Rapport projet tpl

Pour des informations de compilation et exécution veuillez lire le fichier README dans le repertoire du rendu.

Introduction:

L'objectif principal de ce projet est de simuler l'ensemble des opérations qu'une équipe de robot de différentes caractéristiques fait pour éteindre des incendies créés avec des intensités différentes, sur des terrains différents. En effet, on distribue les incendies aux robots selon la stratégie sélectionnée, et puis on effectue des calculs de chemin et on programme des allers-retours pour remplir le réservoir et éteindre définitivement l'incendie.

En plus des données de l'énoncé : les classes et leurs attributs, la nouvelle chose qu'on doit mentionner est la classe abstraite chefPompier qui sera détaillée dans la partie des stratégies au-dessous.

Gestion des évenements:

A la question de la gestion des évenements, on a utilisé une collection permettant l'ordre et l'unicité *SortedMap<Long, LinkedList<Evenement>>* pour stocker tous les événements générer au cours de l'exécution. Donc en cliquant sur Next ou sur lecture (qui fait des nexts successives), la liste des événements de date k sera exécuté mais si la liste est vide on fera aucune mise à jour et d'ailleurs c'est la manière qui permet de représenter le temps d'attente.

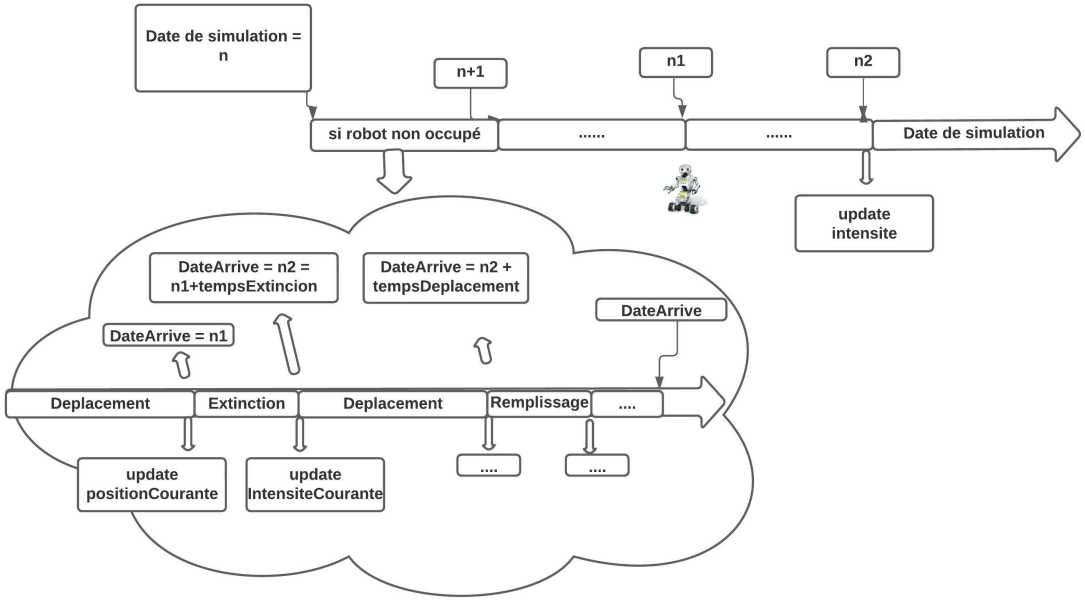
Les mécanismes de la gestion du temps est la partie la plus importante de notre conception, vu qu'on a essayé de conserver les données différentes de chaque robot (temps d'extinction, temps de déplacement en fonction de la vitesse, temps de la charge du réservoir). Les décalages entre ces temps étaient énormes, c'est la raison pour laquelle on a décidé de créer une échelle pour chaque événement tout en conservant le décalage visuel (le nombre d'incrémentation de la date courante est énorme, c'est tout à fait logique vu que par exemple drone remplit son réservoir en 30 $min = 60 \times 30 = 3600s$) Donc on calcule un temps d'attente pour chaque événement qui sera converti en une date qui representera une clé dans la structure des évenements.

Notre idée de la conception peut paraître perturbante, mais elle devient très intuitive si on imagine deux files chronologiques d'action : la prise de décision et l'exécution réelle. Les variables courantes (reservoirCourant, intensiteCourante, positionCourante) changent à chaque insertion des événements au moment de la prise de décision et les autres (réservoir, intensité, position) changent pendant leurs dates effectives dans la liste des événements.

Au moment du démarrage du programme, et après l'affectation des incendies on est toujours dans la date de simulation 1, or il faut ajouter les déplacements élémentaires de la position actuelle du robot vers sa destination. Si un déplacement élémentaire coutera 4 dates par exemple, donc la position réelle du robot (référée dans le code par position) ne changera qu'à la date courante de l'exécution 4. Enfin, si on veut ajouter d'autres événements avant la date 4, toujours pendant la date courante 1, on utilisera la position virtuelle du robot référée dans le code par positionCourante . La file de la prise de décision est toujours en avance de la file de l'exécution car elle nous permet d'insérer plusieurs événements à la fois (déplacement + éteindre incendie et l'aller-retour pour remplir le réservoir) dans une même date courante.

L'attribut clé dans cette conception est datearrivee dans la classe robot. Il sert à détecter la date dans la liste des événements à partir de laquelle le robot est libre. C'est une date qui change au cours

de chaque prise de décision d'un événement en fonction du temps d'attente calculé.



Plus court chemin:

Il y'a plusieurs algorithmes, Breadth First Algorithm BFS, Dijkstra's Algorithm et A* Algorithm. L'idée générale de ces algorithmes est que nous gardons une idée sur la frontière en expansion. L'algorithme BFS explore toutes les directions, L'algorithme de Dijkstra permet de donner une certaine priorité à des pistes. Au lieu d'explorer toutes les voies possibles de la même manière, il privilégie les chemins à moindre coût mais, il fait perdre du temps en explorant des directions qui ne donnent pas le plus court chemin. L'algorithme A* est une version optimisée de l'algorithme de Dijkstra. A* trouve des chemins vers un emplacement, ou le plus proche de plusieurs. Et bien sûr, donne la priorité aux chemins qui semblent se rapprocher de la destination.

Algorithme choisi : A*
Mais pour que l'algorithme A* soit le plus optimal possible, il lui faut une fonction de coût bien choisie, qu'on appelle f, On a $f = g + h$. g : $g(current)$ c'est le nombre de pas fait pour aller de la case source vers la case current. h : $h(current)$ c'est le coût heuristique estimé de current vers la destination. Donc pour bien choisir f, il faut bien choisir h, (car g reste la même).

Choix de la fonction heuristique h:
Vu que les robots n'ont qu'un mouvement dans 4 directions, on ne peut pas utiliser la distance euclidienne. Donc la distance la plus adaptée aux cartes de 4 directions est la distance de Manhattan. C'est la somme de deux différences des abscisses et des ordonnées de la case et la destination en valeur absolue.

$$|(source.x - destination.x)| + |(source.y - destination.y)|$$

Il faut aussi prendre en compte le temps nécessaire pour arriver à la destination. du coup la fonction heuristique qu'on va utiliser est de la forme :

$$h(case) = manhattanDistance(case, destination) \times \frac{(tailleCase)}{vitesse}$$

vitesse : vitesse du robot dans cette case.
tailleCase : taille de la case en mètre.

Mais, il faut privilégier les chemins qui sont (quasiment) sur la ligne droite de la case source à la destination. Pour ce faire, on calcule le produit vectoriel entre le vecteur (source → destination) et (case Courante → destination). Lorsque ces vecteurs ne sont pas (quasiment) colinéaires, ce produit vectoriel sera plus grand. donc on va privilégier les chemins qui sont sur la ligne du premier vecteur (source → destination) en augmentant(légèrement) les coûts des autres chemins. Et cela aide l'algotithme A* à explorer moins de chemins.

Donc la fonction heuristique finale est :

$$\boxed{h(case) = manhattanDistance(case, destination) * ((tailleCase)/vitesse) + produitVectoriel * 0.001}$$

produitVectoriel = (source → destination) ∧ (case Courante → destination)

NB : on n'utilise pas A* pour les drones, car il n'y a pas d'obstacles pour eux. on utilise un algorithme très simple. On essaye d'augmenter (ou diminuer) les abscisses (colonnes) et les ordonnées (ligne) de la position du drone jusqu'à l'arrivée du drone vers la destination.

Stratégies:

Stratégie élémentaire : C'est la stratégie adoptée par le chef pompier ChefPompierSimple. Ce dernier choisit (aléatoirement) une incendie et l'affecte à un robot(disponible) aléatoirement, même s'il est très loin. Le robot reçoit les ordres du chef pompier, calcul le plus court chemin pour arriver à l'incendie qui lui est affectée. Chaque robot(sauf le robot à pattes) a un reservoir fini, donc quand le reservoir est vide, le robot doit nécessairement le remplir dans la case la plus proche qui contient l'eau à l'aide d'une méthode closestWaterDestination() qui retourne la case de nature EAU la plus proche du robot. Il doit aprés continuer son intervention, ou bien se dériger vers une nouvelle incendie désignée par le chef ChefPompierSimple, ou bien tout simplement être prêt à intervenir après.

Stratégie évoluée : C'est la stratégie adoptée par le chef pompier ChefPompierEvolue, ce dernier affecte une incendie au robot le plus proche((disponible bien sûr), et ceci est à l'aide d'une méthode closestRobot(Incendie incendie). Et comme dans la première stratégie,le robot trouve le plus court chemin pour aller à l'incendie. Si le réservoir est vide, le robot cherche la case la plus proche de nature EAU et le remplit pour continuer d'éteindre les incendies.