Drones and Rooms

Voronoi, Graph Routing, Drone Animation

Advanced Algorithms Mini-Project Report

Ahmed Almuharaq
Boluwatife Abiona
Master IoT 1 – UFR STGI

January 17, 2026

Supervisor: Prof. Benoît Piranda

Project source: DroneAndRooms (provided initial code + exercises)

Contents

1 Project Overview

1.1 Context

This mini-project simulates drones navigating inside a 2D environment partitioned into Voronoi regions ("rooms"). Each room contains a server represented by a point (name + position + color). A drone starts from an initial position and must reach a target server. The application is implemented in C++/Qt and visualized with `QPainter`, while animation runs through periodic updates (timer ticks).

1.2 Objectives

The project consists of three parts:

1. **Exercise 1 (Geometry):** Draw doors of fixed width at the middle of each polygon edge (room boundary).
2. **Exercise 2 (Graph + Routing):** Create a graph between servers whose Voronoi areas share an edge, then compute shortest paths and fill routing tables `bestDistance`.
3. **Exercise 3 (Animation):** Implement drone motion rules (server → door → server ...) using computed routing tables.

2 System Architecture

2.1 Main Classes

Server Stores `id`, `name`, `position`, `area` (Voronoi polygon), neighbor `links`, and the routing table `bestDistance`.

Polygon Stores the vertices of the Voronoi cell and draws boundaries and doors.

Link Connects two neighbor servers and stores the shared edge (and its center). The link distance corresponds to the path: server → edge center → other server.

Drone Stores `position`, `speed`, `connectedTo` (current server/area), `target`, and movement state through `destination`.

MainWindow/Canvas Loads JSON, computes Voronoi cells, builds links, computes routing tables, and updates animation.

2.2 Execution Pipeline

1. Load servers/drones from a JSON file.
2. Compute Voronoi polygons and assign each polygon to its server.
3. Build links between neighboring servers: `MainWindow::createServersLinks()`.
4. Compute all-pairs shortest paths and fill `bestDistance`: `MainWindow::fillDistanceArray()`.
5. Each timer tick: update drones with `Drone::move(dt)` and repaint.

3 Exercise 1: Adding Doors (Geometry)

Question 1: Mathematical expressions for Q_0 and Q_1

Consider the polygon edge between vertices P_i and P_{i+1} (points A and B):

$$A = P_i, \quad B = P_{i+1}$$

The midpoint is:

$$M = \frac{A + B}{2}$$

Define the unit direction vector along the edge:

$$\mathbf{u} = \frac{B - A}{\|B - A\|}$$

If the door width is `doorWidth`, then half-width is $\frac{\texttt{doorWidth}}{2}$ and:

$$Q_0 = M - \mathbf{u} \cdot \frac{\texttt{doorWidth}}{2}, \qquad Q_1 = M + \mathbf{u} \cdot \frac{\texttt{doorWidth}}{2}$$

These points define the door segment, centered and aligned with the edge.

Question 2: Implement door drawing in `Polygon::draw(...)`

We added door rendering inside `Polygon::draw(QPainter&)`. Doors are drawn with a white pen of width 7, and for every edge we compute Q_0, Q_1 and draw a segment:

```
// --- Draw Doors (Exercise 1) ---
QPen doorPen(Qt::white);
doorPen.setWidth(7);
painter.setPen(doorPen);

const int n = nbVertices();
for (int i = 0; i < n; ++i) {
    Vector2D A = tabPts[i];
    Vector2D B = tabPts[i + 1]; // polygon is closed in data structure

    Vector2D AB = B - A;
    const double len = AB.length();
    if (len < 1e-9) continue;

    Vector2D u = (1.0 / len) * AB;           // Unit direction
    Vector2D M = 0.5 * (A + B);              // Midpoint
    Vector2D half = (doorWidth * 0.5) * u;   // Half door vector

    Vector2D Q0 = M - half;
    Vector2D Q1 = M + half;

    painter.drawLine(QPointF(Q0.x, Q0.y), QPointF(Q1.x, Q1.y));
}
```

4 Exercise 2: Graph Computation and Shortest Paths

Question 1: Create links between neighboring areas (`createServersLinks`)

Two servers are neighbors if their Voronoi polygons share a common edge (the same segment, possibly reversed). We compare edges of every pair of polygons; when a common edge is found, we create a `Link` storing that shared edge and attach it to both servers.

```
void MainWindow::createServersLinks() {
    // Cleanup previous links
    for (auto *l : ui->canvas->links) delete l;
    ui->canvas->links.clear();
    for (auto &s : ui->canvas->servers) s.links.clear();

    auto samePoint = [](const Vector2D &a, const Vector2D &b) -> bool {
        const double eps2 = 1e-6;
        return a.distance2(b) <= eps2;
    };

    const int n = ui->canvas->servers.size();
    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {
            Polygon &polyA = ui->canvas->servers[i].area;
            Polygon &polyB = ui->canvas->servers[j].area;

            bool found = false;
            QPair<Vector2D, Vector2D> commonEdge;

            for (int ea = 0; ea < polyA.nbVertices() && !found; ++ea) {
                auto eA = polyA.getEdge(ea);
                for (int eb = 0; eb < polyB.nbVertices() && !found; ++eb) {
                    auto eB = polyB.getEdge(eb);

                    bool same = samePoint(eA.first, eB.first) &&
                                samePoint(eA.second, eB.second);
                    bool opp  = samePoint(eA.first, eB.second) &&
                                samePoint(eA.second, eB.first);

                    if (same || opp) {
                        commonEdge = eA;
                        found = true;
                    }
                }
            }

            if (found) {
                Link *link = new Link(&ui->canvas->servers[i],
                                      &ui->canvas->servers[j],
                                      commonEdge);
                ui->canvas->links.append(link);
                ui->canvas->servers[i].links.append(link);
                ui->canvas->servers[j].links.append(link);
            }
        }
    }
}
```

Question 2: Complexity of the graph construction algorithm

Let n be the number of servers and v the average number of polygon vertices.

- We test every pair of servers: $O(n^2)$.
- For each pair, we compare all edges in polygon A against all edges in polygon B: $O(v^2)$.

Therefore, the complexity is:

$$O(n^2 \cdot v^2)$$

Question 3: Propose an algorithm for all minimum distances + routing table

We need all-pairs shortest paths and the first link to follow from any server i to any target j:

- **Option A:** Run Dijkstra from each server (n times). Complexity about $O(n \cdot (m \log n))$.
- **Option B:** Floyd–Warshall, simpler for small n, complexity $O(n^3)$.

We chose **Floyd–Warshall** because the number of servers is small and we want all-pairs distances directly. We also maintain a `next` matrix to reconstruct the first hop and store it into `server.bestDistance[j].first` as the first `Link` to take.

Question 4: Implement the algorithm in `fillDistanceArray`

We implemented Floyd–Warshall and then populated `bestDistance`. For each pair (i, j):

- `bestDistance[j].second` is the shortest distance value.
- `bestDistance[j].first` is the first link to follow (first hop).

```
void MainWindow::fillDistanceArray() {
    const int nServers = ui->canvas->servers.size();
    const float INF = 1e30f;

    QVector<QVector<float>> dist(nServers, QVector<float>(nServers, INF));
    QVector<QVector<int>> next(nServers, QVector<int>(nServers, -1));

    for (int i = 0; i < nServers; ++i) {
        dist[i][i] = 0.0f;
        next[i][i] = i;
    }

    // Direct links
    for (auto *l : ui->canvas->links) {
        int a = l->getNode1()->id;
        int b = l->getNode2()->id;
        float w = static_cast<float>(l->getDistance());
        if (w < dist[a][b]) {
            dist[a][b] = w; dist[b][a] = w;
```

```
            next[a][b] = b; next[b][a] = a;
        }
    }

    // F l o y d Warshall
    for (int k = 0; k < nServers; ++k)
        for (int i = 0; i < nServers; ++i)
            for (int j = 0; j < nServers; ++j)
                if (dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                    next[i][j] = next[i][k];
                }

    // Fill bestDistance: (firstLink, distance)
    for (int i = 0; i < nServers; ++i) {
        ui->canvas->servers[i].bestDistance.resize(nServers);
        for (int j = 0; j < nServers; ++j) {
            if (dist[i][j] >= INF || i == j) {
                ui->canvas->servers[i].bestDistance[j] = {nullptr, dist[i][j]};
                continue;
            }

            int hop = next[i][j]; // first hop from i toward j
            Link *firstLink = nullptr;
            for (auto *l : ui->canvas->servers[i].links) {
                int n1 = l->getNode1()->id;
                int n2 = l->getNode2()->id;
                if ((n1 == i && n2 == hop) || (n2 == i && n1 == hop)) {
                    firstLink = l;
                    break;
                }
            }
            ui->canvas->servers[i].bestDistance[j] = {firstLink, dist[i][j]};
        }
    }

    // Optional: print distance table (Question 5)
    // (See next section for formatted printing)
}
```

Question 5: Print a table of all computed distances

We printed a distance matrix after computing `dist`. The following snippet prints a clear table containing server names and all shortest-path distances:

```
// --- Print distance table (Exercise 2, Q5) ---
QString header = "From/To\t";
for (int j = 0; j < nServers; ++j)
    header += ui->canvas->servers[j].name + "\t";
qDebug().noquote() << header;

for (int i = 0; i < nServers; ++i) {
    QString row = ui->canvas->servers[i].name + "\t";
    for (int j = 0; j < nServers; ++j) {
        if (dist[i][j] >= INF/2) row += "INF\t";
```

```
            else row += QString::number(dist[i][j], 'f', 2) + "\t";
    }
    qDebug().noquote() << row;
}
```

5 Exercise 3: Drone Animation

Question 1: Complete `Drone::move(dt)` to follow motion rules

The drone logic follows the required rule sequence:

1. Associate drone to its current overflown area server (`connectedTo` already assigned using the provided method).
2. First destination: fly to the local server position.
3. When at a server: pick next hop link toward the target using `bestDistance`, and set destination to the link edge center (door).
4. When at door center: switch `connectedTo` to the opposite server and then fly to that new server position.

```
void Drone::move(qreal dt) {
    // Initial movement: if destination not set, go to local server
    if (connectedTo != nullptr && destination.distance2(Vector2D()) == 0.0) {
        destination = Vector2D(connectedTo->position.x(), connectedTo->position.y());
    }

    auto distTo = [&](const Vector2D &p) -> double { return (p - position).length(); };
    auto serverPos = [&](Server *s) -> Vector2D {
        return Vector2D(s->position.x(), s->position.y());
    };

    if (connectedTo != nullptr) {
        double dServer = distTo(serverPos(connectedTo));

        // If we are at the server: choose next door toward target
        if (dServer < minDistance) {
            if (target != nullptr && target->id < connectedTo->bestDistance.size()) {
                Link *nextLink = connectedTo->bestDistance[target->id].first;
                if (nextLink) destination = nextLink->getEdgeCenter();
            }
        } else {
            // If we reached the door: switch to neighbor server and go to it
            if (distTo(destination) < minDistance) {
                for (auto *l : connectedTo->links) {
                    if ((l->getEdgeCenter() - destination).length() < minDistance) {
                        Server *a = l->getNode1();
```

```
                        Server *b = l->getNode2();
                        connectedTo = (connectedTo == a) ? b : a;
                        destination = serverPos(connectedTo);
                        break;
                    }
                }
            }
        }
    }

    // Physics integration
    Vector2D dir = destination - position;
    double d = dir.length();

    if (d < slowDownDistance) {
        speed = (d * speedLocal / slowDownDistance) * dir;
    } else {
        speed += (accelation * dt / d) * dir;
        if (speed.length() > speedMax) {
            speed.normalize();
            speed *= speedMax;
        }
    }

    position += (dt * speed);

    // Azimuth (rotation)
    if (speed.length() > 1e-9) {
        Vector2D Vn = (1.0 / speed.length()) * speed;
        azimut = (Vn.y > 0) ? (180.0 - 180.0 * atan(Vn.x / Vn.y) / M_PI)
                            : (-180.0 * atan(Vn.x / Vn.y) / M_PI);
    }
}
```

Question 2: Propose a new JSON configuration and test

We created a new JSON configuration file named `Ahmed.json` with:

- A larger window (`origine = -120,-80` and `size = 1600,1100`).
- Seven servers using cities in the region (Besançon, Montbéliard, Belfort, Vesoul, Dole, Lons-le-Saunier, Pontarlier) with new color palette.
- Seven drones with different start positions and targets.

This configuration was loaded through `File/Open` and the algorithm produced:

- Correct doors placement on Voronoi edges.
- Correct graph creation between neighboring cells.
- Successful drone motion: each drone reaches its local server then crosses doors following the shortest path to its target.

6 Algorithmic Complexity Summary

- **Graph construction (shared edges search):** $O(n^2 \cdot v^2)$.
- **Shortest paths (Floyd–Warshall):** $O(n^3)$.
- **Per-drone update per frame:** $O(\text{degree})$ (checking neighbor links), typically very small.

7 Conclusion

We implemented all required steps of the mini-project:

- Doors drawn correctly at the middle of each polygon edge.
- Graph built between neighboring Voronoi areas, with correct link distances.
- All-pairs shortest paths computed and stored into `bestDistance` as (firstLink, distance).
- Drones animated according to the motion rules using the computed routing table.
- A new custom JSON configuration (`Ahmed.json`) was created and successfully tested.