
Institut d'informatique d'Auvergne
(ISIMA)

---

# Simulation multi-agents

---

***Etudiant :***
AMAL Saad
ID BENOUAKRIM Brahim
NASRI Ayoub

***Enseignant :***
RONDEAUX Claire

5 janvier 2025

# Table des matières

**1 Introduction** **2**

**2 Survie dans un Monde à Ressources Limitées** **2**
2.1 Les Ressources Naturelles : Une Quantité Limitée . . . . . 2
2.2 Croissance Démographique et Pression sur les Ressources . . . . . 2

**3 Outils et Technologie** **3**
3.1 Environnement de Développement . . . . . 3
3.2 Collaboration et Gestion de Version . . . . . 3
3.3 Avantages des Outils Utilisés . . . . . 3

**4 Conception** **4**
4.1 Différence par rapport à l'analyse . . . . . 4
4.2 Classe `Environnement` et le `Singleton Pattern` . . . . . 4
4.3 Classe `Agent` et le `Strategy Pattern` . . . . . 4
4.4 État des Agents : `State Pattern` . . . . . 5
4.5 Classe `Ressource` et le `State Pattern` . . . . . 5
4.6 Interaction entre les Classes . . . . . 5

**5 Développement du Code et Algorithmes de Survie** **7**
5.1 BFS . . . . . 7
5.2 Stratégie de l'Agent Vorace : Maximisation de la Valeur . . . . . 7
5.3 Stratégie de l'Agent Local : Proximité Maximale . . . . . 8
5.4 Strategie de l'Agent rentable : la plus profitable . . . . . 8
5.5 RandomMove . . . . . 9
5.6 Génération du map . . . . . 9

**6 Test unitaire** **11**
6.1 Tests sur la barre de santé et initialisation . . . . . 11
6.2 Tests sur le déplacement . . . . . 11
6.3 Tests sur les stratégies . . . . . 11

**7 Résultats de la simulation :** **12**
7.1 Affichage de la Carte et Visualisation des Agents . . . . . 12
7.2 Ressources . . . . . 13
7.3 Agents . . . . . 14
7.4 Survie . . . . . 15
7.4.1 Taux de Victoire des Agents . . . . . 15
7.4.2 Mortalité des Agents . . . . . 15
7.4.3 Répartition des Simulations avec et sans Survivants . . . . . 16

**8 Conclusion** **17**

# 1 Introduction

Dans le cadre de ce travail pratique, nous nous penchons sur l'étude d'un environnement limité, un concept fondamental dans de nombreux domaines scientifiques et techniques.

Un environnement limité se définit comme un espace ou un système dans lequel les ressources, les interactions ou les mouvements sont soumis à des contraintes spécifiques ou à des frontières clairement établies. L'étude de ces environnements est essentielle pour comprendre les dynamiques internes d'un système, modéliser des phénomènes complexes ou tester des hypothèses dans un cadre contrôlé.

L'objectif principal de ce TP est de modéliser et d'analyser les particularités d'un environnement limité afin de mieux appréhender ses propriétés fondamentales, ses contraintes, ainsi que les stratégies permettant d'optimiser l'efficacité ou de résoudre des problèmes spécifiques découlant de ces restrictions. Pour ce faire, nous utiliserons des outils et des méthodes adaptés, en mettant l'accent sur la résolution de défis pratiques et le développement de solutions innovantes.

# 2 Survie dans un Monde à Ressources Limitées

Dans un monde où les ressources naturelles sont limitées et la population ne cesse de croître, la survie devient un défi collectif. Depuis l'aube de l'humanité, les sociétés ont prospéré ou décliné en fonction de leur capacité à exploiter, gérer et préserver les ressources disponibles. Aujourd'hui, face à des pressions environnementales et démographiques sans précédent, il est crucial de comprendre comment ces dynamiques influencent notre avenir. La survie dans un tel contexte repose sur notre aptitude à innover, à coopérer et à repenser nos modes de vie pour assurer la pérennité de la planète.

## 2.1 Les Ressources Naturelles : Une Quantité Limitée

Les ressources naturelles, qu'il s'agisse de l'eau douce, des terres arables, des forêts ou des énergies fossiles, sont limitées. Par exemple :

- **Eau douce :** Seulement 2,5% de l'eau sur Terre est douce, et moins de 1% est facilement accessible pour l'usage humain.
- **Énergies fossiles :** Selon l'Agence internationale de l'énergie, au rythme actuel de consommation, les réserves de pétrole pourraient s'épuiser d'ici la fin du siècle.
- **Terres arables :** Environ 33% des sols mondiaux sont déjà dégradés en raison de l'érosion, de la surexploitation et de la pollution.

Ces limites physiques imposent des contraintes sévères aux systèmes sociaux et économiques.

## 2.2 Croissance Démographique et Pression sur les Ressources

La population mondiale est passée de 1 milliard en 1800 à plus de 8 milliards en 2023. Cette croissance exponentielle augmente considérablement la demande pour :

- **Alimentation :** La production agricole devra augmenter de 60% d'ici 2050 pour nourrir la population mondiale.
- **Énergie :** La consommation mondiale d'énergie a été multipliée par 3 depuis 1960.

— **Eau douce :** Près de 4 milliards de personnes font face à une pénurie d'eau au moins un mois par an.

Cette pression accentue les inégalités et intensifie les conflits pour l'accès aux ressources essentielles.

Cette réalité complexe, marquée par des interactions entre croissance démographique, pression sur les ressources et impacts environnementaux, peut sembler écrasante. Toutefois, pour mieux comprendre ces dynamiques et explorer des solutions, il est utile de les simplifier. En isolant les principaux facteurs et en les modélisant, nous pouvons créer des outils qui simulent les interactions essentielles entre les agents et leur environnement. Ces modèles simplifiés offrent une perspective précieuse pour analyser les mécanismes sous-jacents et envisager des stratégies de gestion durable dans un monde aux ressources limitées.

# 3 Outils et Technologie

La réalisation de ce projet a nécessité l'utilisation de divers outils et technologies modernes pour assurer une collaboration efficace et une gestion optimale du développement.

## 3.1 Environnement de Développement

Nous avons utilisé **JDK 21** pour le développement en Java, offrant les dernières fonctionnalités et améliorations de performance. **Maven** a été choisi comme outil de gestion de dépendances et de construction, facilitant l'organisation et l'automatisation des tâches liées au projet.

## 3.2 Collaboration et Gestion de Version

Pour le travail collaboratif, nous avons adopté **Git** comme système de gestion de versions distribué. Notre équipe a centralisé son travail sur la plateforme **GitLab**, qui a permis de :

— Partager et synchroniser le code entre les membres.
— Gérer les branches pour différentes fonctionnalités ou correctifs.
— Suivre les problèmes et tâches grâce à l'outil intégré de suivi des tickets.

## 3.3 Avantages des Outils Utilisés

Ces outils ont offert plusieurs avantages :

— **Efficacité :** La gestion automatisée des dépendances et des versions avec Maven a simplifié le processus de développement.
— **Fiabilité :** L'utilisation de GitLab a permis une traçabilité complète des modifications et une résolution rapide des conflits.
— **Collaboration :** La plateforme GitLab a favorisé une communication fluide au sein de l'équipe et a assuré un suivi structuré du projet.

Ces outils combinés ont constitué une base solide pour mener à bien le projet tout en maintenant un haut niveau d'organisation et de qualité.

# 4 Conception

La conception du système repose sur une architecture modulaire et flexible, utilisant des design patterns bien établis pour gérer les comportements des agents et des ressources dans l'environnement. Voici un aperçu des principales classes et de leurs interactions.

## 4.1 Différence par rapport à l'analyse

Dans le cadre de cette étude, nous avons modifié les agents pour explorer leurs performances dans un environnement donné. L'environnement reste inchangé et est modélisé sous la forme d'un plan bidimensionnel, avec des frontières fixes, des ressources à collecter, et des obstacles à éviter. Chaque agent est doté d'un foyer fixe, servant de point de départ ou de retour pour gérer ses ressources et sa barre de santé.

Tous les agents commencent avec une barre de santé initiale de 100 unités, qui diminue à chaque mouvement dans l'environnement. Pour restaurer leur santé, ils doivent localiser des ressources, les collecter, et retourner à leur foyer pour les "consommer". Les ressources dans cet environnement ont une valeur et un poids, et chaque agent est soumis à une contrainte de capacité maximale. Ainsi, les agents ne peuvent collecter que les ressources dont le poids total est inférieur ou égal à leur capacité restante.

Cette configuration impose aux agents de faire des choix stratégiques concernant les ressources à collecter, tout en optimisant leurs déplacements pour éviter l'épuisement de leur barre de santé. Ces contraintes permettent d'évaluer les dynamiques entre gestion des ressources, optimisation des trajets, et survie dans un environnement limité. Cette étude contribue à mieux comprendre ces interactions et à explorer des approches efficaces pour gérer des systèmes autonomes dans des contextes contraints.

## 4.2 Classe `Environnement` et le `Singleton Pattern`

La classe `Environnement` représente l'environnement global dans lequel les agents évoluent. Il s'agit d'un **singleton** afin de garantir qu'il n'existe qu'une seule instance de l'environnement dans l'application. Cette classe est responsable de la gestion des ressources, de la simulation du passage du temps et de la coordination des interactions entre les agents.

## 4.3 Classe `Agent` et le `Strategy Pattern`

Les agents sont des entités qui interagissent avec l'environnement pour survivre. Pour offrir de la flexibilité dans le comportement des agents, nous avons utilisé le **Strategy Pattern**. Ainsi, la classe `Agent` est dotée d'une référence vers une stratégie qui détermine son comportement. Nous avons trois types d'agents :

- **Agent Vorace** : Un agent qui consomme la ressources de plus grande valeur.
- **Agent Local** : Un agent qui préfère consommer la ressource la plus proche.
- **Agent Rentable** : Un agent qui évalue les ressources en fonction de leur rapport coût/bénéfice et choisit d'agir en conséquence.

Chaque agent peut donc changer de stratégie en fonction de son état ou des conditions de l'environnement, ce qui permet une grande flexibilité et une gestion dynamique des comportements.

## 4.4 État des Agents : `State Pattern`

Les agents peuvent se trouver dans deux états principaux : **Vivant** ou **Mort**. Nous avons utilisé le **State Pattern** pour gérer ces transitions d'état. La classe `Agent` contient un état qui définit son comportement spécifique à chaque étape de sa vie. Si un agent meurt, il passe à l'état "mort" et ne peut plus interagir avec l'environnement, ce qui simplifie la gestion de ses actions et rend l'architecture plus extensible.

## 4.5 Classe `Ressource` et le `State Pattern`

Les ressources dans l'environnement peuvent être dans deux états : **Collectée** ou **Non Collectée**. Ce changement d'état est également géré par le **State Pattern**. Lorsqu'une ressource est collectée par un agent, elle passe à l'état "Collectée", et l'agent ne peut plus interagir avec cette ressource. Cette gestion permet d'assurer une utilisation efficace des ressources disponibles tout en minimisant les erreurs liées à la tentative de collecte d'une ressource déjà épuisée.

## 4.6 Interaction entre les Classes

Les classes `Environnement`, `Agent`, `Ressource` et `Obstacle` interagissent de manière cohérente pour simuler les dynamiques de survie dans un environnement à ressources limitées.

- `Environnement` : Gère l'état global et les ressources disponibles. C'est une classe Singleton qui centralise la gestion des agents et des ressources.
- `Agent` : Chaque agent adopte une stratégie différente (pattern **Strategy**) pour interagir avec l'environnement, choisissant parmi des comportements comme être vorace, local ou rentable.Dans cet environnement, chaque déplacement effectué par un agent coûte une portion de sa barre de santé, ce qui rend chaque choix crucial. Les agents doivent donc prendre des décisions rigoureuses et optimisées, afin de minimiser la perte de santé tout en maximisant leurs chances de survie.
- `Ressource` : Chaque ressource peut être dans un état `Non Collectée` ou `Collectée`, géré par le pattern **State**. Lorsqu'un agent collecte une ressource, son état change.
- `Obstacle` : Ajoute des obstacles à l'environnement, affectant le déplacement des agents et leur stratégie.

Ces interactions, rendues possibles par l'application des design patterns, permettent une gestion dynamique de l'environnement et des agents, tout en garantissant une architecture modulaire et évolutive.

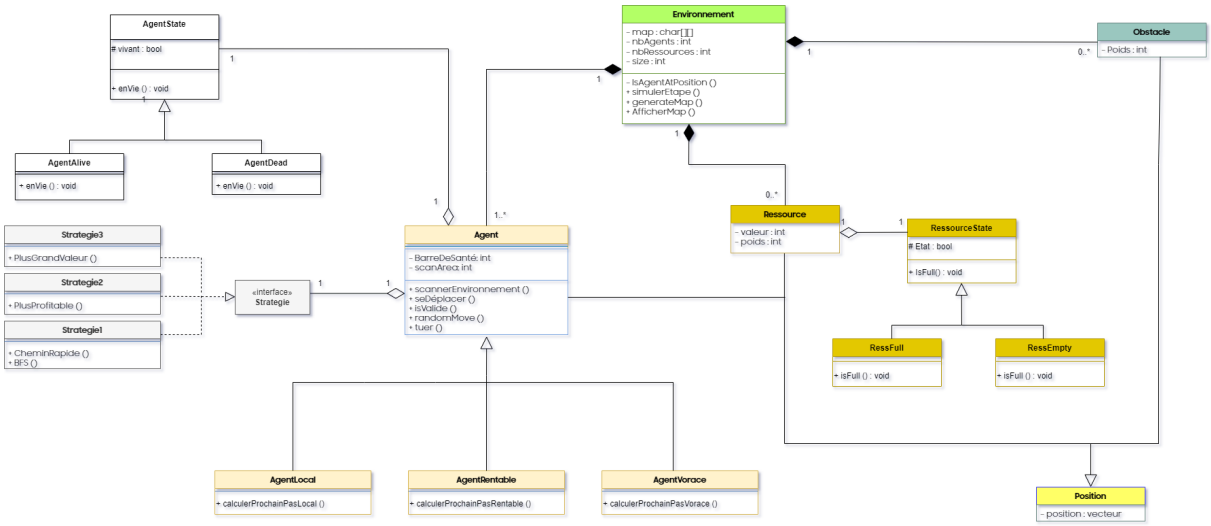


FIGURE 1 – DiagrammeUML

# 5 Développement du Code et Algorithmes de Survie

## 5.1 BFS

L'algorithme BFS (Breadth-First Search) est l'algorithme utilisé par les agents pour explorer leur environnement. Chaque agent scanne son environnement en fonction de son rayon de détection, puis sélectionne les ressources en fonction de sa stratégie.

L'algorithme fonctionne de manière simple : il commence par explorer la partie de la carte dans le rayon de détection de l'agent. Ensuite, il recherche les ressources présentes dans cette zone en calculant leur valeur et le nombre de pas nécessaires pour les atteindre. L'agent choisit ensuite la ressource à collecter en fonction de ces critères, tout en tenant compte de sa stratégie spécifique.

```
1   public List<List<Integer>> BFS(char[][] map, int rayon) {
2       List<List<Integer>> ans = new ArrayList<>();
3       int[][] visite = new int[2 * rayon + 1][2 * rayon + 1];
4       int[][] directions = { { 0, 1 }, { 1, 0 }, { 0, -1 }, { -1, 0 } };
5       int size, r, c, pas;
6       int x = rayon, y = rayon;
7       int[] cur;
8       Deque<int[]> q = new LinkedList<>();
9       q.offer(new int[] { x, y, 0 });
10      visite[x][y] = 1;
11      while (!q.isEmpty()) {
12          size = q.size();
13          for (int k = 0; k < size; k++) {
14              cur = q.poll();
15              for (int[] d : directions) {
16                  r = cur[0] + d[0];
17                  c = cur[1] + d[1];
18                  pas = cur[2] + 1;
19                  if (r >= 0 && r < (2 * rayon + 1) && c >= 0
20                          && c < (2 * rayon + 1) && visite[r][c] == 0
21                          && (map[r][c] == 'O' || map[r][c] == '*')) {
22                      if (map[r][c] == 'O') {
23                          List<Integer> temp = Arrays.asList(r, c, pas);
24                          ans.add(temp);
25                      }
26                      q.offer(new int[] { r, c, pas });
27                      visite[r][c] = 1;
28                  }
29              }
30          }
31      }
32      return ans;
33  }
```

## 5.2 Stratégie de l'Agent Vorace : Maximisation de la Valeur

L'agent vorace, comme son nom l'indique, adopte un comportement axé sur la consommation. Lors de son scan de l'environnement, il cherche à maximiser sa récolte en se concentrant sur la ressource ayant la valeur la plus élevée. Ainsi, il choisit toujours la ressource dont la valeur est maximale, sans tenir compte d'autres critères, ce qui fait de lui un agent particulièrement "vorace" dans sa quête de survie.

```java
public List<Integer> plusGrandValeur(List<List<Integer>> ress, int[] valeurs) {
        int index = -1;
        int max = Integer.MIN_VALUE;
        for (int i = 0; i < ress.size(); i++) {
            if (valeurs[i] > max) {
                index = i;
                max = valeurs[i];
            }
        }
        if (index == -1)
            return Arrays.asList(-1, -1, -1);
        return ress.get(index);
    }
```

## 5.3 Stratégie de l'Agent Local : Proximité Maximale

L'agent local, un peu paresseux par nature, adopte une approche simpliste pour sa survie. Lors de son scan de l'environnement, il cherche à trouver la ressource la plus proche de lui. Plutôt que de parcourir de longues distances, il privilégie toujours la ressource la plus proche, minimisant ainsi ses efforts tout en maximisant ses chances de collecte rapide.

```java
public List<Integer> CheminRapide(List<List<Integer>> res, char[][] originalMap) {
        List<Integer> ans = Arrays.asList(-1, -1, -1);
        int d = Integer.MAX_VALUE;
        for (var l : res) {
            int distance = l.get(2);
            if (d > distance) {
                ans = l;
                d = distance;
            }
        }
        return ans;
    }

```

## 5.4 Strategie de l'Agent rentable : la plus profitable

Cet agent, que l'on peut qualifier d'agent "en papier", adopte une stratégie particulièrement prudente. Il n'aime pas prendre de risques et cherche toujours à minimiser les dommages à sa barre de santé. Lorsqu'il choisit une ressource, il privilégie celles qui entraîneront la plus faible perte de santé en fonction du déplacement nécessaire. Ainsi, l'agent cherche à éviter les ressources qui impliquent des trajets longs ou dangereux, optant pour celles qui réduiront au minimum sa barre de santé, assurant ainsi sa survie sur le long terme.

```java
public List<Integer> PlusProfitable(List<List<Integer>> ress,
                                    int[] valeurs,
                                    int energie,
                                    char[][] originalMap) {
    List<Integer> ans = Arrays.asList(-1, -1, -1);
    int maxProfit = Integer.MIN_VALUE;
    int profit;
    for (int i = 0; i < ress.size(); i++) {
        profit = energie + valeurs[i] - ress.get(i).get(2);
        if ((profit > maxProfit) && (energie - ress.get(i).get(2)) >= 0)) {
            ans = ress.get(i);
            maxProfit = profit;
        }
    }
    return ans;
}
```

## 5.5 RandomMove

Si aucune ressource n'est trouvée dans le rayon de détection de l'agent, la fonction `randomMove` entre en jeu. Dans ce cas, l'agent se déplace vers une position aléatoire qui est vacante, puis effectue un nouveau scan de son environnement. Cela permet à l'agent de continuer à explorer sans rester bloqué dans une zone sans ressources, augmentant ainsi ses chances de trouver une ressource à collecter.

```java
public void randomMove(char[][] originalMap) {
    Random rand = new Random();
    int[][] directions = { { 0, 1 }, { 1, 0 }, { 0, -1 }, { -1, 0 } };
    Set<Integer> count = new HashSet<>();
    while (true) {
        int index = rand.nextInt(4);
        int[] d = directions[index];
        int newX = (this.getX() + d[0]);
        int newY = (this.getY() + d[1]);
        if (isValid(originalMap, newX, newY)) {
            seDeplacer(newX, newY);
            setBarreDeSante(getBarreDeSante() - 5);
            break;
        }
        count.add(index);
        if (count.size() == 4)
            break;
    }
}

```

## 5.6 Génération du map

Pour la génération de la carte, il ne suffit pas de placer aléatoirement des obstacles et des ressources. Les obstacles doivent être stratégiquement positionnés pour créer des défis réalistes, forçant les agents à adapter leur stratégie. Ils doivent avoir un sens, ajoutant de la complexité à l'écosystème.

Les ressources, quant à elles, doivent être accessibles mais pas triviales, afin de stimuler les choix tactiques des agents. Il est crucial que toutes les ressources soient atteignables,

mais pas trop faciles à obtenir.

Le processus de génération commence par une position fixe, suivie de placements aléatoires de murs verticaux et horizontaux, formant des zones variées. Ces murs ne doivent pas bloquer l'accès complet à certaines parties de la carte. À chaque étape, des ressources sont ajoutées de manière à équilibrer leur disponibilité et les obstacles, créant ainsi un environnement dynamique et stratégique.

```
    public void generateMap(int starti, int startj) {

        int[][] directions = { { 1, 0 }, { -1, 0 }, { 0, 1 }, { 0, -1 } };

        List<int[]> shuffledDirections = Arrays.asList(directions);
        Collections.shuffle(shuffledDirections, rand);

        for (int[] dir : shuffledDirections) {
            int newi = starti + dir[0];
            int newj = startj + dir[1];

            if (newi >= 0 && newi < size && newj >= 0 && newj < size &&
            map[newi][newj] != '*'
                    && map[newi][newj] != 'X') {
                int r = rand.nextInt(20);
                map[newi][newj] = '*';
                if (newi >= 3 && newi < size - 3 && newj >= 3 && newj < size - 3) {
                    if (r == 0) {
                        if (newi - 1 >= 0) {
                            map[newi - 1][newj] = 'X';
                        }
                        if (newi + 1 < size) {
                            map[newi + 1][newj] = 'X';
                        }
                    } else if (r == 1) {
                        if (newj - 1 >= 0) {
                            map[newi][newj - 1] = 'X';
                        }
                        if (newj + 1 < size) {
                            map[newi][newj + 1] = 'X';
                        }
                    }
                    if (map[newi][newj] != 'X' && isValid(newi - 1, newj) &&
                    map[newi - 1][newj] != 'X'&& isValid(newi + 1, newj) &&
                    map[newi + 1][newj] != 'X' && isValid(newi, newj - 1) &&
                    map[newi][newj - 1] != 'X' && isValid(newi, newj + 1) &&
                    map[newi][newj + 1] != 'X') {
                        map[newi][newj] = '0';
                    }
                }
                generateMap(newi, newj);
            }
        }
    }

```

# 6 Test unitaire

Les tests unitaires sont essentiels pour garantir que chaque composant du système fonctionne correctement de manière isolée avant l'intégration. Ces tests assurent la robustesse du code, détectent les erreurs précoces et facilitent la maintenance du système.

## 6.1 Tests sur la barre de santé et initialisation

Les tests relatifs à la barre de santé se concentrent sur la gestion de la santé des agents lors des déplacements. Nous vérifions que l'initialisation de la santé est correcte, ainsi que la mise à jour de la santé après chaque déplacement. Ces tests s'assurent également que la santé ne peut pas devenir négative et que l'agent est correctement initialisé avec des valeurs valides. De plus, chaque agent doit démarrer avec une santé maximale et la perdre progressivement à chaque déplacement effectué.

Les tests d'initialisation vérifient également que tous les autres paramètres des agents, comme leur position, leurs ressources et les obstacles dans leur environnement, sont correctement initialisés avant le début de la simulation.

Exemples de tests :

— Vérifier que la barre de santé d'un agent est correctement initialisée à une valeur maximale.
— Vérifier que la santé d'un agent est correctement réduite après chaque déplacement.
— Tester que la santé d'un agent ne devient pas négative après plusieurs déplacements.
— Vérifier que l'agent est initialisé avec une position et des ressources valides.

## 6.2 Tests sur le déplacement

Les tests sur le déplacement s'assurent que les agents se déplacent correctement sur la carte, en respectant les contraintes imposées par leur environnement (par exemple, les obstacles). Ces tests vérifient que les agents peuvent se déplacer d'une case à l'autre et que les déplacements sont bien réalisés en tenant compte des obstacles et des autres agents.

Les tests de déplacement incluent également la vérification du comportement aléatoire dans les cas où aucune ressource n'est trouvée dans le rayon de détection, comme dans la stratégie `RandomMove`.

Exemples de tests :

— Vérifier qu'un agent peut se déplacer d'une position à une autre sur une carte vide.
— Tester que l'agent respecte les obstacles et ne peut pas traverser des murs.
— Vérifier que l'agent effectue un mouvement aléatoire valide lorsqu'aucune ressource n'est trouvée.
— Tester que l'agent effectue un scan de son environnement après chaque déplacement.

## 6.3 Tests sur les stratégies

Les tests sur les stratégies valident que chaque agent adopte la bonne logique comportementale en fonction de sa stratégie spécifique. Par exemple, un agent vorace devrait toujours choisir la ressource de valeur maximale, tandis qu'un agent local devrait se diriger vers la ressource la plus proche. De plus, l'agent "en papier" doit optimiser ses déplacements en fonction des coûts associés à la perte de santé.

Ces tests vérifient que les agents appliquent correctement leurs stratégies et que leurs choix de ressources sont conformes à leur logique interne.

Exemples de tests :

— Vérifier que l'agent vorace choisit toujours la ressource avec la plus grande valeur.
— Tester que l'agent local choisit la ressource la plus proche.
— Vérifier que l'agent rentable minimise ses déplacements pour préserver sa barre de santé.

```
[INFO] Running toto.AgentTest
[INFO] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.644 s -- in toto.AgentTest
[INFO] Running toto.TestLocal
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.007 s -- in toto.TestLocal
[INFO] Running toto.TestRentable
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.018 s -- in toto.TestRentable
[INFO] Running toto.TestStrategie
[INFO] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.004 s -- in toto.TestStrategie
[INFO] Running toto.TestVorace
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.002 s -- in toto.TestVorace
[DEBUG] Closing the fork 1 after saying GoodBye.
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 26, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
```

FIGURE 2 – Illustration des résultats des tests unitaires

# 7 Résultats de la simulation :

Dans le cadre de notre simulation, chaque environnement est une grille de 50x50 qui comprend des obstacles et comprend en moyenne 67 ressources, réparties de manière stratégique pour offrir aux agents des défis variés. Pour garantir une diversité d'interactions et de comportements, trois types d'agents sont présents : vorace, local et rentable. Chaque type d'agent suit une stratégie spécifique pour collecter les ressources et optimiser sa survie. Afin d'assurer une certaine uniformité dans les capacités d'exploration, tous les agents partagent un rayon de scan identique de 4, leur permettant d'évaluer l'environnement et de prendre des décisions en fonction des ressources disponibles dans leur périmètre de détection. La simulation prend fin lorsque un seul agent reste en vie, c'est-à-dire lorsqu'il n'y a plus d'autres agents capables de collecter des ressources et de maintenir leur survie.

## 7.1 Affichage de la Carte et Visualisation des Agents

Dans notre simulation, l'affichage de l'environnement et des agents se fait directement dans le terminal, sans interface graphique avancée. Cette approche permet de visualiser les interactions entre les agents et les ressources de manière simple et efficace.

Les agents sont représentés par des couleurs distinctes, facilitant leur identification rapide dans l'environnement :

— **Les agents voraces** sont représentés par des **A rouges**. Ces agents ont une stratégie axée sur la collecte des ressources les plus abondantes et les plus proches, au risque de sacrifier leur propre santé.

— **Les agents rentables** sont symbolisés par des A jaunes. Ces agents adoptent une approche plus stratégique, cherchant à maximiser leur efficacité tout en minimisant les risques pour leur santé.
— **Les agents locaux** sont identifiés par des A verts. Ils privilégient la collecte des ressources les plus proches, évitant ainsi des déplacements risqués.

De même, les ressources sont affichées avec des couleurs correspondant à leur valeur.

Chaque fois qu'un agent interagit avec une ressource, il est mis à jour dans le terminal, avec la couleur de l'agent et de la ressource changée pour refléter l'état actuel de la simulation.

L'utilisation de ces codes de couleur dans le terminal permet une visualisation claire de l'état de la simulation et aide à mieux comprendre les dynamiques des agents et de l'environnement. Cela donne également un aperçu immédiat du succès ou de l'échec de chaque stratégie adoptée par les différents types d'agents.

FIGURE 3 – Affichage du map

## 7.2 Ressources

Il existe donc une concurrence directe entre les agents pour la collecte des ressources, ce qui ajoute un aspect stratégique à leur comportement. De plus, il y a une uniformité dans la détection des ressources, chaque agent étant capable de scanner la même zone. En

moyenne, il reste 1 ressource sur les 67 initiales à la fin de la simulation, ce qui témoigne de l'intensité de la compétition entre les agents.

## 7.3 Agents

Dans notre simulation, l'objectif principal est d'identifier l'agent le plus performant. En traçant l'évolution de la barre de santé à la fin de chaque simulation, nous avons obtenu le graphique suivant, qui montre la dynamique de survie de chaque agent et permet de comparer leur efficacité respective.

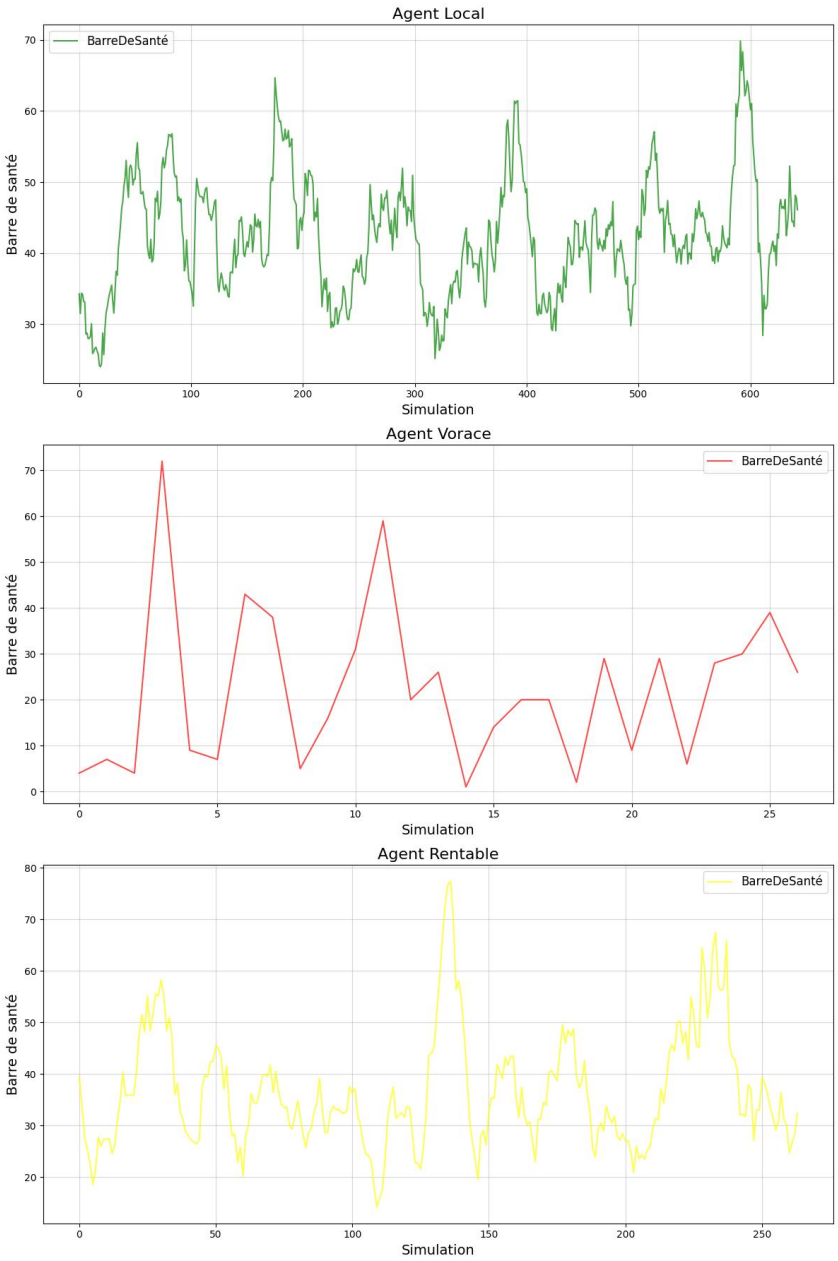


FIGURE 4 – Évolution de la barre de santé des agents

On peut observer un comportement presque chaotique dans la simulation. Il est important de noter que toutes les fonctions ont été lissées pour une représentation plus

esthétique des résultats. Parfois, certains agents parviennent à survivre avec une barre de santé élevée, tandis que d'autres échouent, ce qui accentue la concurrence entre eux. Cette variation illustre bien la dynamique compétitive et stratégique présente dans l'environnement.

## 7.4 Survie

Dans cette partie, nous analysons les résultats de la simulation concernant la survie des agents et la répartition des ressources. Plusieurs graphiques ont été générés pour illustrer différents aspects de la simulation.

### 7.4.1 Taux de Victoire des Agents

Le premier graphique montre le taux de victoire de chaque type d'agent au cours des simulations. Ce taux est calculé en fonction du nombre de fois où un agent a survécu jusqu'à la fin de la simulation.

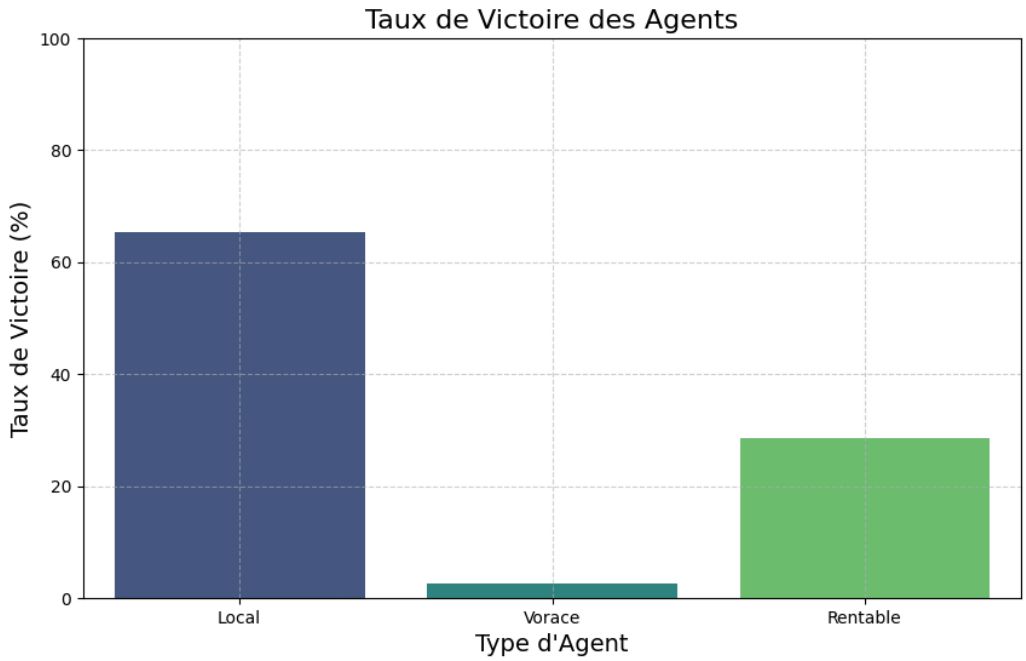


FIGURE 5 – Taux de Victoire des Agents

### 7.4.2 Mortalité des Agents

Le second graphique montre le nombre d'agents morts pour chaque type d'agent (local, vorace et rentable). Il permet d'analyser la résilience de chaque type d'agent au sein de l'environnement simulé.

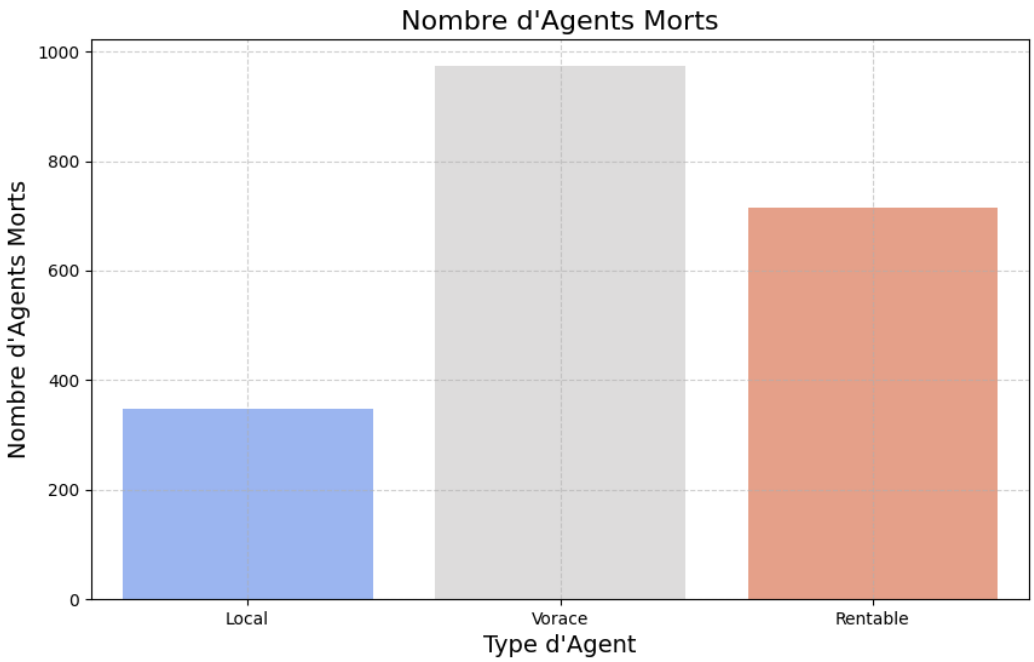


FIGURE 6 – Nombre d'Agents Morts

### 7.4.3 Répartition des Simulations avec et sans Survivants

Enfin, ce graphique en camembert montre la répartition des simulations avec et sans survivants. Il est utile pour visualiser la fréquence des simulations où aucun agent n'a survécu à la fin.

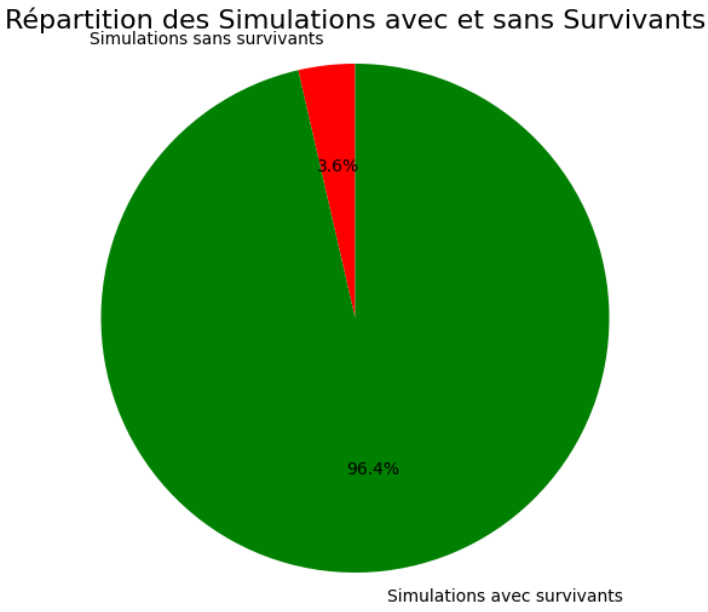


FIGURE 7 – Répartition des Simulations avec et sans Survivants

# 8 Conclusion

La simulation menée sur la survie des agents dans un environnement à ressources limitées a révélé des dynamiques fascinantes et parfois imprévisibles. En observant le comportement des agents, qu'ils soient voraces, locaux ou rentables, on constate que les décisions prises par chaque type d'agent influencent non seulement leur propre survie, mais aussi l'équilibre de l'ensemble de l'écosystème. Les résultats obtenus montrent que la stratégie adoptée par chaque agent joue un rôle crucial dans sa performance.

L'agent vorace, malgré sa capacité à récolter les ressources rapidement, se retrouve souvent vulnérable en raison de sa tendance à privilégier les ressources les plus abondantes, parfois au détriment de sa propre santé. L'agent local, en revanche, privilégie la proximité des ressources, ce qui lui permet d'éviter les déplacements risqués, mais le rend également moins compétitif en termes de récolte. L'agent rentable, quant à lui, adopte une stratégie plus calculée, cherchant à maximiser l'efficacité de ses déplacements tout en minimisant les pertes, mais sa prudence peut parfois l'amener à manquer des opportunités.

Cependant, la véritable question qui émerge de cette simulation est la suivante : jusqu'à quel point ces stratégies peuvent-elles être considérées comme "optimales" dans un monde incertain et dynamique ? Les agents, malgré leur intelligence artificielle et leurs stratégies bien définies, sont toujours confrontés à des éléments aléatoires, comme l'agencement des ressources et des obstacles, qui échappent à tout contrôle. Cela reflète une réalité fondamentale de notre monde : les choix que nous faisons, aussi rationnels soient-ils, sont souvent influencés par des facteurs imprévus et des contraintes que nous ne pouvons anticiper.

De plus, les résultats de cette simulation nous rappellent l'importance de la compétition dans les systèmes complexes. Comme dans la nature, où les ressources sont limitées, la survie des agents dans cette simulation dépend de leur capacité à s'adapter, à coopérer ou à rivaliser pour maximiser leurs chances de succès. En fin de compte, la vraie question n'est pas seulement qui survit, mais aussi comment les agents — ou, par extension, nous — évoluons face aux défis imposés par un environnement en constante évolution.

Cette simulation, bien que simplifiée, offre une réflexion sur la gestion des ressources, la prise de décision stratégique et la compétition dans des environnements complexes et dynamiques. Les enseignements tirés peuvent être appliqués à des domaines aussi variés que la gestion des ressources naturelles, la planification urbaine, ou même l'économie mondiale, où les stratégies d'optimisation, d'adaptation et de coopération jouent un rôle central. Finalement, ce travail nous pousse à nous interroger sur notre propre approche face aux ressources limitées et à la compétitivité dans un monde de plus en plus interconnecté et imprévisible.