# Compte-Rendu du Projet de Programmation Orientée Objet (POO)

Groupe numéro 20

## 1 Introduction

Dans le cadre du TP de Programmation Orientée Objet de 2$^{e}$ année à l'ENSIMAG, nous avons été chargés de développer une application Java simulant une équipe de robots pompiers opérant de manière autonome dans un environnement naturel comportant des incendies à éteindre. Ce projet a pour but de renforcer notre compréhension des concepts avancés de la programmation orientée objet.

## 2 Présentation de la Simulation

La simulation repose sur une carte composée de cases de différents types (*forêt, roche, eau, terrain libre, etc.*) où des incendies peuvent se déclarer. Chaque robot possède des caractéristiques spécifiques en termes de déplacement, capacité de réservoir, et vitesse d'extinction.

Les classes de base définies incluent :

- La carte, qui modélise les types de terrain et les positions des incendies.
- Les incendies, caractérisés par leur intensité et leur emplacement.
- Les robots (*pattes, roues, chenilles, drone*), abstraits pour partager des attributs et méthodes communs, mais avec des implémentations spécifiques (*dessin, intervention, accessibilité, temps de déplacement, etc.*).

## 3 Simulation

### 3.1 Affichage de la Carte

Nous avons suivi l'exemple donné par la classe `TestInvader` pour réussir à afficher les cartes. L'affichage a été conçu pour représenter visuellement les différents types de cases (*eau, forêt, rochers, terrain libre*) et pour positionner les robots et les incendies de manière claire.

Une partie importante du projet a été l'implémentation d'un simulateur à événements discrets pour gérer les interactions entre les robots et les incendies.

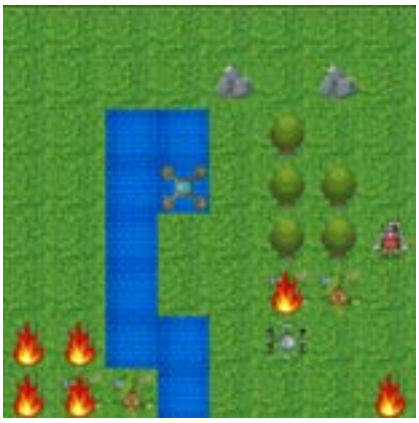

Figure 1: carteSujet

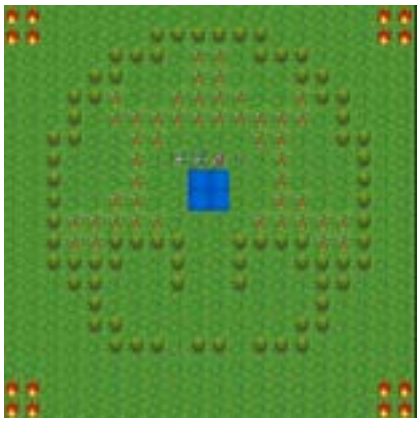

Figure 2: mushroom-of-hell

### 3.2 Stockage des données : la classe `DonneesSimulation`

La classe `DonneesSimulation` stocke les données de la carte. Nous avons choisi une liste chaînée pour gérer les robots et les incendies en raison de sa complexité O(1) pour l'ajout et la suppression d'éléments, contrairement à une `ArrayList` où ces opérations peuvent aller jusqu'à O(n).

Comme les accès par index sont rares dans notre application (seulement dans une configuration spécifique du `ChefPompier`), la complexité O(n) des listes chaînées pour cette opération est négligeable. Les parcours étant principalement effectués avec des boucles `for-each`, une liste chaînée offre un bon compromis pour nos besoins.

### 3.3 Gestion des événements

La classe `Simulateur` gère les événements grâce à une structure de type `Map<Integer, LinkedList<Evenement>>`. Chaque date (`Integer`) est associée à une liste chaînée (`LinkedList`) d'événements.

Cette structure permet un accès rapide par clé (O(1) et une gestion efficace des ajouts et suppressions dans les listes (O(1). Elle offre une organisation claire et performante pour modéliser le déroulement des opérations et synchroniser les actions des robots.

## 4 Calcul des Plus Courts Chemins

Après avoir implémenté notre simulateur, il était temps de mettre au point un programme qui calcule le plus court chemin entre deux cases pour un robot donné afin que celui-ci puisse se rendre le plus rapidemnt possible sur un incendie ou encore à côté d'une case d'eau. Nous avions besoin d'un algorithme

précis mais surtout rapide et c'est pourquoi nous avons choisi d'implémenter l'algorithme A* (un cas particulié de l'algorithme de Dikjstra).

### 4.1 Fonctionnement de l'algorithme

Il s'agit d'un algorithme de recherche du plus court chemin entre deux noeuds dans un graphe en prenant en compte un certain coût. A chaque noeud du graphe est attribué un coût total f(n) égal à la somme de la distance parcourue depuis le point de départ g(n) et de la distance estimée jusqu'au point d'arrivée h(n) (c'est l'heuristic). Tout au long de l'algorithme on travaille avec deux listes, une qui contient les points sur lesquels on est déjà passé (la liste fermée) et une contenant les points qu'il nous faut tester (la liste ouverte). A chaque itération de l'algorithme on va prendre le point de la liste ayant le cout f(n) le plus faible, vérifier si c'est le point destination, et si ce n'est pas le cas on ajoute ce point à la liste fermée, puis on ajoute les voisin accessibles à la liste ouverte en mettant à jour les valeurs de f(n), g(n) et h(n). On répète le processus jusqu'à avoir trouvé le point d'arrivée, ou jusqu'à ce que la liste ouverte soit vide. Dans ce deuxième cas cela signifie qu'il n'existe pas de chemin entre les deux points.

### 4.2 Notre implémentation

Afin d'implémenter cet algorithme nous avons utilisé une `PriorityQueue` pour la liste ouverte afin de trier les noeuds (dans notre cas les cases) par leur coût f(n). Pour la liste fermée nous avons utilisé une `ArrayList` car les ajouts se font en O(1). Cependant une autre opération que l'on effectue sur cette liste fermée est .contains, c'est pourquoi il aurait été plus judieux d'utiliser un `HashSet` car cette opération aurait été en O(1) contre O(n) dans notre cas. Ensuite nous itérons jusqu'à ce que la liste ouverte soit vide ou alors que l'on ait trouvé la case destination. Nous avons du définir une class Cell, contenant trois attribus pour les coûts f, g et h, et un pour la case courante et un dernier correspondant à la case de laquelle on vient. C'est ce dernier qui nous permet de reconstruire le chemin. En effet, l'algorithme A* nous renvoit une Cell, et c'est grâce à une autre fonction qui parcourt les Cell de parents en parents que l'on peut reconstruire le chemin de Cases. Enfin, nous avons décidé par soucis de simpliciter de transformer cette `LinkedList` de Case en une `LinkedList` de directions (Nord, Sud, Est, Ouest).

## 5 Stratégies d'Extinction des Incendies

Pour optimiser l'attribution des incendies, nous avons expérimenté plusieurs stratégies. En premier lieu, nous avons implémenté une stratégie de base où chaque incendie est attribué aléatoirement à un robot disponible. Nous avons ensuite développé une stratégie avancée, dans laquelle chaque robot calcule le temps nécessaire pour atteindre un incendie donné, et l'incendie est assigné au robot pouvant intervenir le plus rapidement. Enfin nous avons mis au point

une dernière stratégie encore plus efficace, visant à trier les robots par vitesse de déplacement croissante (grâce à une `PriorityQueue`) et d'asigner au robot le plus lent l'incendie le plus proche de lui et ainsi de suite pour tous les robots. Cette stratégie permet d'éviter aux robots les plus lents de perdre trop de temps à se déplacer étant donnée que ceux-ci ont un réservoir d'eau infinie. En travaillant ensemble sur ces stratégies, nous avons pu évaluer leur efficacité respective et améliorer la réactivité globale de la simulation.

# 6 Défis Rencontrés

Lors de ce projet, nous avons rencontré un certain nombre de défis, notamment en ce qui concerne la gestion des événements successifs. Lorsque nous avons voulu mettre en place la stratégie d'extinction des incendies, nous nous sommes rendu compte que nous ne pouvions gérer les événements qu'individuellement et non en série. Nous avons donc dû adapter nos événements afin de pouvoir les enchaîner. Cette modification nous a permis de réussir l'extinction des incendies.

Nous avons également essayé de développer des extensions. Cependant, lors de la création de l'extension qui gère la propagation des incendies, nous avons rencontré un problème lorsque nous ajoutions un nouvel incendie. En raison de ce dysfonctionnement, cette extension n'est malheureusement pas opérationnelle. De même pour une consommation d'energie trop importante nous n'avons pas implémenté le déplacement continu des robots.

# 7 Conclusion et Résultats

Ce projet a été une excellente opportunité pour chacun de nous d'approfondir nos compétences en programmation orientée objet et de découvrir des concepts de simulation avancés. Le travail en équipe nous a permis d'aborder des aspects variés du projet, d'échanger sur les meilleures pratiques, et d'optimiser notre application de manière collaborative.

Les tests finaux montrent que l'application gère efficacement les incendies dans différents scénarios, avec une bonne réactivité et une coordination optimisée des robots. La stratégie avancée a démontré une nette amélioration en termes de rapidité d'extinction par rapport à la stratégie de base, validant ainsi l'intérêt de notre approche collective pour l'affectation des robots.