HiveMind - Proiect POO

Chiriac Mihai-Ciprian

9 ianuarie 2026

Cuprins

1 Introducere

Pe scurt, am folosit RNG pentru a genera random coordonatele x și y ale diverselor elemente statice ale hărții (Zid (#), Bază (B), Stație (S), Client (D)). De fapt, am plasat Baza în mijlocul hărții (*randuri*/2, *coloane*/2). Validarea hărții (toți Curierii pot ajunge la toți Clienții) este făcută prin BFS flood fill.

Am folosit algoritmul Hungarian pentru a face o matrice de fezabilitate a livrării pachetelor, pentru a alege ce curier va primi ce pachet. Uneori am văzut că niciun pachet nu e fezabil și simularea s-ar bloca (Dispecerul nu mai dădea nimănui niciun pachet,

simularea rula până la ultimul tic și pachetele erau pierdute), așa că am adăugat o oarecare rută disperată pentru a livra toate pachetele.

Cel puțin cu datele de test, am văzut că pachetele sunt livrate destul de repede, așa că nu instanțiez toți curierii din prima, am implementat un threshold care, odată atins, sunt adăugați mai mulți curieri pentru a face față fluxului de pachete. Fără acest lucru, majoritatea curierilor ar fi Idle în Bază, consumând activ $/tick și făcând profitul imposibil.

2 Diagrama claselor

Accesați `class_diagram.png` sau `class_diagram.svg`, nu am reușit să dau embed în document diagramei astfel încât să fie lizibilă.

3 Algoritmul de generare a hartii

Funcția `generate` construiește o hartă procedurală pe baza configurației primite. Ea plasează baza, clienții, stațiile și generează pereți aleatoriu.

3.1 Semnătura funcției

```
void ProceduralMapGenerator::generate(Config& cfg, std::mt19937& rng,
                                      std::vector<std::string>& grid,
                                      Vec2& basePos,
                                      std::vector<Vec2>& clients,
                                      std::vector<Vec2>& stations)
```

3.2 Parametri

- **cfg** - struct ce conține opțiunile de configurare din fișierul simulation_setup.txt (dimensiuni hartă, număr clienți, număr stații etc.)
- **rng** - generatorul de numere aleatorii (Mersenne Twister)
- **grid** - matricea de caractere care reprezintă harta
- **basePos** - poziția bazei
- **clients** - vector cu pozițiile clienților
- **stations** - vector cu pozițiile stațiilor

3.3 Etapele generării

3.3.1 Inițializarea hărții

```
grid.assign(cfg.rows, std::string(cfg.cols, '.'));
```

Creează o hartă de `rows * cols` plină cu caracterul `'.'` (celulă liberă).

3.3.2 Plasarea bazei în centru

```
basePos = {cfg.rows/2, cfg.cols/2};
grid[basePos.x][basePos.y] = 'B';
```

Baza este plasată în centrul hărții și marcată cu `'B'`.

3.3.3 Inițializarea distribuțiilor aleatorii

```
std::uniform_int_distribution<int> rx(0, cfg.rows-1);
std::uniform_int_distribution<int> ry(0, cfg.cols-1);
```

Distribuții uniforme (șanse egale ca orice valoare din interval să fie aleasă) pentru generarea coordonatelor valide în grilă.

3.3.4 Plasarea clienților

```
clients.clear();
for (int i = 0; i < cfg.clientsCount; ++i) {
    while (true) {
        int x = rx(rng);
        int y = ry(rng);
        if (grid[x][y] == '.') {
            grid[x][y] = 'D';
            clients.push_back({x,y});
            break;
        }
    }
}
```

Pentru fiecare client:

- se caută o poziție aleatoare liberă;
- celula este marcată cu `'D'`;
- poziția este salvată în vectorul `clients`.

3.3.5 Plasarea stațiilor

```
stations.clear();
for (int i = 0; i < cfg.maxStations; ++i) {
    while (true) {
        int x = rx(rng);
        int y = ry(rng);
        if (grid[x][y] == '.') {
            grid[x][y] = 'S';
            stations.push_back({x,y});
            break;
        }
    }
}
```

Funcționează identic cu plasarea clienților, dar stațiile sunt marcate cu `'S'` și au o culoare galbenă (pentru a le distinge de `'S'`-ul magenta al Scooterelor).

3.3.6 Generarea aleatoare a pereților

```
std::uniform_real_distribution<> frac(0.0, 1.0);
for (int x = 0; x < cfg.rows; ++x) {
    for (int y = 0; y < cfg.cols; ++y) {
        if (grid[x][y] == '.' && frac(rng) < wallProb)
            grid[x][y] = '#';
    }
}
```

Pentru fiecare celulă liberă:

- se generează un număr real între 0 și 1;
- dacă valoarea este sub `wallProb`, celula devine perete (`'#'`).

3.4 Rezumat

Funcția:

1. creează o hartă goală;
2. plasează baza în centru;
3. plasează clienți și stații în poziții aleatorii neocupate;
4. adaugă pereți aleatoriu pe celulele libere.

4 Algoritmul `hiveMindDispatch()`

Algoritmul `hiveMindDispatch()` reprezintă mecanismul central de alocare globală a pachetelor către curieri în cadrul simulării. Acesta transformă problema într-o instanţă a *problemei de asignare* (assignment problem), pe care o rezolvă folosind algoritmul maghiar/Kuhn/Kuhn-Munkres/Hungarian, completat cu două mecanisme de rezervă (uneori toate pachetele devin nefezabile și niciun Curier nu mai livrează nimic).

4.1 Obiectiv

Scopul algoritmului este de a maximiza profitul total estimat prin asignarea optimă a pachetelor aflate în aşteptare către capacităţile libere ale curierilor, respectând constrângerile de:

- baterie,
- distanţă şi accesibilitate,
- capacitate de transport,
- tipul curierului (Dronă, Robot, Scooter),
- deadline-ul pachetului.

4.1.1 Construirea listei de pachete şi a sloturilor de curieri

Se defineşte:

$P =$ numărul de pachete aflate în *packagePool*, $M =$ numărul total de sloturi libere ale curierilor.

Fiecare curier cu capacitate liberă contribuie cu câte un *slot* în matricea de asignare. Astfel, dacă un curier are capacitate k şi transportă deja r pachete, atunci el generează $k - r$ coloane în matricea de cost.

Dacă $P = 0$ sau $M = 0$, algoritmul se opreşte.

4.1.2 Construirea matricei de cost

Se construieşte o matrice pătrată:

$$C \in \mathbb{R}^{n \times n}, \qquad n = \max(P, M),$$

unde:

$$C_{ij} = \begin{cases} -\text{score}(p_i, c_j), & \text{dacă asignarea este fezabilă,} \\ \text{INF}, & \text{altfel.} \end{cases}$$

Fezabilitatea include:

- distanţa până la destinaţie este accesibilă,
- curierul poate reveni la bază cu bateria rămasă,
- Dronele nu iau pachete cu recompensă < 300,
- Roboţii nu iau pachete prea îndepărtate,
- scorul calculat de `computePriority()` nu este un număr extrem de mic (negativ).

Pentru coloanele sau rândurile fictive (dacă $P \neq M$), costul este 0.

4.1.3 Aplicarea algoritmului maghiar

Se aplică algoritmul maghiar pentru minimizarea costului total:

$$\text{match} = \text{Hungarian}(C).$$

Rezultatul este un vector:

$$match[i] = j,$$

care indică faptul că pachetul i este asignat slotului j.
Asignările sunt acceptate doar dacă:

$$C_{ij} < \text{INF}.$$

4.1.4 Primul mecanism de rezervă

Dacă algoritmul maghiar nu a produs nicio asignare validă, se aplică un fallback greedy:

1. Se colectează toate perechile fezabile (p_i, c_j).
2. Se sortează descrescător după profitul estimat:

$$profit = -C_{ij}.$$

3. Se selectează cele mai bune perechi, evitând reutilizarea aceluiaşi slot sau pachet.

   Se acceptă doar asignări cu:

$$profit \geq -1000.$$

4.1.5 Eliminarea pachetelor asignate

Pachetele asignate sunt eliminate din `packagePool`:

$$packagePool \leftarrow packagePool \setminus \{p_i \mid assigned[i] = 1\}.$$

4.1.6 Mecanismul de rezervă final

Dacă:

$$assignedCount = 0, \quad P > 0, \quad spawnedPackages = totalPackages,$$

şi nu există curieri activi, atunci se aplică un mecanism de urgenţă:

- fiecare pachet este asignat celui mai apropiat curier fezabil,
- se ignoră constrângerile de baterie şi euristicile stricte.

Scopul este de a evita blocarea simulării în situaţii-limită.

4.2 Complexitate

Complexitatea este dominată de algoritmul maghiar:

$$O(n^3), \qquad n = \max(P, M).$$

4.3 Rezumat

Algoritmul `hiveMindDispatch()` realizează o strategie hibridă:

1. optimizare globală prin algoritmul maghiar,
2. fallback greedy pentru robusteţe,
3. fallback de urgenţă pentru evitarea blocării execuției, mai ales dacă per total simularea a generat profit considerabil.