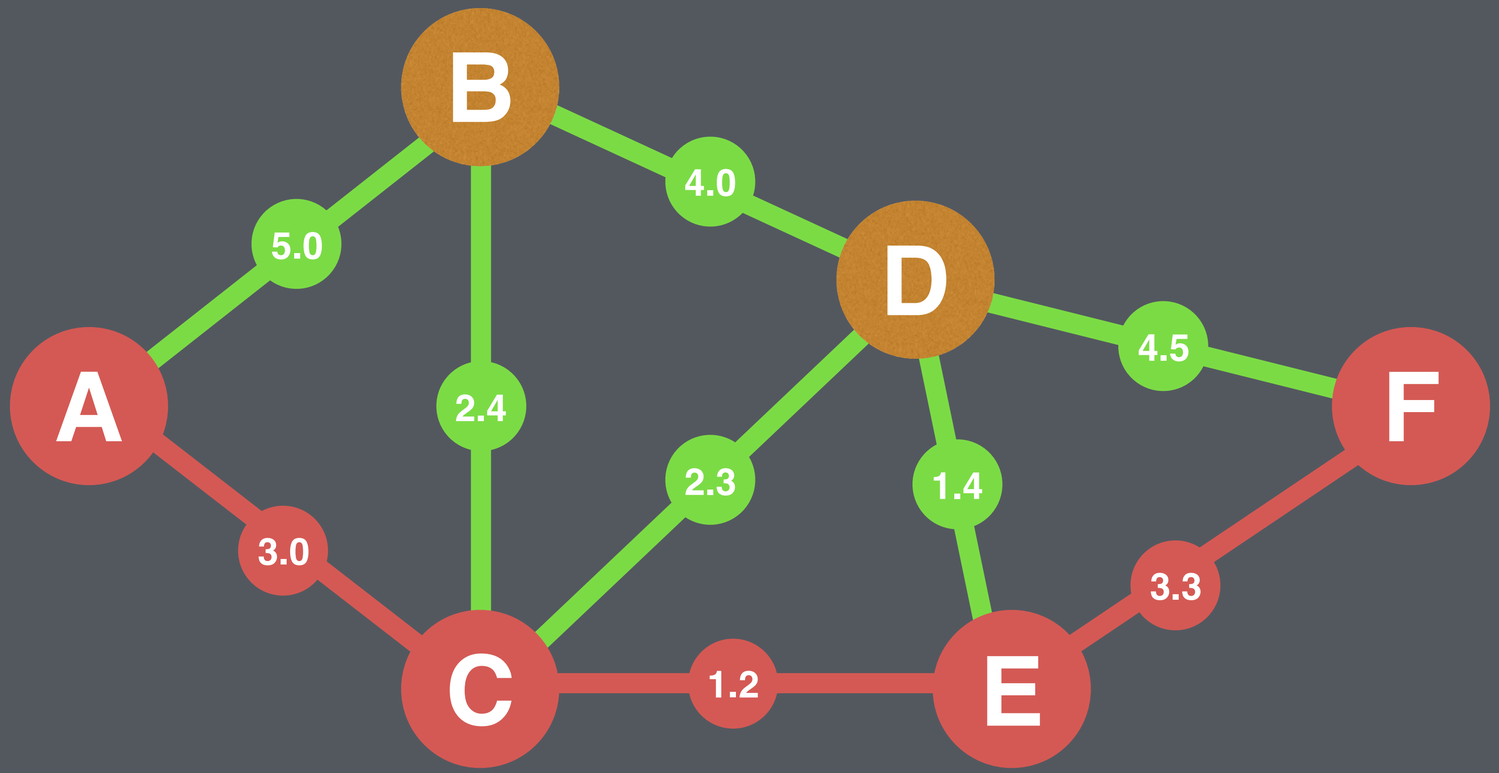
PRIORITY_QUEUE: E(4.2) B(5.0) D(5.3)

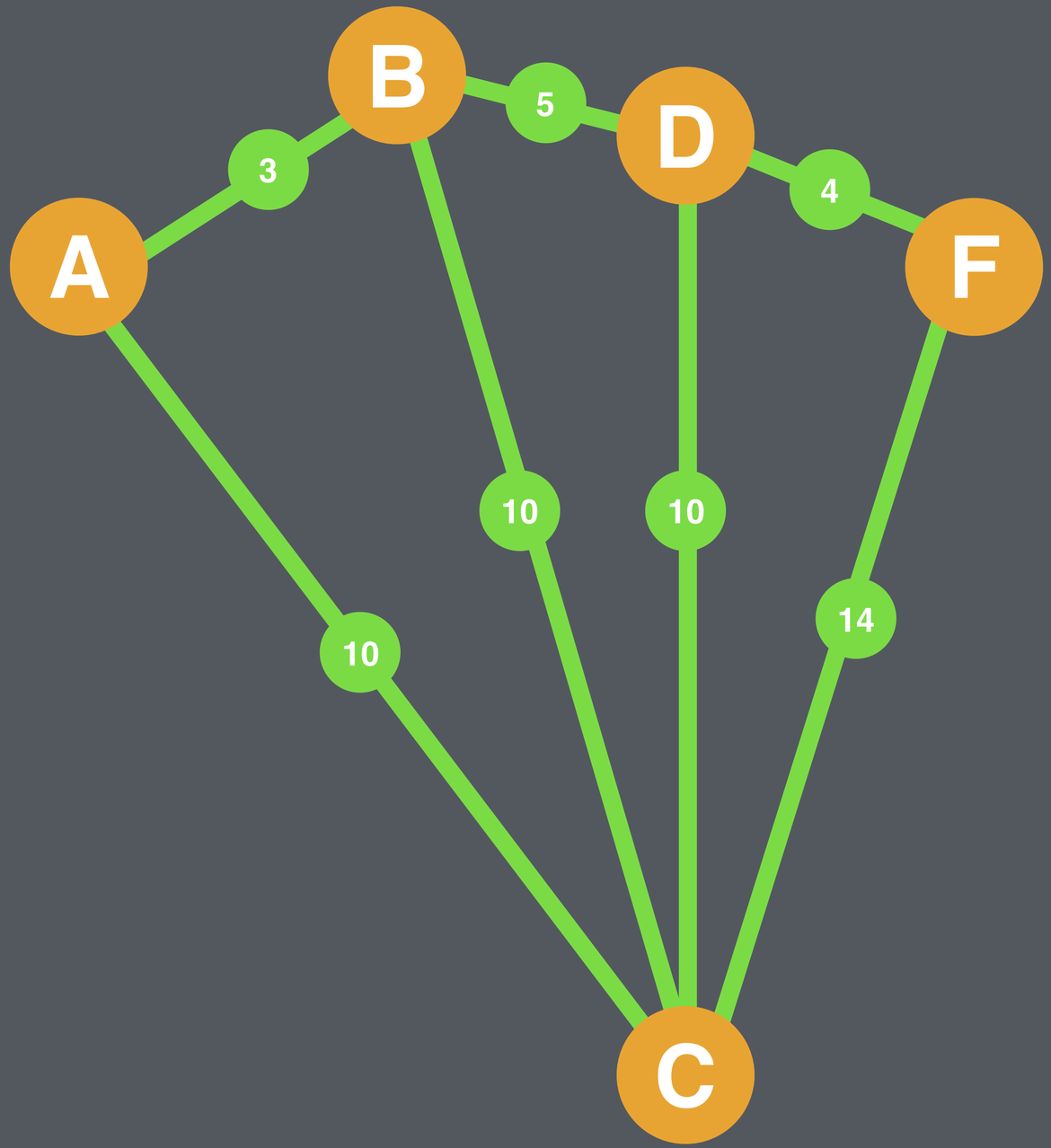


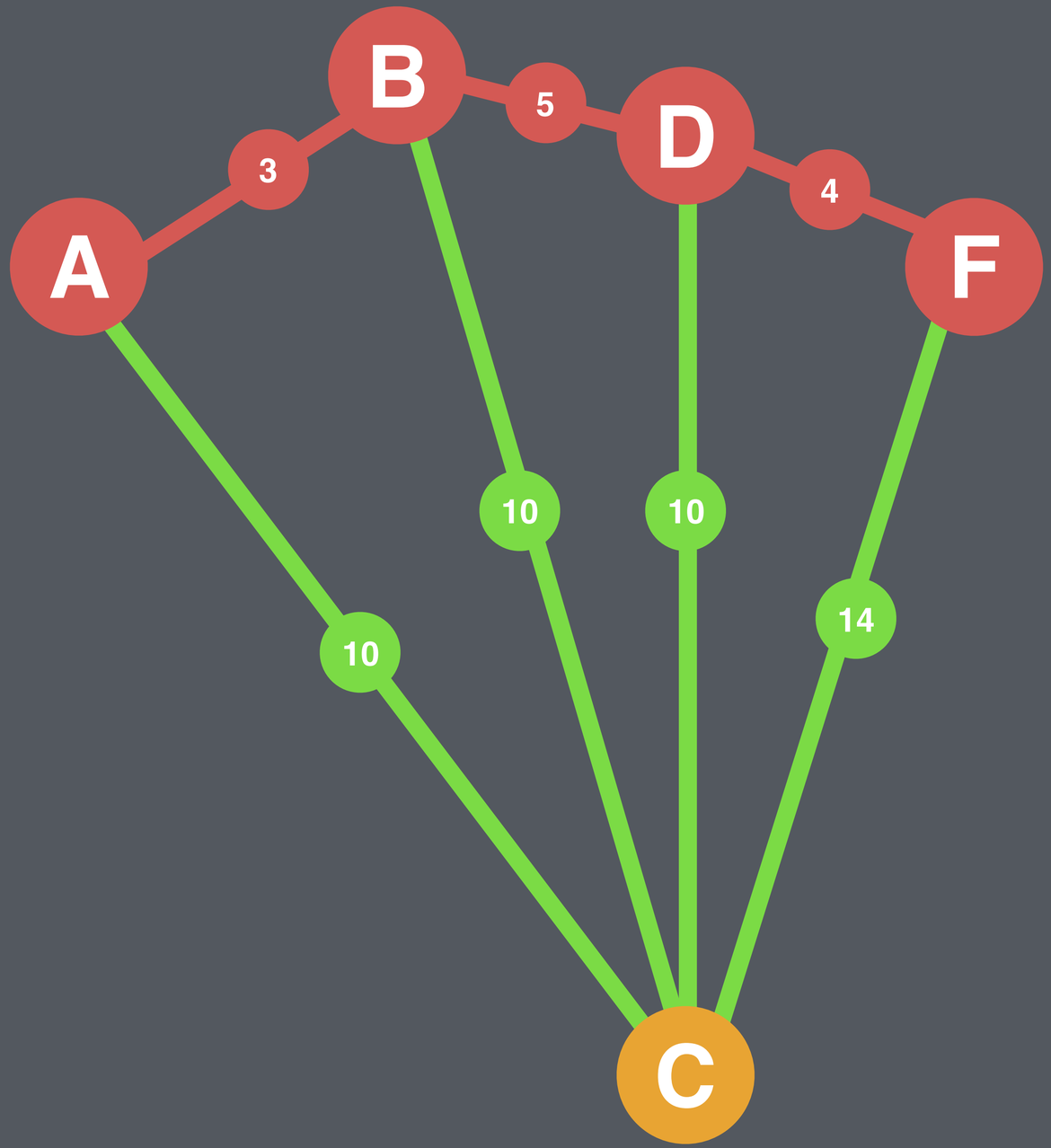
While queue has items in it, pop the front queue item, add its connected nodes to the queue if they haven't been visited OR IF THE COST OF THE CURRENT NODE PLUS THE CONNECTION COST IS LESS THAN THE COST OF THAT NODE, mark them visited, set their "cameFrom" node to the node we popped from the queue and SET THEIR COST TO THE CURRENT NODE COST + CONNECTION COST.

CURENT NODE: E

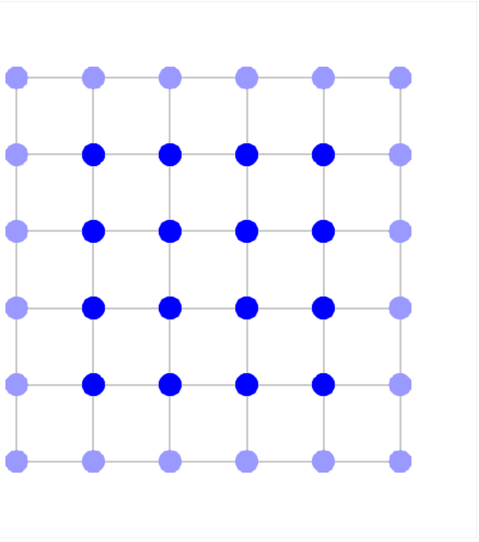
PRIORITY_QUEUE: B(5.0) D(5.3) F(7.5)


B
4.0
5.0
D
4.5
A
2.4
F
2.3
1.4
3.0
3.3
C
1.2
E

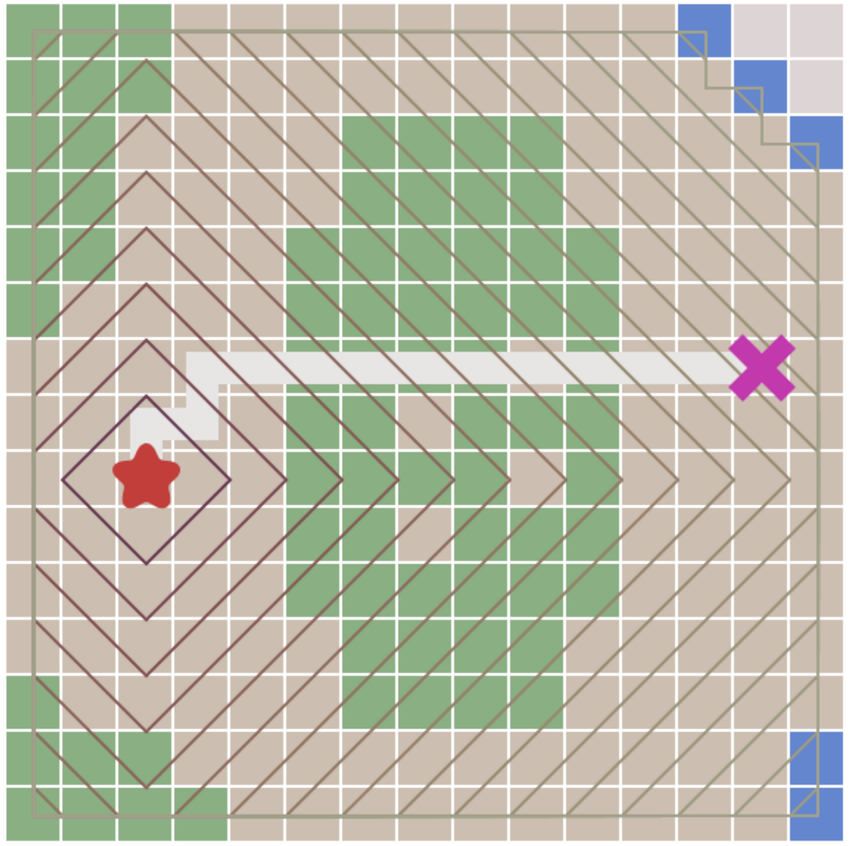

B
5
D
3
4
A
F
10
10
10
14
C


B
5
D
3
4
A
F
10
10
10
14
C

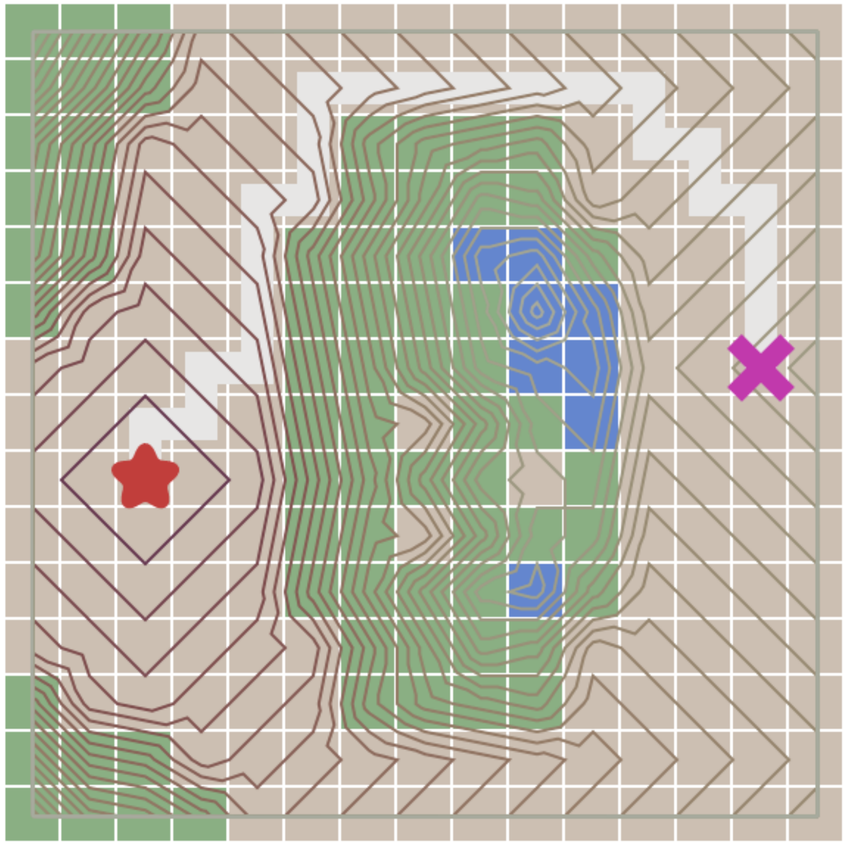

0
0
0
0
Bronze Age
Diplomacy
Menu
S
?



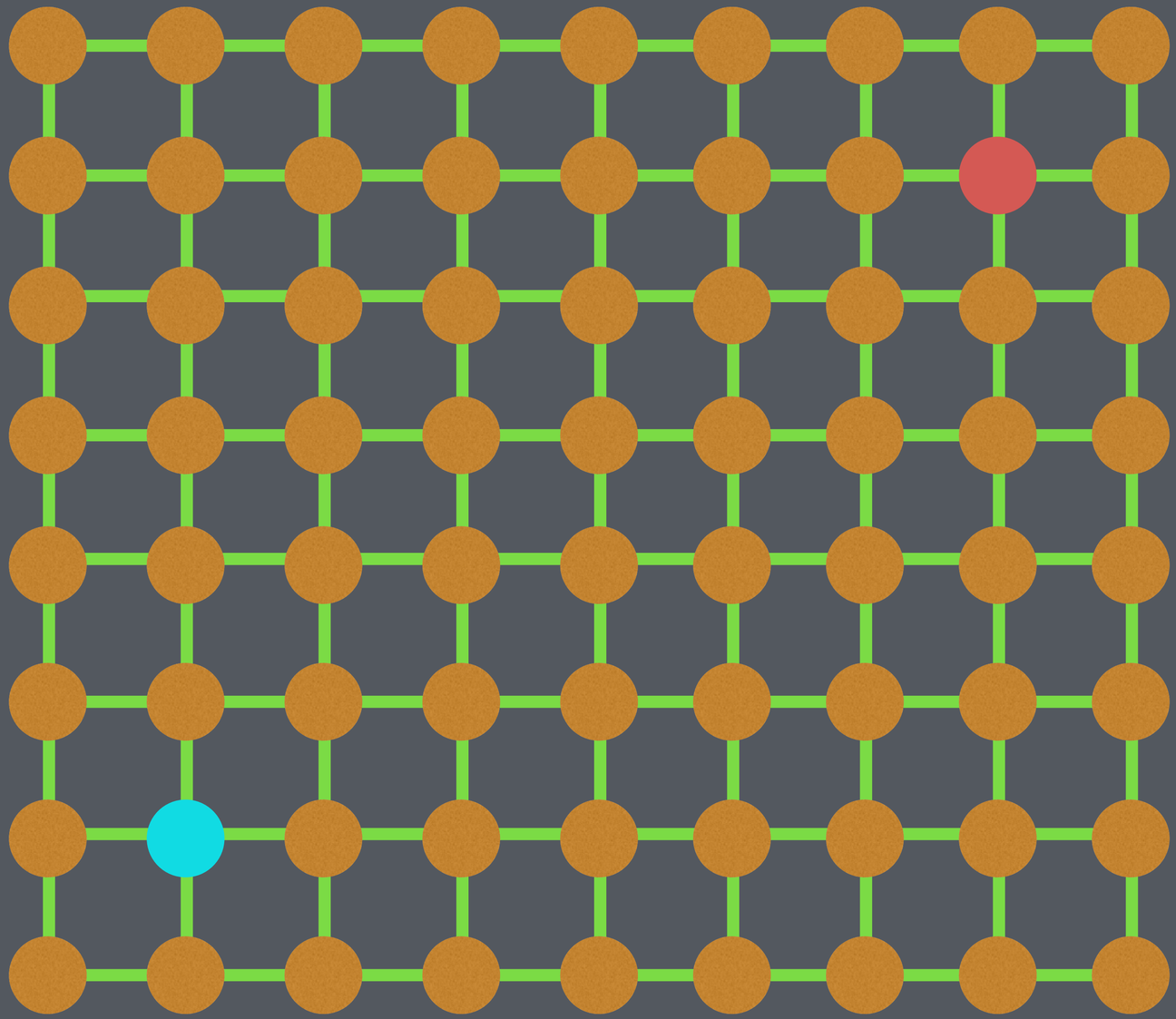
Breadth First Search



Dijkstra's Algorithm

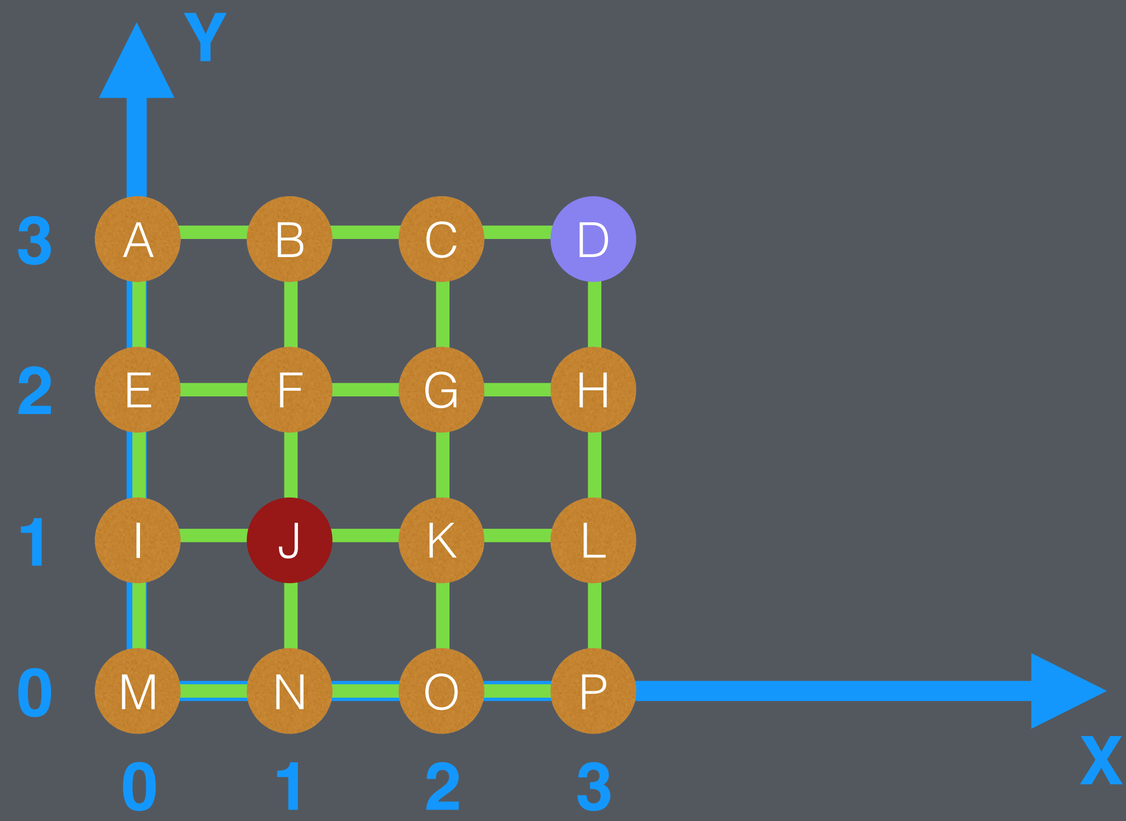


Optimizing using a heuristic.



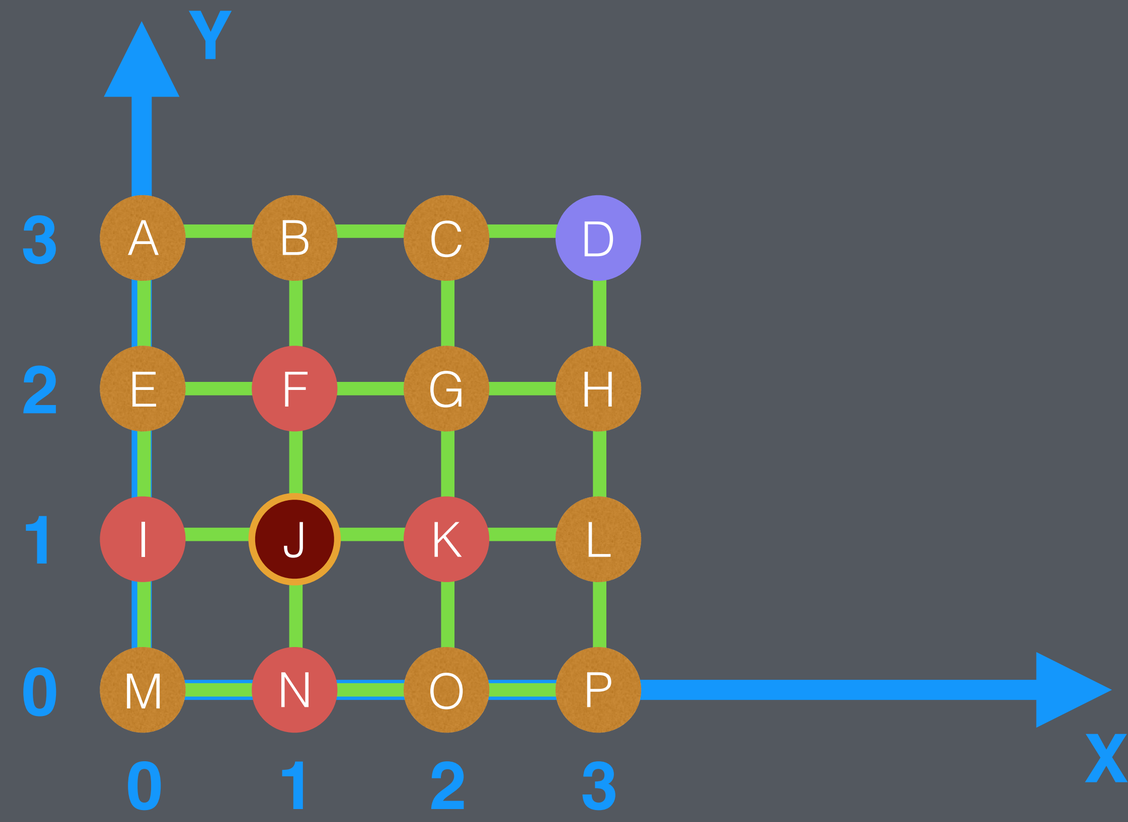
Greedy Best First Search

```
class Node {
    public:
        Node() {}
        bool visited;
        float distanceFromGoal;
        Node *cameFrom;
        std::vector<Node*> connected;
};
```



Add our starting node to the queue with priority 0

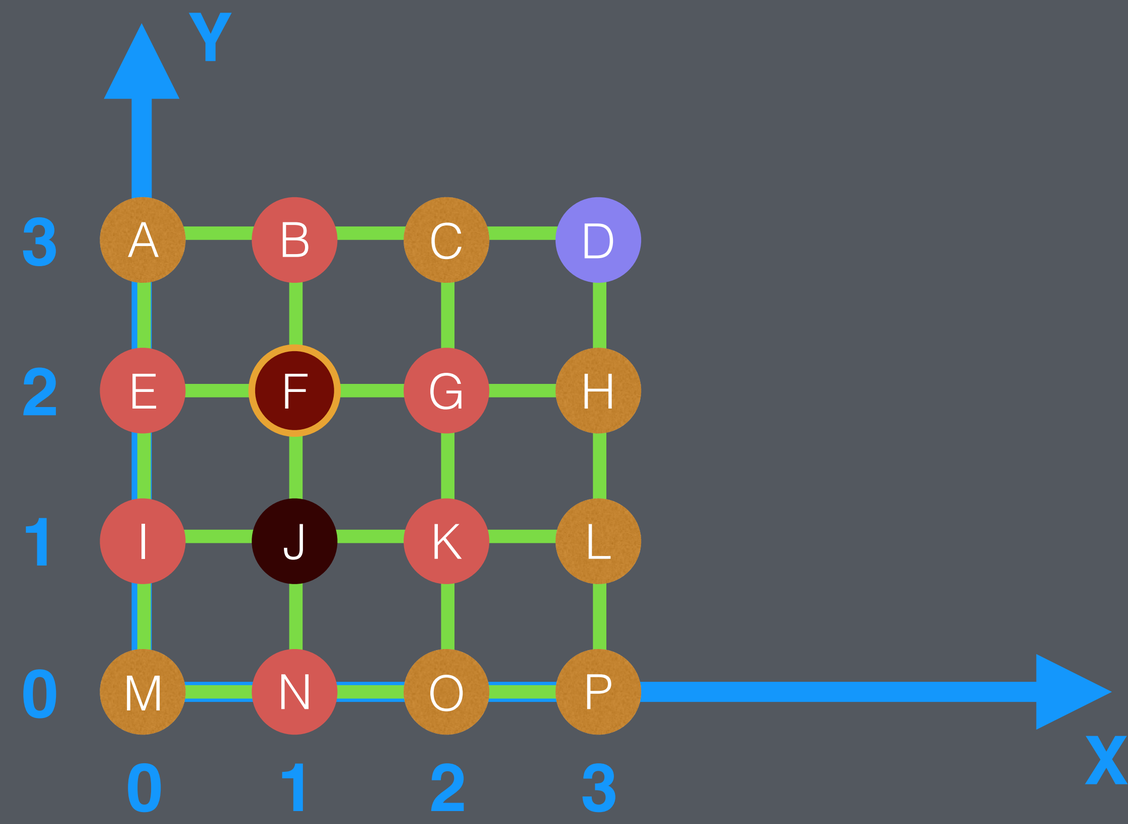
PRIORITY_QUEUE: J(0)



While queue has items in it, pop the front queue item, add its connected nodes to the queue (using their distance from the goal as queue priority) if they haven't been visited, mark them visited, set their "cameFrom" node to the node we popped from the queue.

CURENT NODE: J

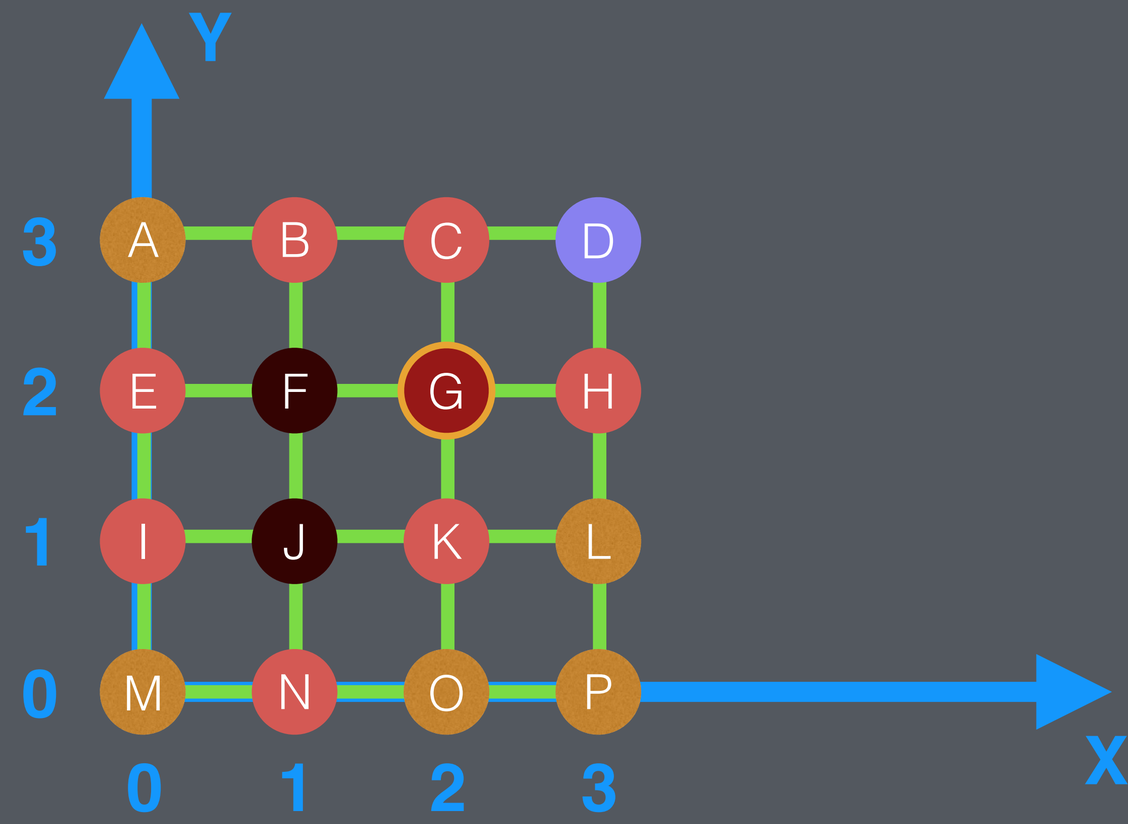
PRIORITY_QUEUE: F K I N



While queue has items in it, pop the front queue item, add its connected nodes to the queue (using their distance from the goal as queue priority) if they haven't been visited, mark them visited, set their "cameFrom" node to the node we popped from the queue.

CURENT NODE: F

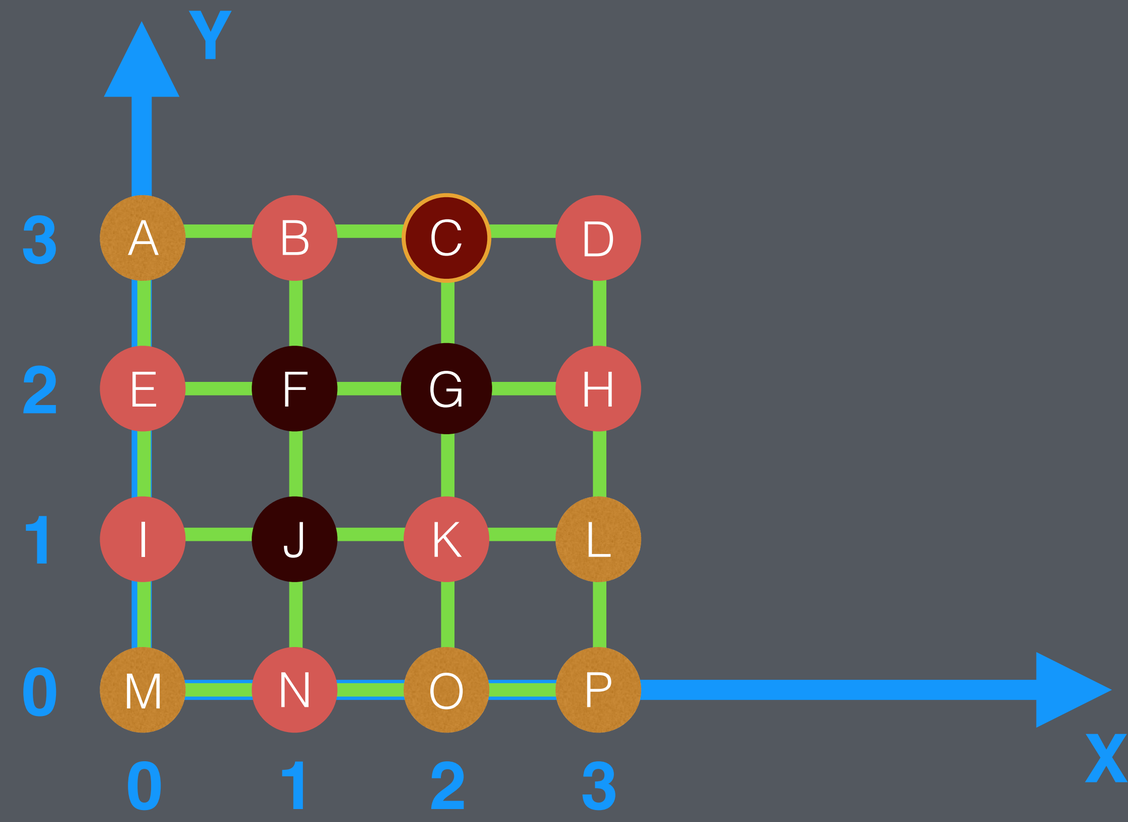
PRIORITY_QUEUE: G B K E I N



While queue has items in it, pop the front queue item, add its connected nodes to the queue (using their distance from the goal as queue priority) if they haven't been visited, mark them visited, set their "cameFrom" node to the node we popped from the queue.

CURENT NODE: G

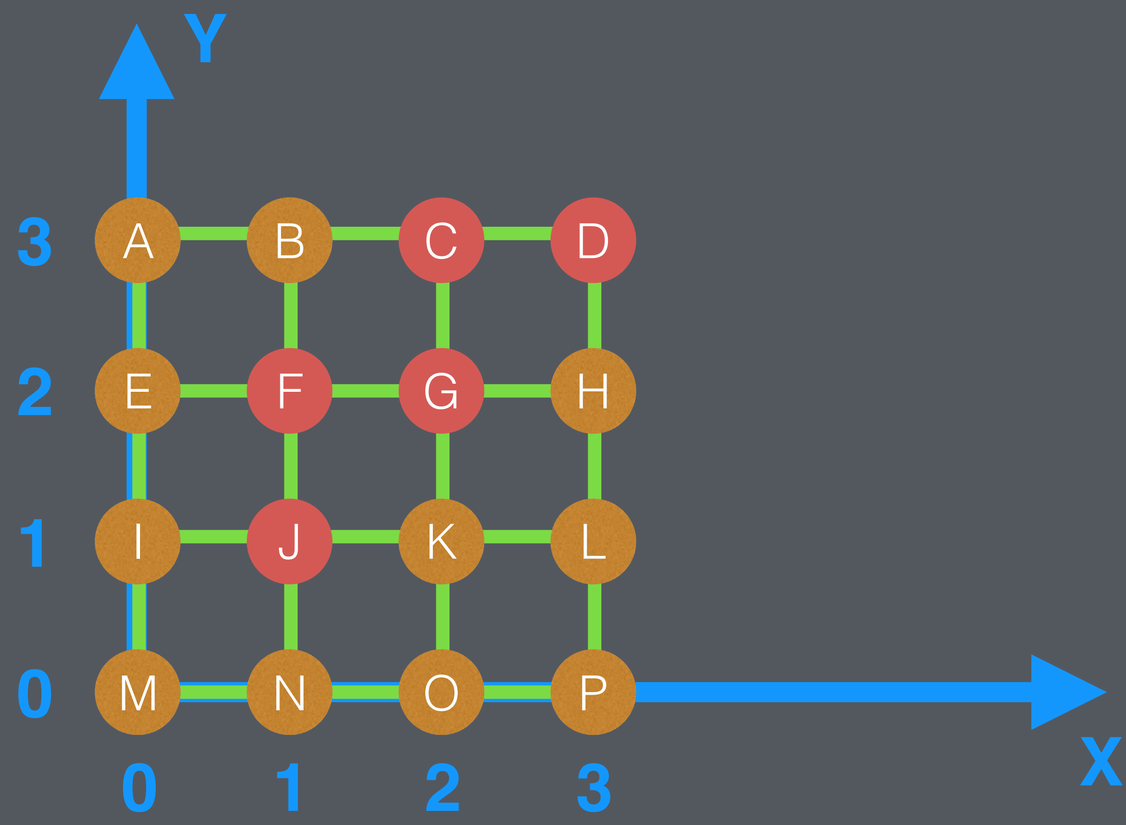
PRIORITY_QUEUE: C H B K E I N



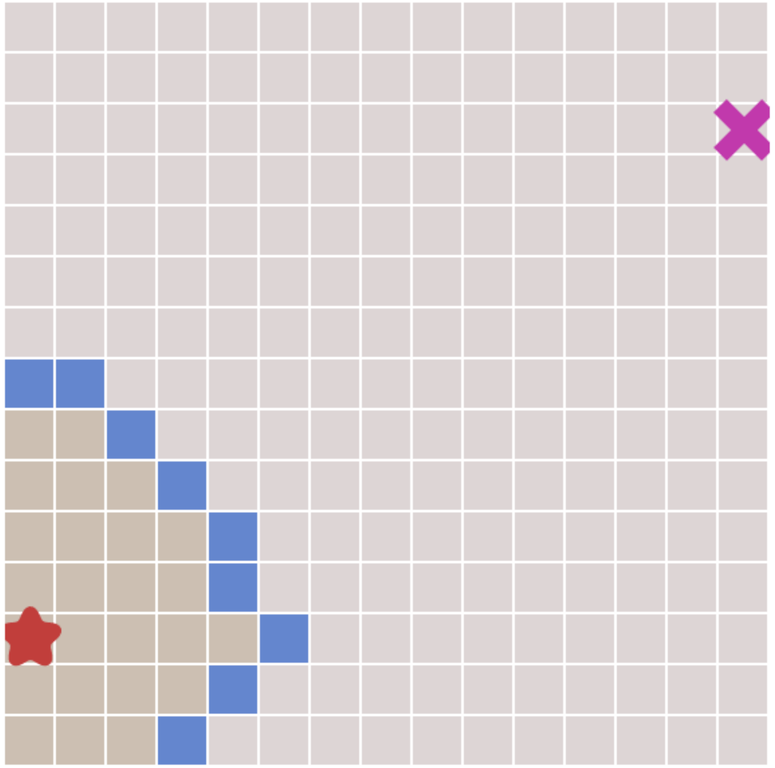
While queue has items in it, pop the front queue item, add its connected nodes to the queue (using their distance from the goal as queue priority) if they haven't been visited, mark them visited, set their "cameFrom" node to the node we popped from the queue.

CURENT NODE: C

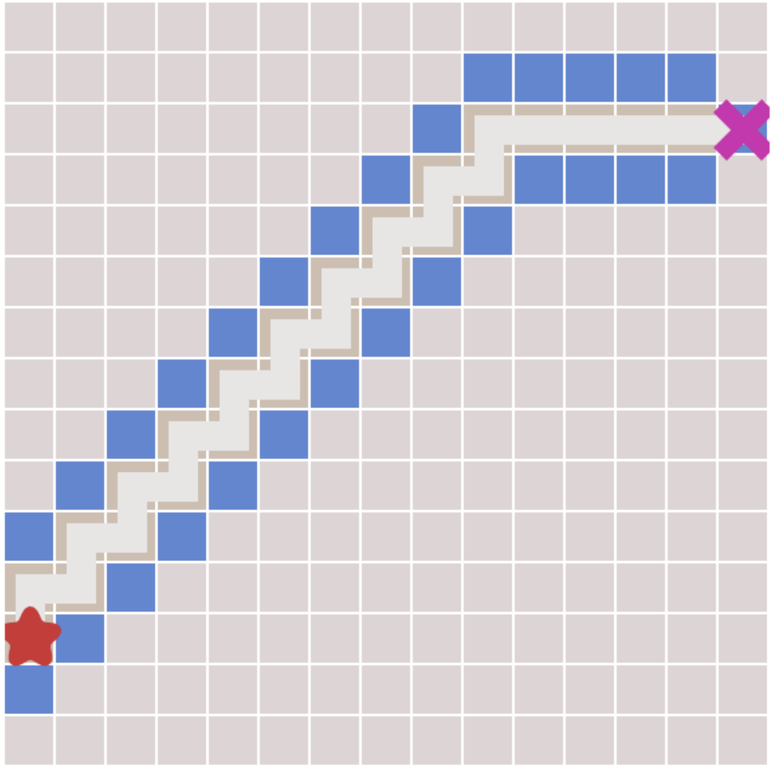
PRIORITY_QUEUE: D H B K E I N


Y
3
2
1
0
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
0
1
2
3
X

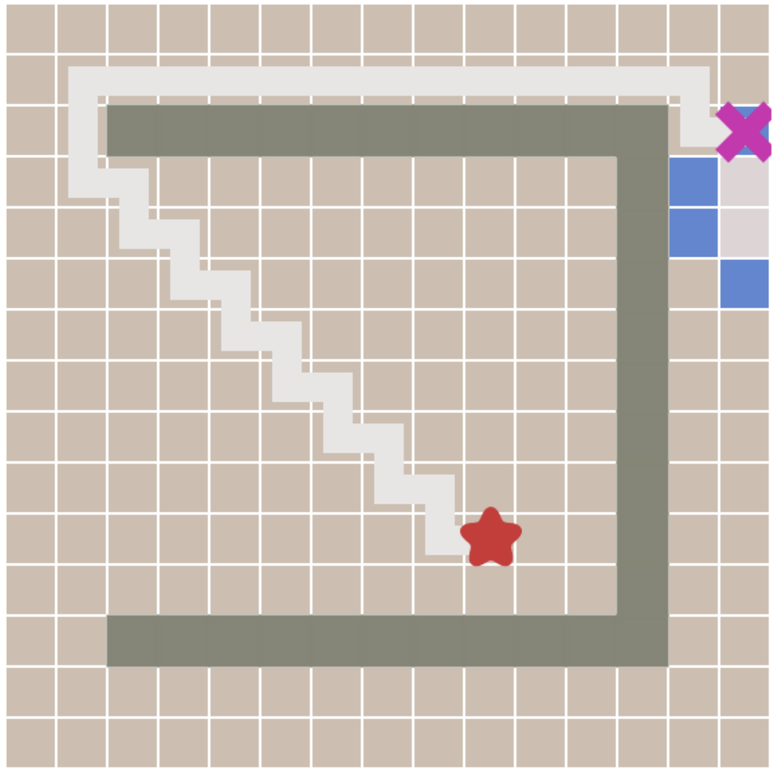
Breadth First Search



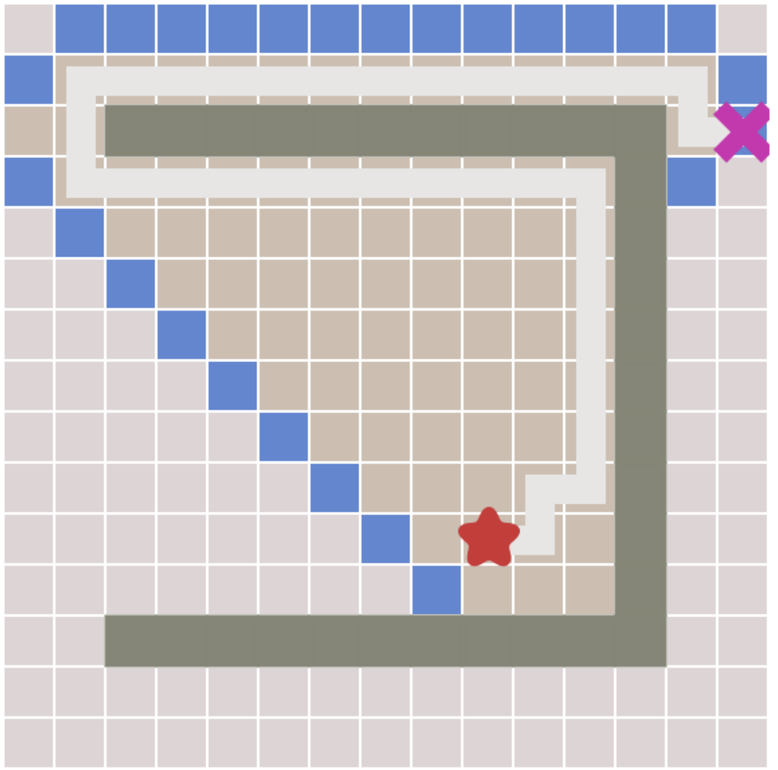
Greedy Best-First Search



Breadth First Search



Greedy Best-First Search



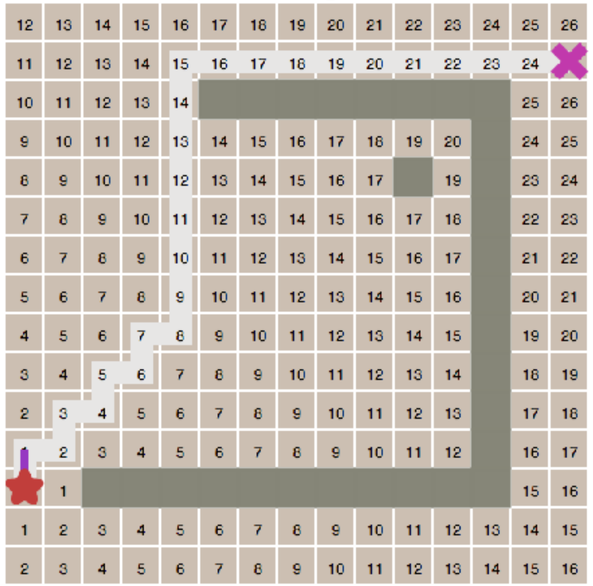
A* algorithm

Dijkstra's algorithm + **Greedy Best First** search

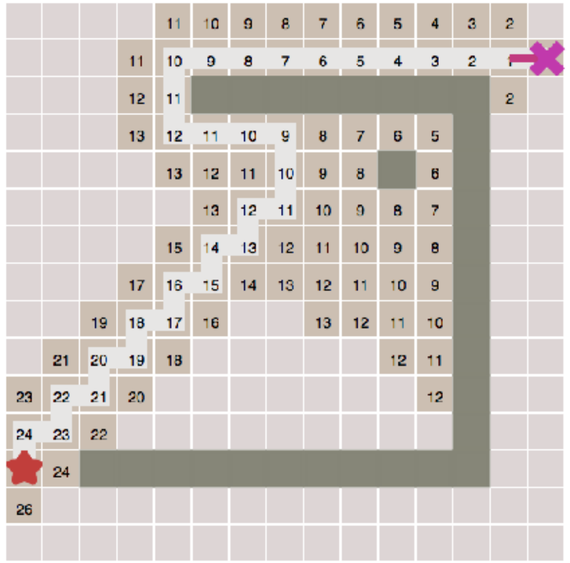
```
class Node {
    public:
        Node() {}
        bool visited;
        float cost;
        float distanceFromGoal;
        Node *cameFrom;
        std::vector<Node*> connected;
        std::vector<float> connectedWeights;
};
```

While queue has items in it, pop the front queue item, add its connected nodes
to the queue (**USING THEIR COST+DISTANCE FROM GOAL AS PRIORITY**) if they haven't been visited OR **IF THE COST OF THE CURRENT NODE PLUS THE CONNECTION COST IS LESS THAN THE COST OF THAT NODE**, mark them visited, set their "cameFrom" node to the node we popped from the queue and SET THEIR COST TO THE CURRENT NODE COST + CONNECTION COST

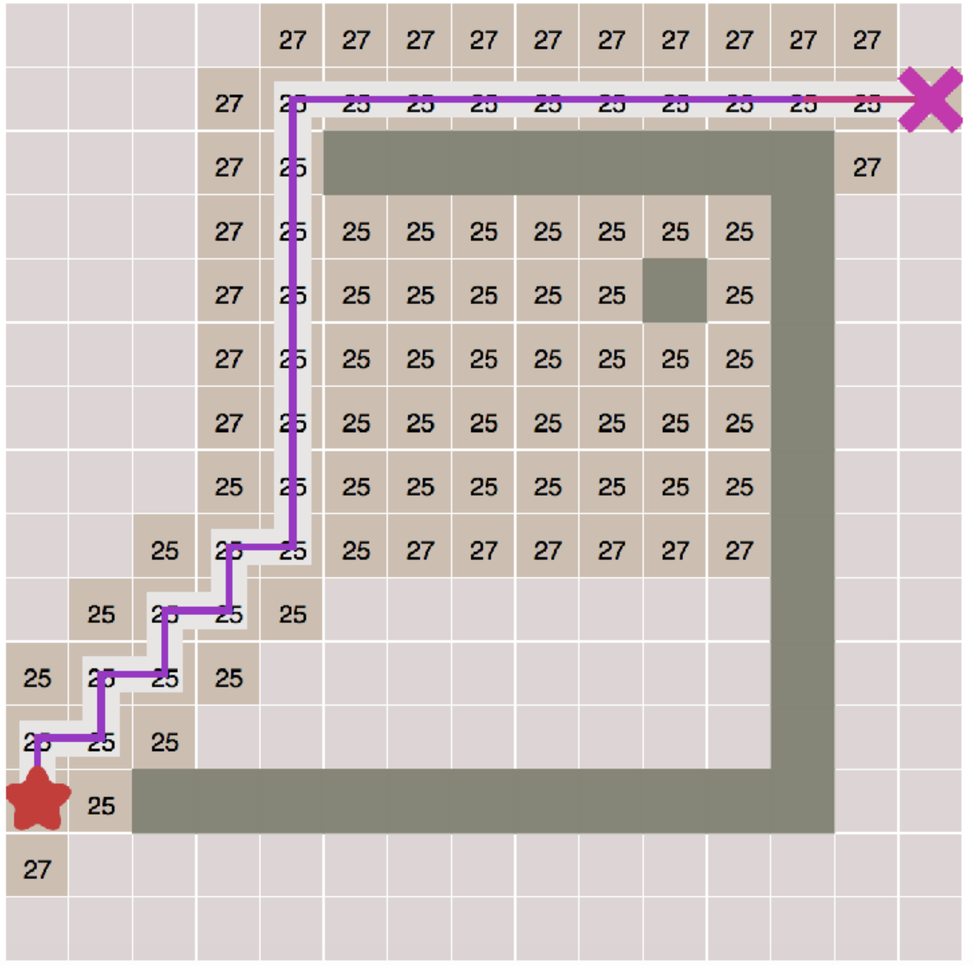
Dijkstra's Algorithm



Greedy Best-First Search

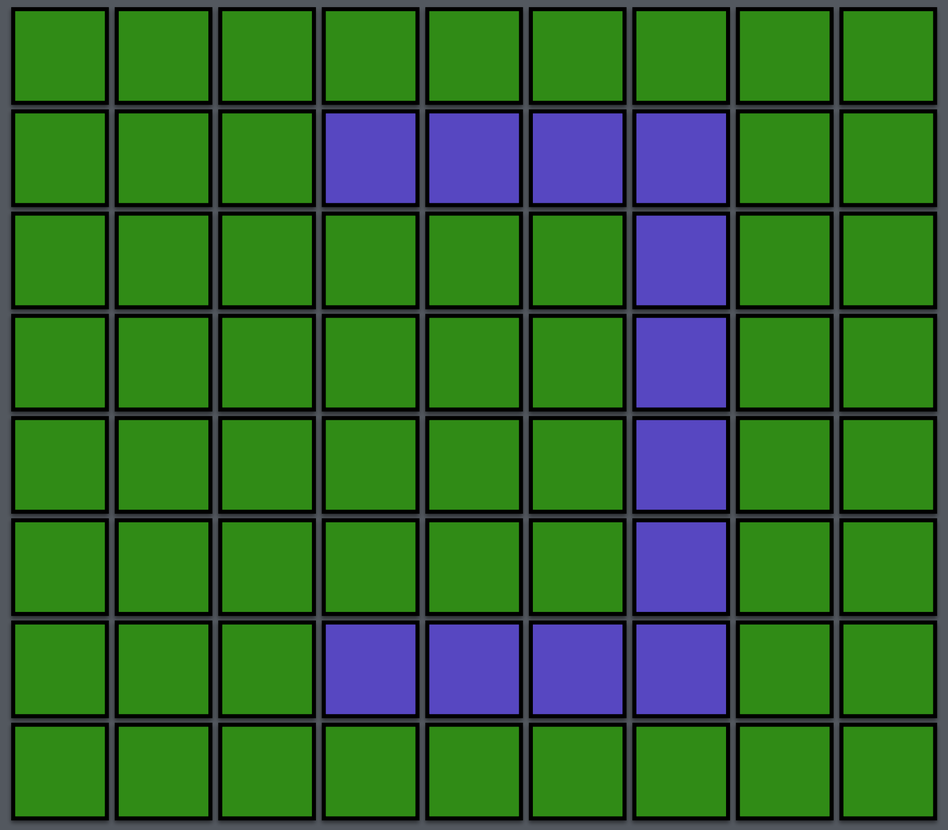


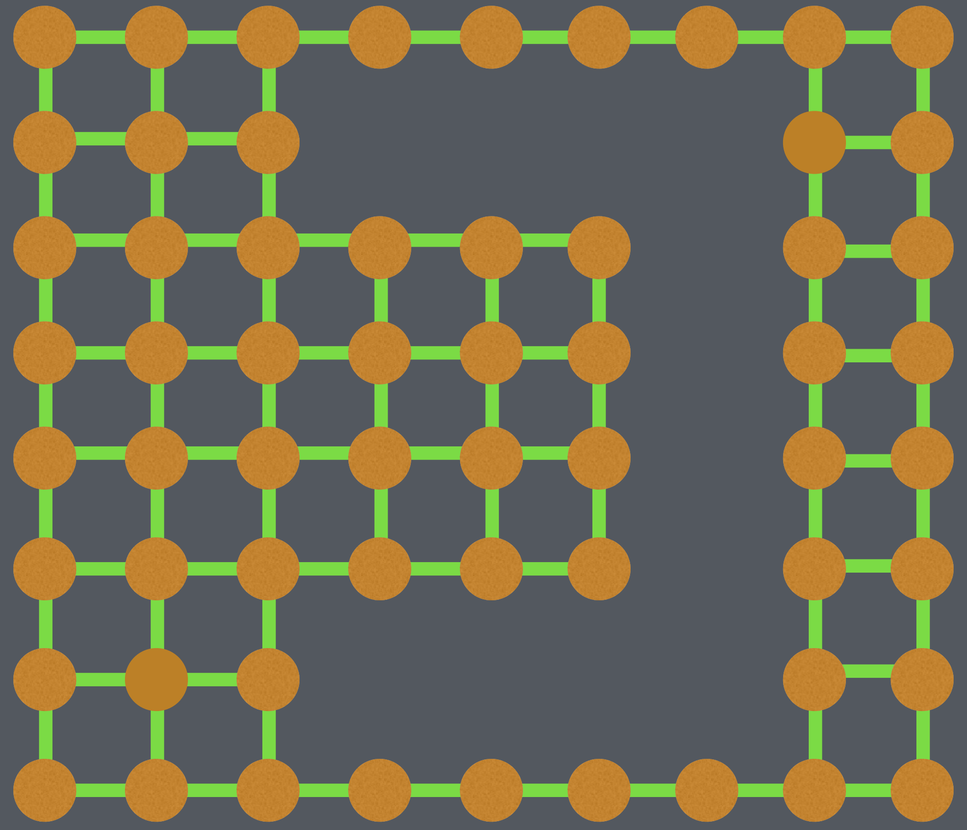
A* Search

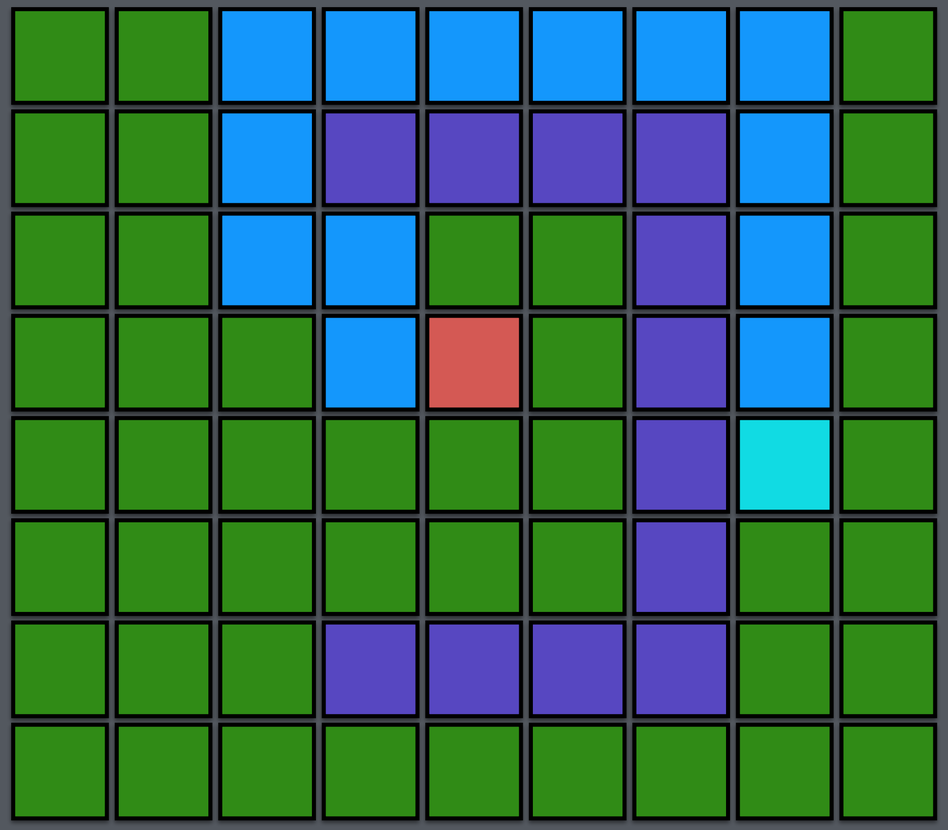


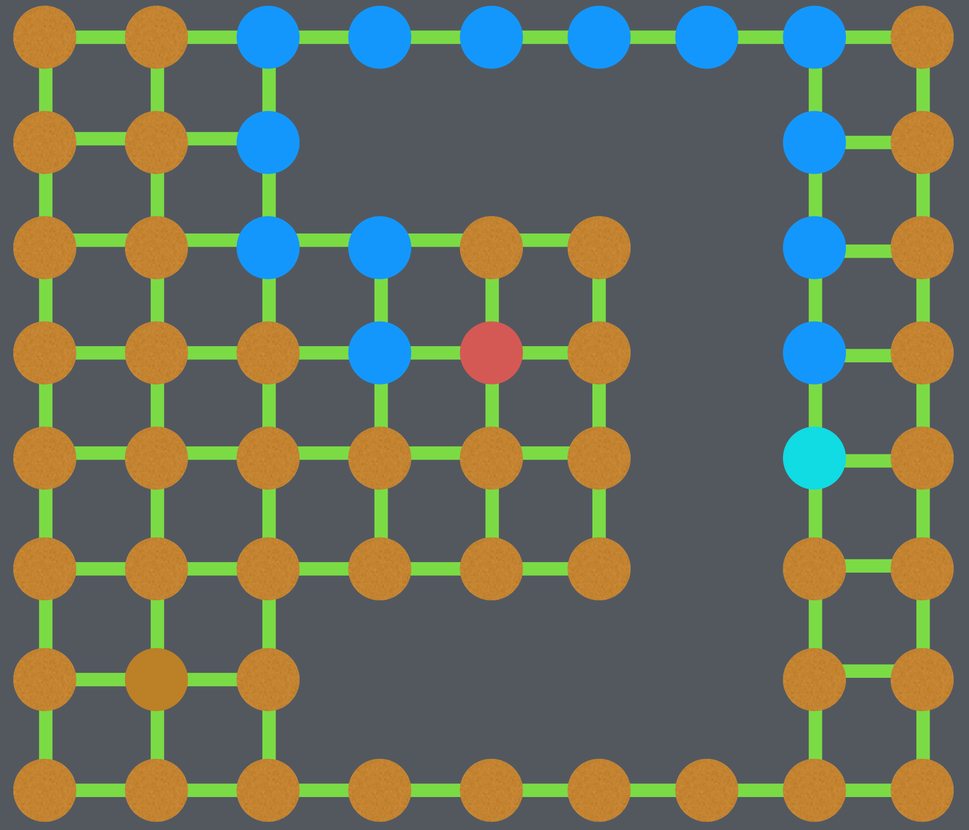
Creating our path nodes.

Grid based maps.





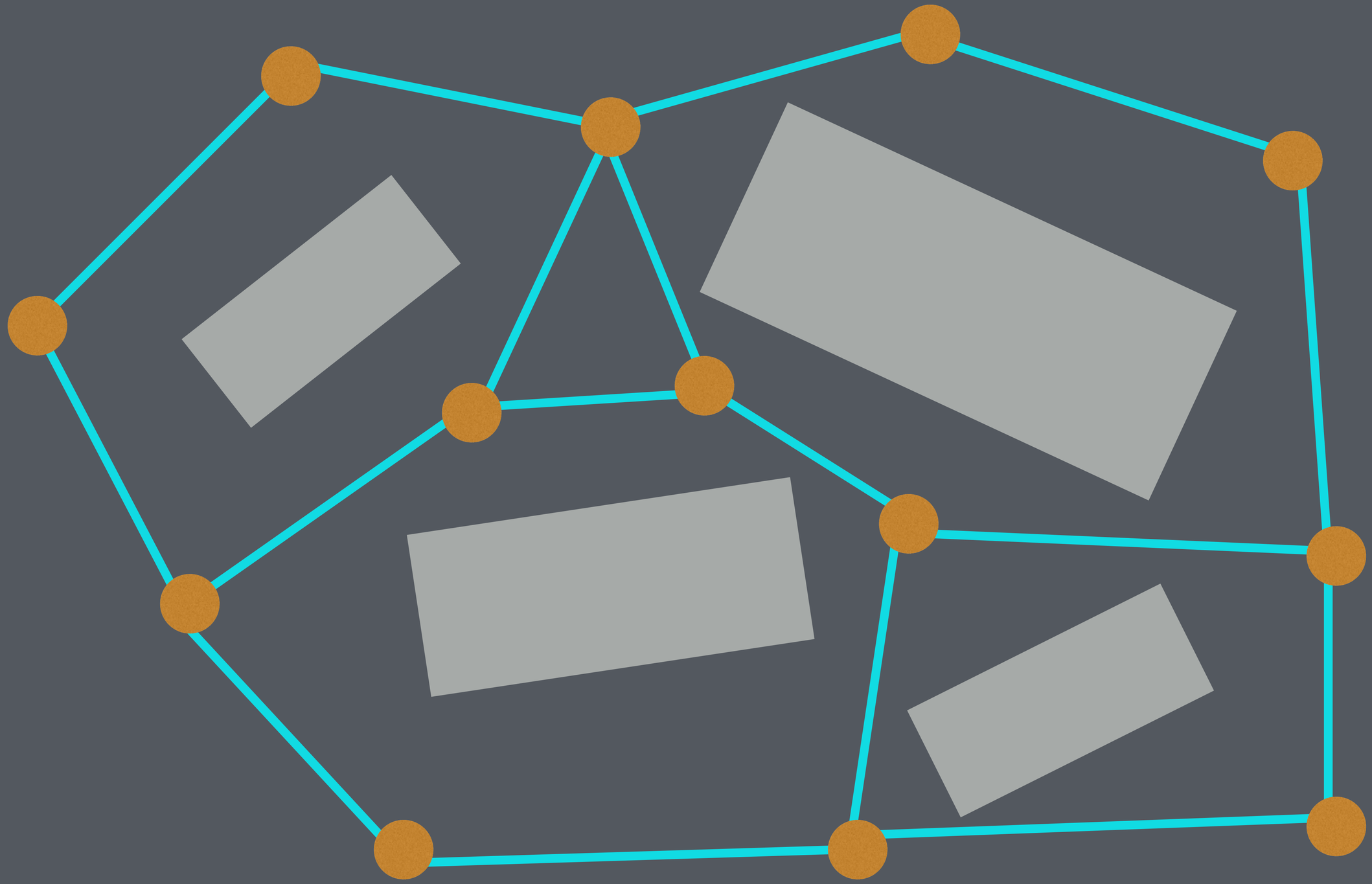




Use larger connection weights for tiles that are harder to pass.

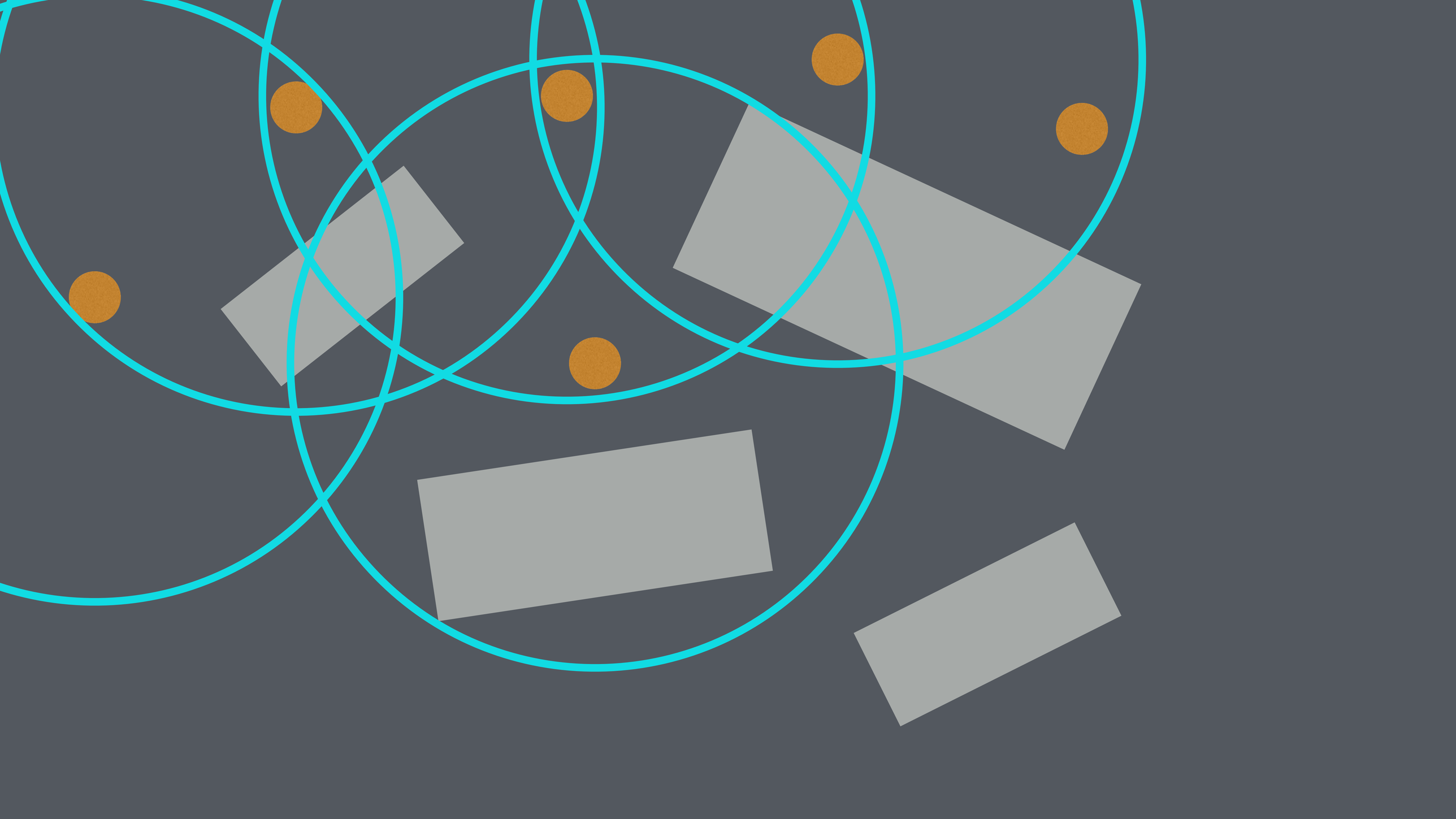
Free-form maps.

Manually place and connect AI nodes so that it is possible to reach one node from another in a straight line.

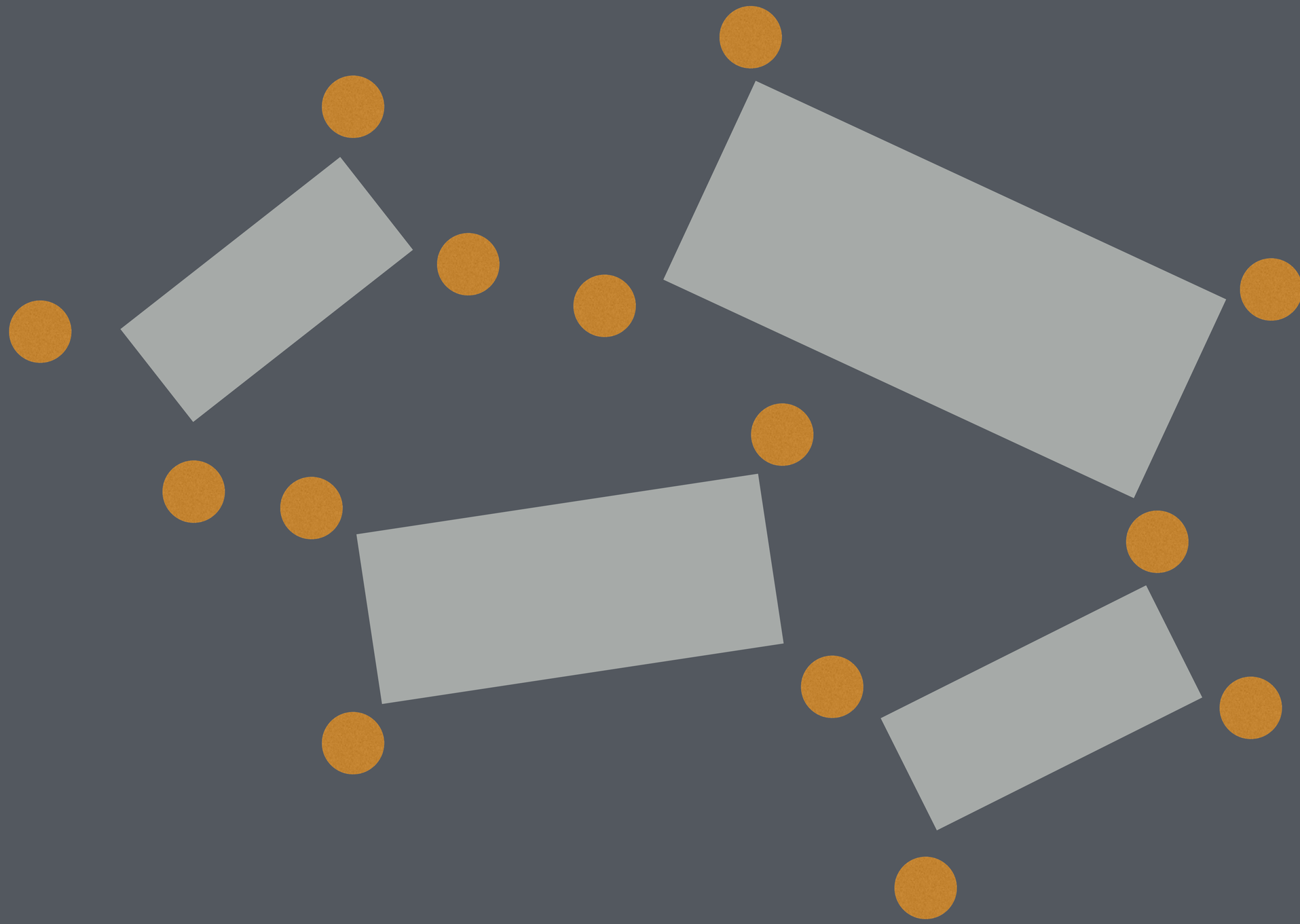


You can connect nodes automatically
based on distance, but you'll have to be careful
where you place them.

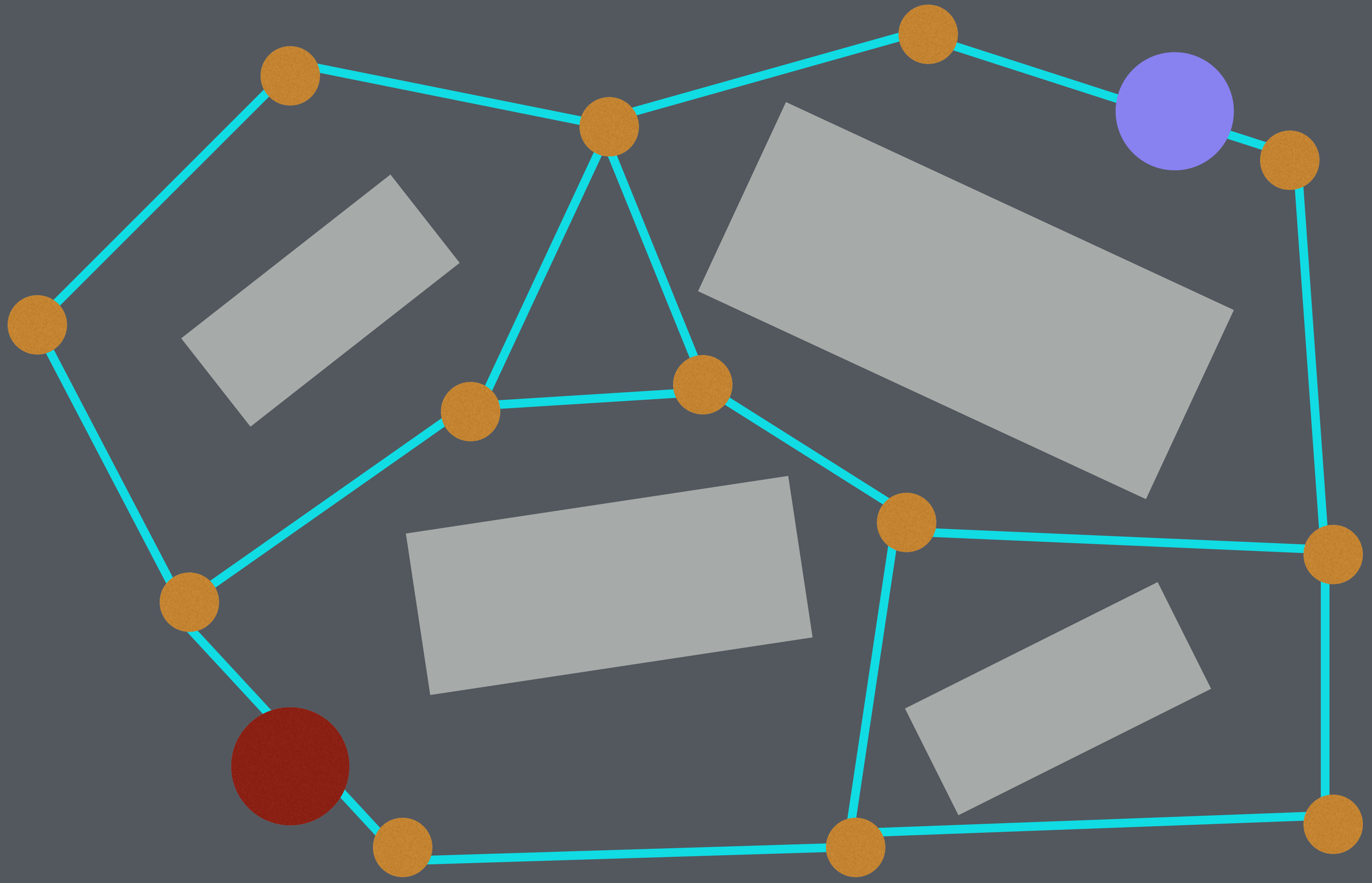
You can use a raycast to check if a node can see another
node and only connect them if they can see each other.



You can use the corners of your static polygon geometry as nodes.

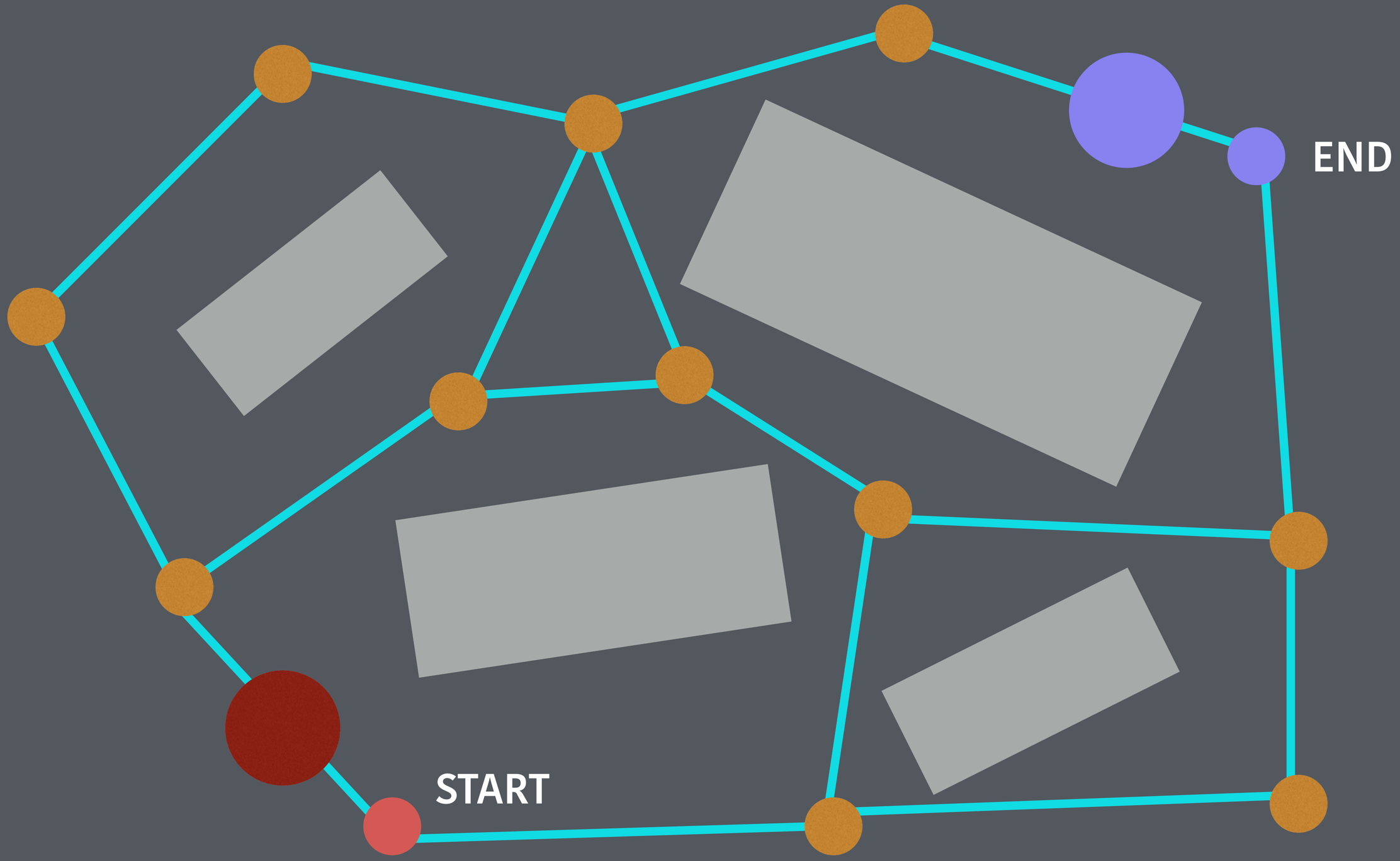


Traversing a path.

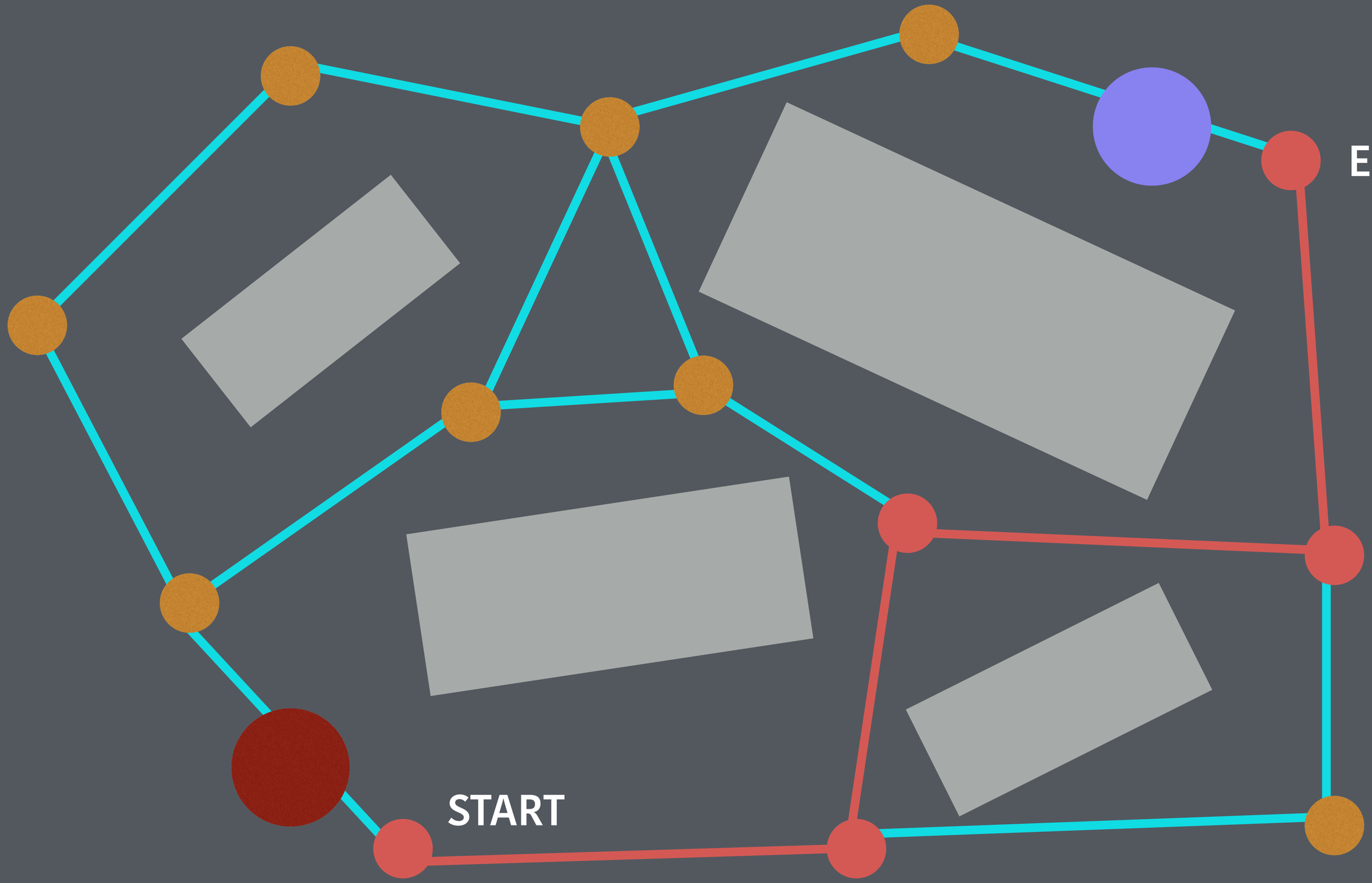


Get the closest node to the AI's position.
This will be the start node.

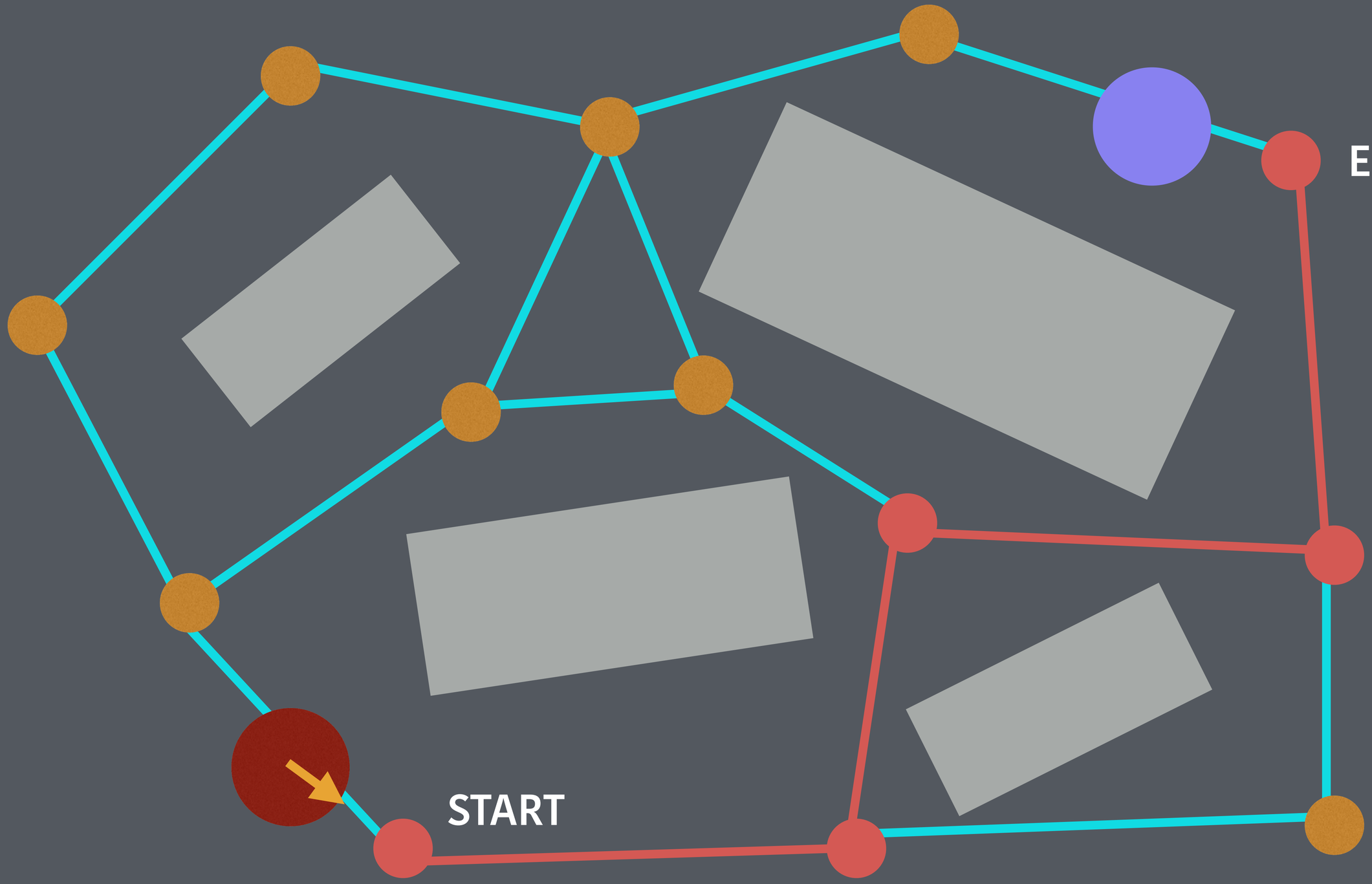
Get the closest node to the target's position.
This will be the end node.


END
START

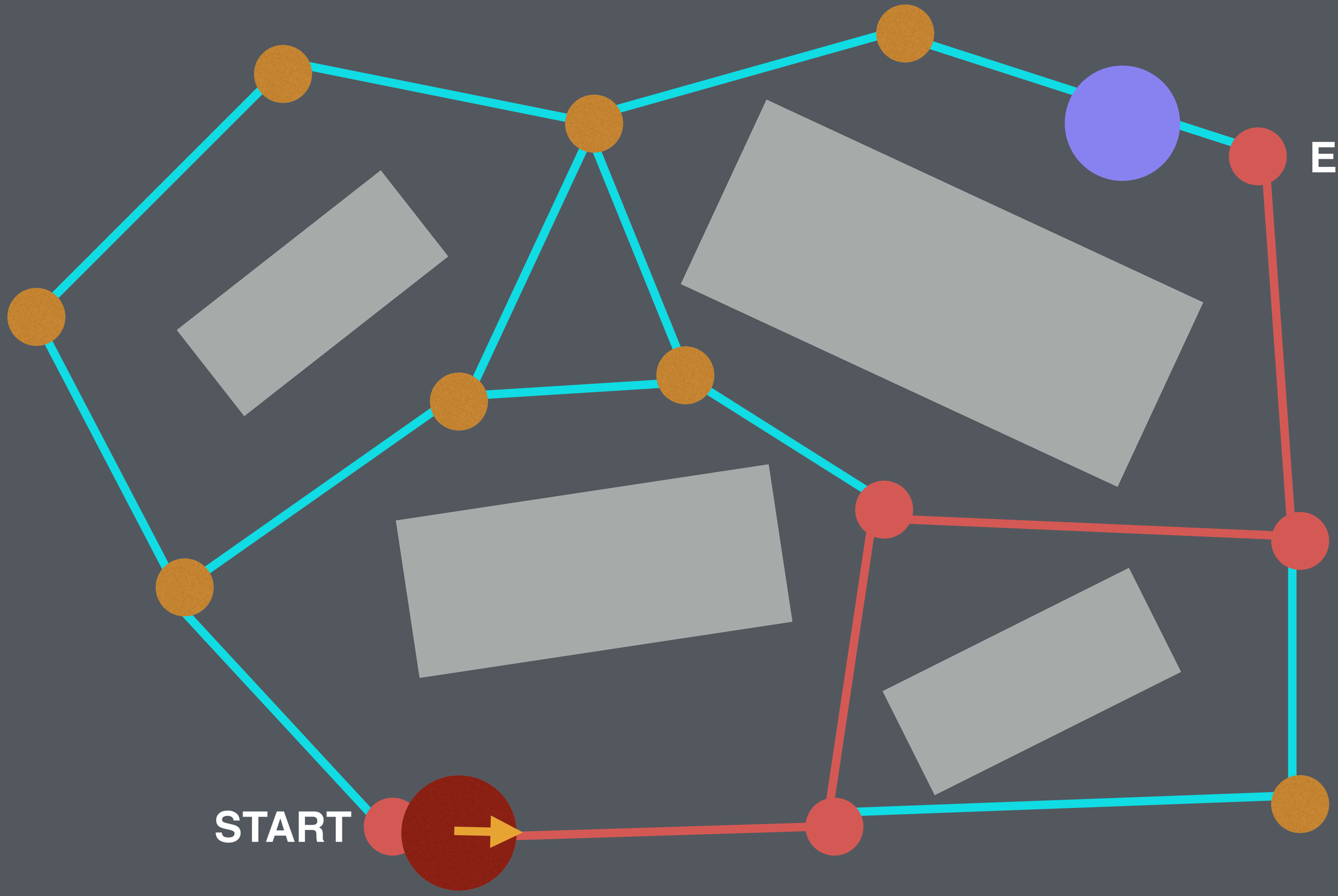
Calculate the best path using A*.


END
START

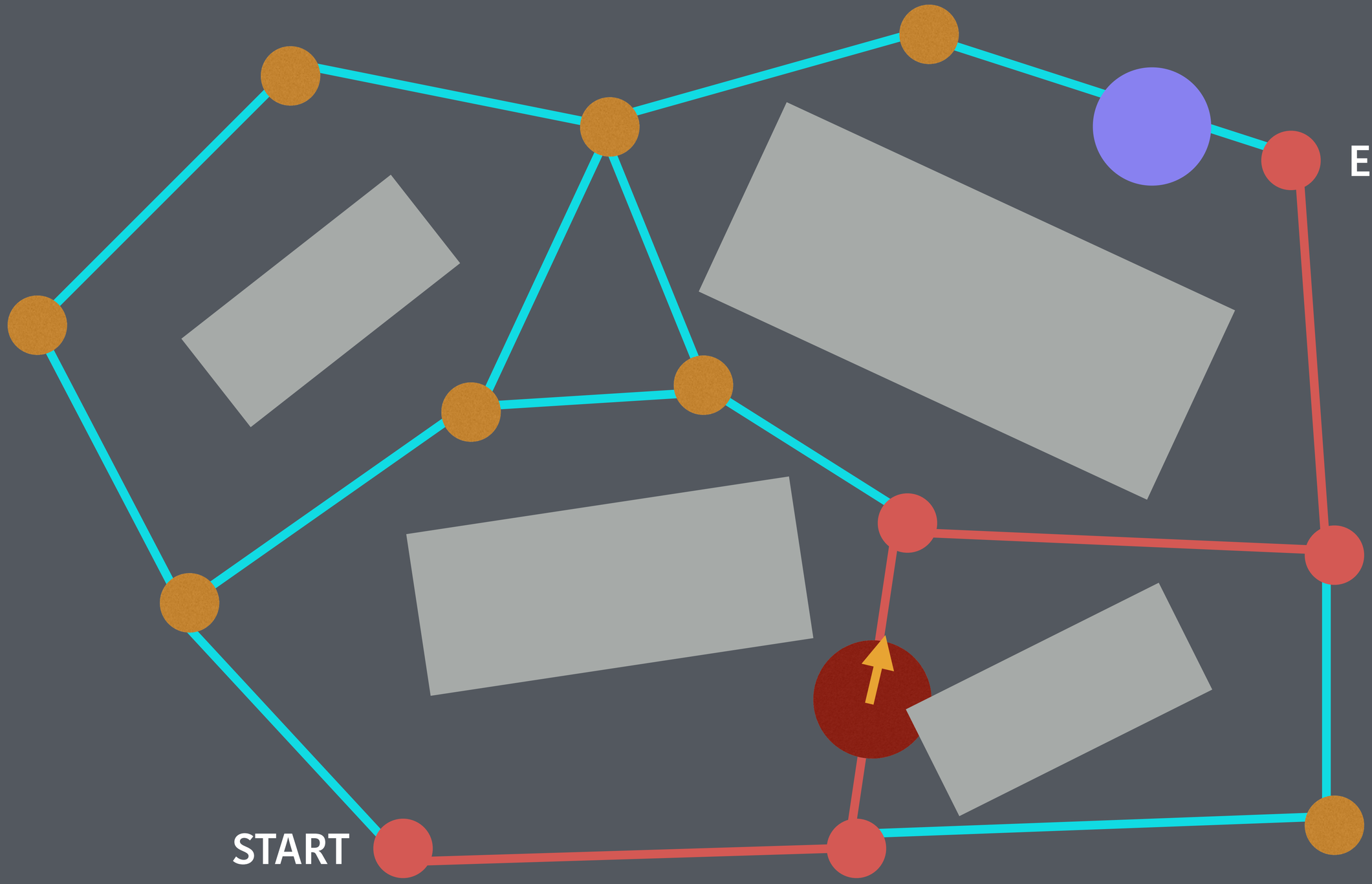
Put the path in a vector and set its first entry as the AI destination (**move towards that point on every frame**).

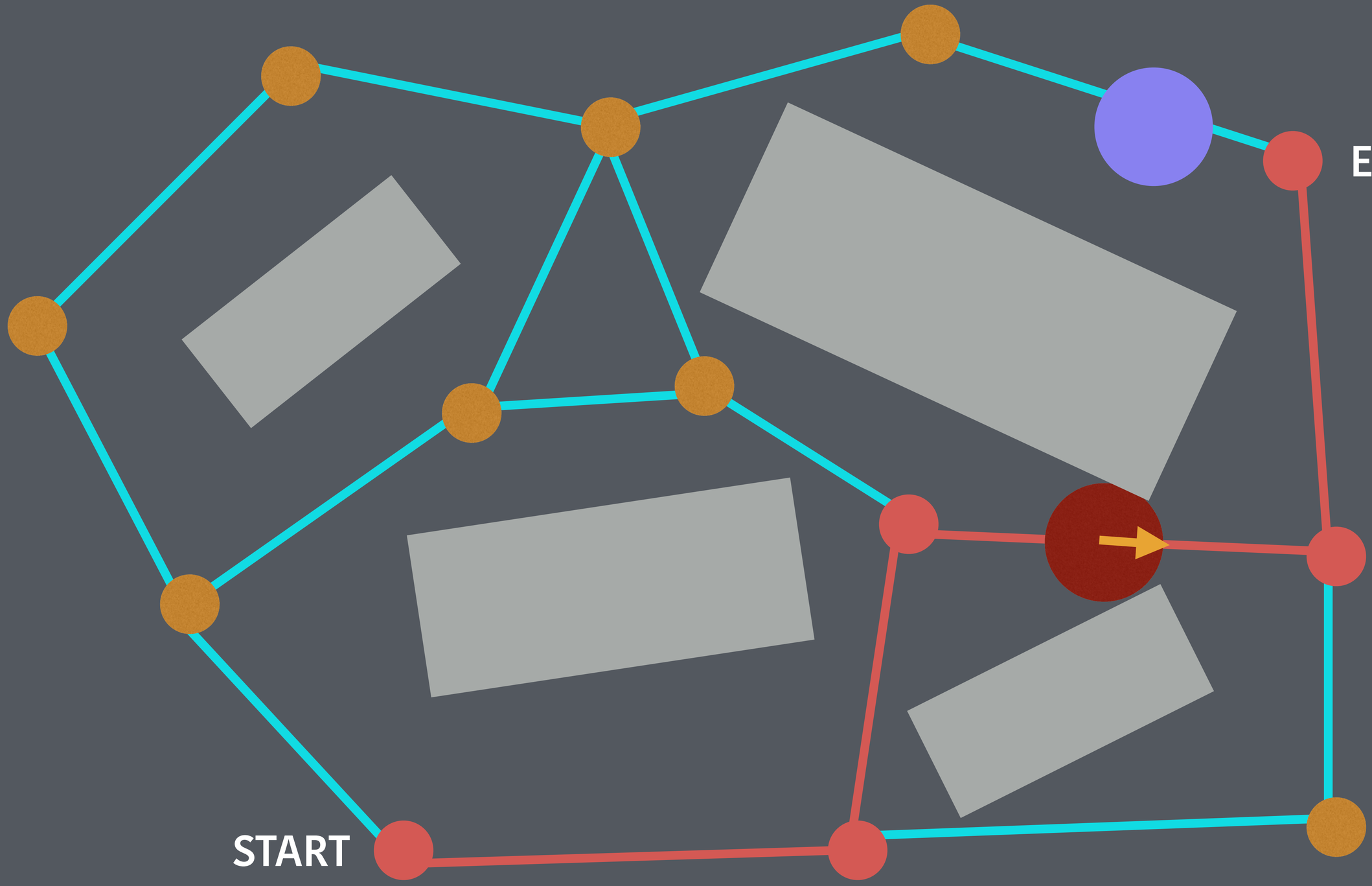

END
START

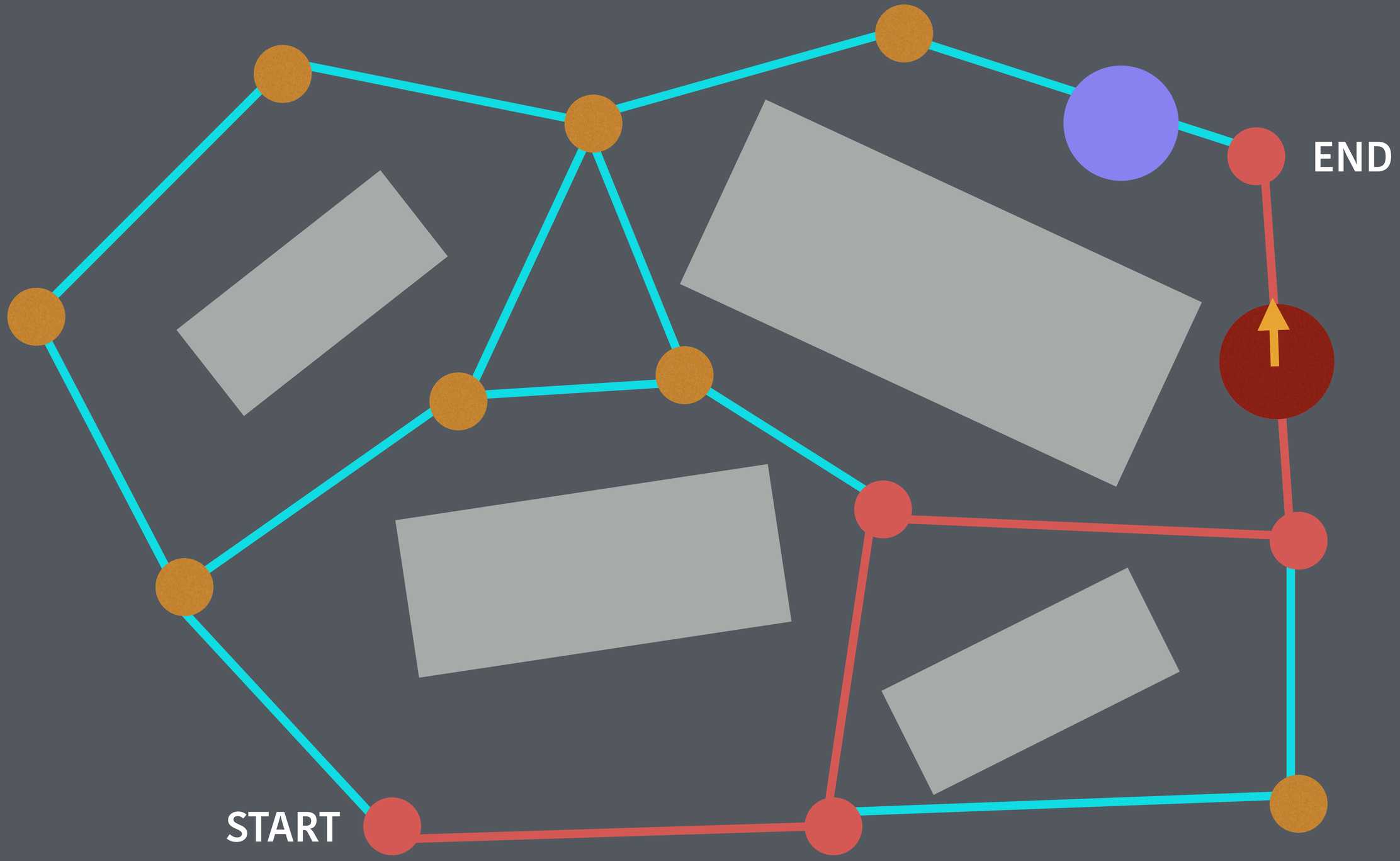
When the destination is reached (distance from AI to it is less than a certain amount), set the goal as the next node in the path.


END
START

Repeat this until you reach the **final node**.


END
START


END
START


END
START

You can recalculate the path on every frame (slow!) or on a certain interval so you can readjust your path if the **target is moving**.