Exercise 1: Adding doors

Question 1: Write the mathematical expressions for the two positions Q0 and Q1 of the extremity of a door segment placed in the middle of the edge PiPi+1.

To center a door of width w (door width) on an edge PiPi+1, we define:

Midpoint (M): M = Pi + Pi+1 / 2

Unit Direction Vector (u): u = Pi+1 – Pi / |Pi+1 - Pi|

Extremities (Q0 and Q1):

- Q0 = M – (w/2)*u
- Q1 = M + (w/2)*u

Question 2: Complete the code in the Polygon::draw(...) method to add the drawing of the doors. Use a white pen with a width of 7 for that.

```
//Draws the polygon and the interactive "doors" between server zones.
void Polygon::draw(QPainter &painter) const {
    // 1. Safety check: avoid drawing if the polygon has no vertices
    if (tabPts.empty()) return;
    // --- Part 1: Drawing the Room Area ---
    // Define the style for the room walls
    QPen pen(Qt::black);
    pen.setWidth(3);
    painter.setPen(pen);

    // Convert internal Vector2D vertices to Qt's QPolygonF for rendering
    QPolygonF poly;
    for (const auto& pt : tabPts) {
        poly << QPointF(pt.x, pt.y);
    }

    // Fill the polygon area using the color assigned to the server
    // The OddEvenFill ensures complex polygons are filled correctly.
    painter.drawPolygon(poly, Qt::OddEvenFill);
```

```
    // --- Part 2: Drawing the Doors (Exercise 1) ---
    // Doors are represented as white segments in the middle of each wall. [cite: 46, 50]
    QPen doorPen(Qt::white);
    doorPen.setWidth(7); // Use a width of 7 as specified in the exercise. [cite: 50]
    painter.setPen(doorPen);
    // Iterate through each edge of the polygon
    for (int i = 0; i < nbVertices(); ++i) {
        Vector2D P1 = tabPts[i];
        Vector2D P2 = tabPts[i+1]; // Note: tabPts[N] is a duplicate of tabPts[0]

        Vector2D edge = P2 - P1;
        double len = edge.length();

        // Only draw a door if the wall is wide enough to fit the doorWidth
        if (len > doorWidth) {
            // Calculate the exact center of the wall segment
            Vector2D mid = 0.5 * (P1 + P2);

            // Calculate the direction vector of the wall to align the door
            Vector2D unit = (1.0 / len) * edge;

            // Calculate the two extremities Q0 and Q1 of the door segment
            // The door is centered on the midpoint 'mid'.
            Vector2D Q0 = mid - (doorWidth / 2.0) * unit;
            Vector2D Q1 = mid + (doorWidth / 2.0) * unit;

            // Draw the white door segment on top of the black wall [cite: 50]
            painter.drawLine(QPointF(Q0.x, Q0.y), QPointF(Q1.x, Q1.y));
```

```
        }
    }
}
```

Exercise 2: Computation of the Graph

Question 1: In the MainWindow::createServersLinks method, create the graph connecting the servers in defining all the links between servers that are in two neighboring areas. Use defined polygons to search for the one with a common edge.

```
void MainWindow::createServersLinks() {
    // 1. Iterate through every unique pair of servers (O(N^2) complexity)
[cite: 72, 75]
    for (int i = 0; i < ui->canvas->servers.size(); ++i) {
        for (int j = i + 1; j < ui->canvas->servers.size(); ++j) {
            Server &s1 = ui->canvas->servers[i]; // First server in the pair
[cite: 17]
            Server &s2 = ui->canvas->servers[j]; // Second server in the pair
[cite: 17]

            // 2. Compare every edge of s1's polygon with every edge of s2's
polygon [cite: 73]
            for (int e1 = 0; e1 < s1.area.nbVertices(); ++e1) {
                // Get the start and end vertices of the current edge for s1
                auto edge1 = s1.area.getEdge(e1);

                for (int e2 = 0; e2 < s2.area.nbVertices(); ++e2) {
                    // Get the start and end vertices of the current edge for
s2
                    auto edge2 = s2.area.getEdge(e2);

                    // 3. Check if the edges are identical (shared boundary)
[cite: 73]
                    // Since polygons are oriented CCW, a shared edge will
have vertices
```

```
                    // in opposite order (e.g., s1: A->B, s2: B->A).
                    if ((edge1.first == edge2.second && edge1.second ==
edge2.first)) {


                        // 4. Create a new Link object representing the path
through the door [cite: 70, 72]
                        // The Link stores references to both servers and the
edge center.
                        Link *link = new Link(&s1, &s2, edge1);


                        // 5. Register the link in the canvas and both
servers for pathfinding [cite: 72, 78]
                        ui->canvas->links.append(link); // For drawing the
graph [cite: 42]
                        s1.links.append(link);          // For Dijkstra from
s1 [cite: 79]
                        s2.links.append(link);          // For Dijkstra from
s2 [cite: 79]
                    }
                }
            }
        }
    }
}
```

Question 2: What is the complexity of your algorithm?

The complexity is ON^2 * E^2, where N is the number of servers and E is the number of edges per polygon.

Question 3: The distance associated with a link is the length of the link as drawn in figure 3. It is the sum of the lengths of the segments from the server position to the middle of the shared edge plus the distance from that point to the second server position. Propose an algorithm that computes all the minimum distance from each server to all the others. This distance is store in the array server.bestDistance. For example server.bestDistance[0].second stores the distance from the current server to the server number id = 0. And server.bestDistance[0].first is the link to follow to reach server 0 using the shortest path.

The optimal algorithm for computing the minimum distance between all servers is Dijkstra's Algorithm

Question 4: Implement this algorithm in the MainWindow::fillDistanceArray() method.

```
void MainWindow::fillDistanceArray() {

    // 1. Get the total number of servers to define array sizes

    int nServers = ui->canvas->servers.size();

    if (nServers == 0) return;



    // 2. Initialize the bestDistance structure for every server

    for (auto &s : ui->canvas->servers) {

        // Resize to match the number of possible destination servers

        s.bestDistance.resize(nServers);

        for (int i = 0; i < nServers; i++) {

            // Initialize with a null link and a very large distance (infinity)

            s.bestDistance[i] = {nullptr, 1e9};

        }

        // The distance from a server to itself is always 0

        s.bestDistance[s.id] = {nullptr, 0};

    }

    // 3. Run Dijkstra's algorithm starting from each server

    // Each iteration treats 'startNode' as the source of the shortest path tree.

    for (int startNode = 0; startNode < nServers; ++startNode) {
```

```
        // Temporary array to store minimum distances found so far for this specific
startNode
        QVector<qreal> minDist(nServers, 1e9);
        minDist[startNode] = 0;

        // Track which servers have been fully processed
        QSet<int> visited;

        // Process all servers in the graph
        for (int count = 0; count < nServers; ++count) {
            // Find the unvisited server with the smallest known distance
            int u = -1;
            for (int i = 0; i < nServers; ++i) {
                if (!visited.contains(i) && (u == -1 || minDist[i] < minDist[u]))
                    u = i;
            }

            // If no reachable server is found, stop processing this startNode
            if (u == -1 || minDist[u] >= 1e9) break;
            visited.insert(u);

            // Explore all neighbors of the current server 'u' [cite: 72]
            for (Link *link : ui->canvas->servers[u].links) {
                // Determine the neighbor server on the other side of the link [cite: 70]
                Server *neighbor = (link->getNode1()->id == u) ? link->getNode2() : link-
>getNode1();
                int v = neighbor->id;

                // Get the physical length of the link (distance through the door)
```

```
                qreal weight = link->getDistance();


                // 4. Relaxation Step: Check if a shorter path to 'v' exists via 'u'
                if (minDist[u] + weight < minDist[v]) {
                    minDist[v] = minDist[u] + weight;


                    // 5. Path Tracking (Exercise 2, Question 3)
                    // We need to store the FIRST link to follow from 'startNode' to reach
'v'.
                    if (u == startNode) {
                        // If 'v' is a direct neighbor, the first step is the link itself
                        ui->canvas->servers[startNode].bestDistance[v] = {link,
minDist[v]};
                    } else {
                        // If 'v' is reached through 'u', the first step to 'v' is the same
                        // as the first step we took to get to 'u'.
                        Link* firstStep = ui->canvas-
>servers[startNode].bestDistance[u].first;
                        ui->canvas->servers[startNode].bestDistance[v] = {firstStep,
minDist[v]};
                    }
                }
            }
        }
    }
}

void MainWindow::printDistanceTable() {
    int nServers = ui->canvas->servers.size();
    if (nServers == 0) return;
```

```
qDebug() << "\n--- Computed Minimum Distances Table (Exercise 2, Q5) ---";

// 1. Print Header Row with destination names
QString header = "Source \\ Dest |";
for (const auto &s : ui->canvas->servers) {
    header += QString(" %1 |").arg(s.name.left(8), -8);
}
qDebug() << header;
qDebug() << QString(header.length(), '-');
```

Question 5: Print a table that gives all the computed distances.

```
// 2. Print each row (Source Server)
for (const auto &src : ui->canvas->servers) {
    QString row = QString("%1 |").arg(src.name.left(12), -12);

    for (int j = 0; j < nServers; ++j) {
        // Retrieve the distance computed in fillDistanceArray [cite: 78]
        double d = src.bestDistance[j].second;

        if (d >= 1e9) {
            row += "  INF   |"; // unreachable
        } else {
            row += QString(" %1 |").arg(d, 6, 'f', 1); // 1 decimal place
        }
    }
    qDebssug() << row;
```

```
    }
    qDebug() << "-------------------------------------------------------\n";
}
```

Exercise 3: Animation of drones

Question 1: Complete the method Drone::move(dt) to apply the motion rules. This function is call during the animation to update the position of the drone every dt seconds.

```
void Drone::move(qreal dt) {
    // 1. Safety check: ensure drone has a target and is within a recognized area
    if (!target || !connectedTo) return;

    // Calculate vector and distance to the current intermediate destination
    Vector2D dir = destination - position;
    double d = dir.length();

    // --- State Transition Logic (Exercise 3) ---
    // Triggered when the drone is close enough to its current waypoint
    if (d < minDistance) {

        // Case A: Drone reached a Server position
        if (destination == Vector2D(connectedTo->position.x(), connectedTo->position.y())) {
            // If this isn't the final destination, find the next door to cross
            if (connectedTo != target) {
                // bestDistance[target->id].first stores the next Link to take
                Link *nextLink = connectedTo->bestDistance[target->id].first;
                if (nextLink) {
                    // Head toward the middle of the shared edge (the door)
                    destination = nextLink->getEdgeCenter();
```

```
                }
            }
        }
        // Case B: Drone reached a Door (Edge Center)
        else {
            Link *currentLink = connectedTo->bestDistance[target->id].first;
            if (currentLink) {
                // Determine the server on the opposite side of the door
                // If node1 is where we just came from, move to node2, and
vice versa.
                connectedTo = (currentLink->getNode1() == connectedTo) ?
                                  currentLink->getNode2() : currentLink-
>getNode1();

                // Set the new destination to the center of the next room's
server
                destination = Vector2D(connectedTo->position.x(),
connectedTo->position.y());
            }
        }

        // Recalculate direction and distance for the new waypoint
        dir = destination - position;
        d = dir.length();
    }

    // --- Physics and Movement Calculations ---
    // Implements acceleration, maximum speed capping, and arrival smoothing



    if (d < slowDownDistance) {
        // Linear deceleration to avoid overshooting the waypoint
        speed = (d * speedLocal / slowDownDistance) * dir;
```

```
    } else {
        // Standard acceleration logic
        speed += (accelation * dt / d) * dir;

        // Cap speed at speedMax
        if (speed.length() > speedMax) {
            speed.normalize();
            speed *= speedMax;
        }
    }

    // Update the position based on calculated velocity
    position += (dt * speed);

    // --- Orientation (Azimut) Update ---
    // Rotates the drone icon to face the direction of travel
    if (speed.length() > 0.01) {
        // Normalize speed to get the direction vector
        Vector2D Vn = (1.0 / speed.length()) * speed;

        // Calculate angle in degrees using trigonometry
        // Handles vertical movement and quadrant adjustments for the UI.
        azimut = (Vn.y == 0) ? (Vn.x > 0 ? -90 : 90) :
                     (Vn.y > 0 ? 180.0 - 180.0 * atan(Vn.x / Vn.y) / M_PI :
                          -180.0 * atan(Vn.x / Vn.y) / M_PI);
    }
}
```