TP 8 : Eviter les obstacles

Dominique Blouin, Télécom Paris, Institut Polytechnique de paris
dominique.blouin@telecom-paris.fr

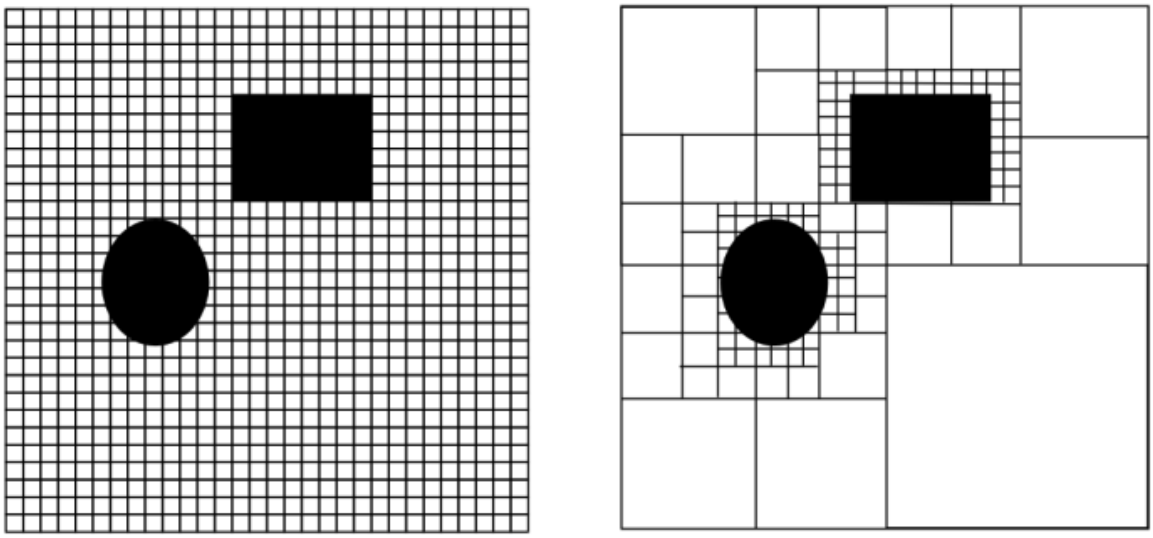
Au cours du TP 6, vous avez mis en œuvre le MVC afin de visualiser dans l'interface graphique tout changement des données du modèle. Puis vous avez défini un comportement très simple des robots de l'usine, consistant à visiter une liste de composants de l'usine. Ces déplacements ne prenaient pas en compte les différents obstacles dans l'usine tels que les murs des salles par exemple.

Au cours de ce TP, vous allez raffiner ce déplacement des robots afin de contourner les obstacles dans l'usine. A cette fin, vous utiliserez une bibliothèque de calcul de trajectoire qui vous sera fournie.

Le calcul de trajectoire

Il existe plusieurs méthodes de calcul de trajectoires pour la robotique, tout comme pour les jeux vidéo. L'approche que vous allez mettre en œuvre pour ce projet s'inspire de ce qui est décrit ici.

La première étape consiste à créer une cartographie de l'espace dans lequel évoluent les robots. Cette cartographie déterminera l'occupation des différents objets contenus dans l'espace représenté. Une approche typique consiste à **discrétiser** cet espace tel qu'illustré par la figure suivante.

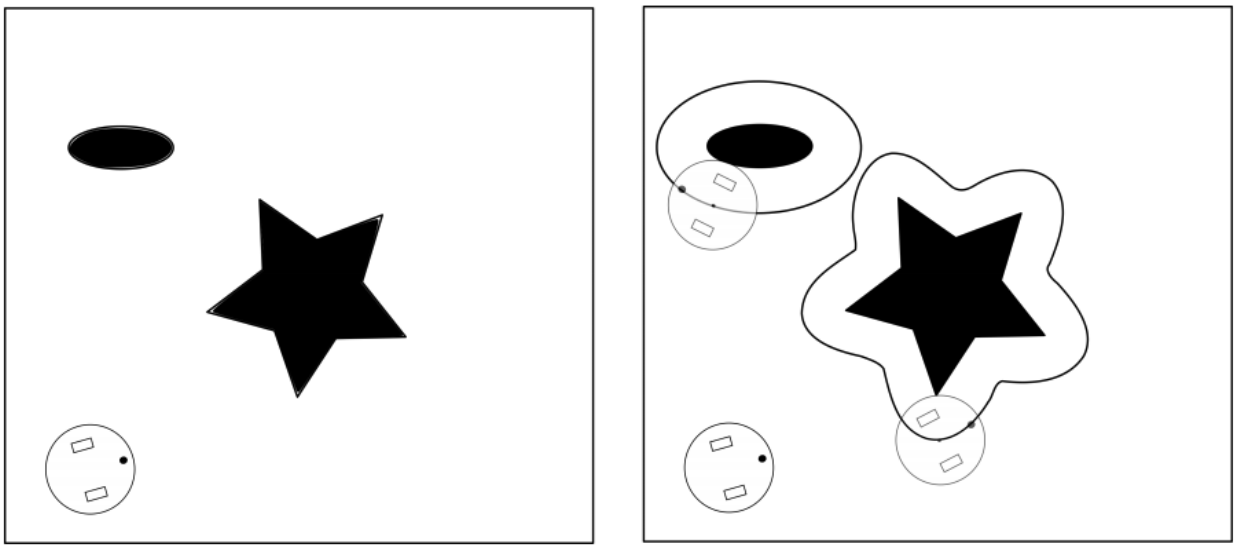


La partie de gauche de la figure représente une discrétisation de l'espace en carrés, dont la taille dépendra de la précision requise pour la simulation, ainsi que des ressources mémoire et de traitement allouées à cette simulation. En effet, une discrétisation très fine permettra de déterminer des trajectoires très précises, mais demandera plus de mémoire et de temps de calcul. Ceci-dit, en cas de problèmes, il est possible d'optimiser la quantité de mémoire

requise pour stocker la cartographie en la représentant sous forme d'arbre kd de manière que seules les surfaces contenant des frontières d'obstacles seront cartographiées, tel qu'illustré par le côté droit figure précédente.

Lorsque l'espace aura été discrétisé, il pourra être vu comme un **graphe**. Ainsi, chaque case sera vue comme un **sommet** du graphe, et pour chaque case, on ajoutera une arête entre le sommet de cette case et le sommet de chacune des cases qui lui sont adjacentes. Cependant, on ne va pas nécessairement prendre en compte toutes les cases adjacentes. En effet, si une case adjacente coïncide avec un obstacle, on n'ajoutera pas d'arête entre le sommet de la case et la case de cet obstacle, ce qui permettra de rendre les sommets des obstacles **non accessibles**.

Il faudra donc être capable de **détecter les obstacles**, afin de savoir si une case adjacente doit être reliée par une arête ou non. Là encore, plusieurs approches sont possibles pour détecter les obstacles. Une approche fréquemment utilisée consiste à considérer que l'objet se déplaçant est un point, et d'augmenter la taille des objets à contourner de la moitié de la dimension maximale de l'objet se déplaçant, tel qu'illustré par la figure suivante.



Cela nécessite cependant des calculs pouvant être assez compliqués, dépendamment de la forme des objets. Il existe toutefois différentes bibliothèques permettant de modéliser des formes géométriques telles que Apache Commons Geometry, mais leur utilisation demeure compliquée.

Une approche très souvent utilisée dans les jeux vidéo consiste à approximer toute forme par une forme de taille similaire à la forme réelle, mais dont les contours sont plus simple tel qu'un cercle ou un rectangle par exemple. C'est cette dernière approximation que vous allez utiliser dans ce TP. Une explication de cette méthode se trouve ici.

Lorsque l'espace sera vu comme un graphe, on pourra utiliser des algorithmes très connus comme celui de Dijkstra, qui permet de trouver le plus court chemin dans un graphe. Cette recherche de chemin, illustrée par une animation ici est celle que vous utiliserez dans ce TP.

Tout comme pour l'interface graphique de visualisation, vous utiliserez une bibliothèque existante à intégrer à votre projet pour calculer le plus court chemin.

Il existe plusieurs autres algorithmes de calcul de trajectoire tel que *A**, qui ajoute des heuristiques à l'algorithme de Dijkstra afin de ne pas avoir à effectuer une recherche complète sur tous les sommets du graphe. Plus récemment, des algorithmes mettant en œuvre de l'apprentissage automatique (machine learning) ont été développées pour calculer des trajectoires de robots.

Créer une interface de calcul de trajectoire

Tel que vu précédemment, le calcul de trajectoire est un sujet relativement complexe, et il existe plusieurs algorithmes ayant des propriétés différentes. Ainsi il sera préférable d'**encapsuler** le calcul de trajectoire dans une classe dédiée, tel que préconisé par le principe de responsabilité unique en OO. Par ailleurs, puisqu'il existe plusieurs algorithmes, il sera préférable que votre modèle d'usine ne connaisse qu'une interface de calcul de trajectoire, découplant ainsi votre modèle d'usine des dépendances techniques liées à des algorithmes précis, ce qui permettra de facilement changer l'algorithme sans modifier les classes du modèle.

A cette fin créer une interface nommée *FactoryPathFinder*. Dans cette interface définir une signature de méthode nommée *findPath*. Cette méthode définira deux paramètres, le premier étant un composant source et le deuxième un composant cible, ces deux paramètres servant de sommet de départ et d'arrivée du chemin le plus court à calculer.

La méthode *findPath* retournera une liste d'objets de type *Position*, une classe simple qui encapsule les coordonnées x et y dans un plan. Si cette classe *Position* n'existe pas encore dans votre modèle, vous pouvez la créer et modifier votre modèle afin qu'il stocke les coordonnées des composants sous forme d'instances de cette classe *Position*.

Modifier ensuite votre classe *Robot* afin de pouvoir lui fournir un objet de type *FactoryPathFinder*. Modifier le comportement de votre classe *Robot* (méthode *behave*) pour utiliser l'objet *FactoryPathFinder*. A chaque fois que le composant à visiter change, calculer une nouvelle trajectoire pour cette nouvelle cible, et à chaque cycle de la simulation, déplacer le robot à la position suivante de la trajectoire calculée précédemment, et ce jusqu'à ce que la cible soit atteinte. Puis mettre à jour la nouvelle cible et ainsi de suite.

Implémenter l'interface de calcul de trajectoire

Il faut maintenant développer une classe implémentant l'interface *FactoryPathFinder*. Cette classe mettra en œuvre l'algorithme de Dijkstra en appelant une bibliothèque de modélisation de graphes existante.

Deux bibliothèques de modélisation de graphes

Choisir la bibliothèque que vous utiliserez parmi les deux bibliothèques décrites dans les sous-sections suivantes.

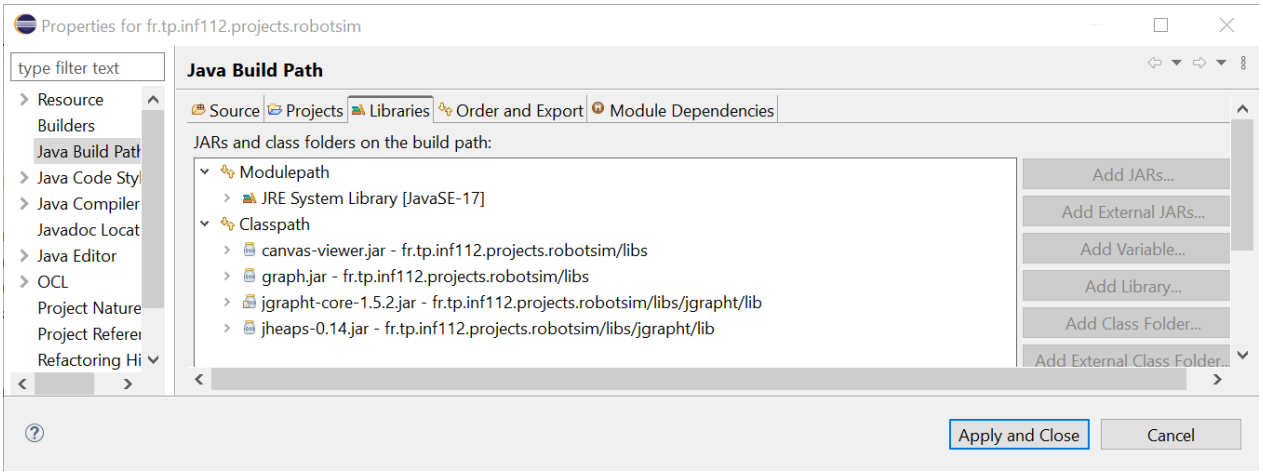
La bibliothèque JGraphT

JGraphT est une bibliothèque très riche et très utilisée pour différents traitements sur des graphes. Elle est également très bien documentée et open source. Elle fournit plusieurs

classes pour modéliser différents types de graphes, dirigés ou non par exemple. Si vous choisissez cette bibliothèque, vous utiliserez la classe *DefaultDirectedGraph*. Cette classe est **générique** (tout comme l'interface *List* et ses classes telle que *ArrayList*), et peut donc être utilisée avec les classes de votre choix pour modéliser les sommets et les arêtes. Vous devrez utiliser votre propre classe pour représenter les sommets, et utiliser la classe *DefaultEdge*, fournie par JGraphT pour représenter les arêtes. Vous utiliserez la classe *DijkstraShortestPath* et sa méthode *findPathBetween* pour calculer le chemin le plus court entre deux sommets.

Configuration du projet du simulateur dans Eclipse

La bibliothèque JGraphT se télécharge ici (menu DOWNLOAD). Ajouter cette bibliothèque à votre projet dans Eclipse tel que décrit dans le TP 5 pour la bibliothèque *canvas-viewer.jar*. Configurer votre projet pour pouvoir utiliser les deux fichiers *jgrapht-core-<version>.jar* et *jheaps-<version>.jar* de JGraphT tel qu'illustré par la capture d'écran suivante.



La documentation de JGraphT se trouve ici, avec une section *Hello JGraphT* pour vous aider à démarrer, si vous choisissez d'utiliser cette bibliothèque.

La bibliothèque « fait-maison » Graph

Cette bibliothèque, plus simple mais plus limitée que JGraphT fourni différentes interfaces modélisant un graphe (*Graph*), ses sommets (*Vertex*) et ses arcs (*Edge*), ainsi que des implémentations de base de ces interfaces. Pour cet exercice, vous utiliserez les classes *GridGraph*, *GridVertex* et *GridEdge*, qui implémentent les interfaces précédentes et facilitent la conversion d'une grille telle que le plan de l'usine robotisée en graphe.

Cette bibliothèque contient également une classe nommée *DijkstraAlgorithm*, qui fournit une méthode statique nommée *findShortestPath*. Cette méthode prend en paramètre un objet de type *Graph* et deux objets de type *Vertex*, représentant le sommet de départ et le sommet d'arrivée de ce chemin. Elle retourne une liste de sommets adjacents, débutant par le sommet de départ et se terminant par le sommet d'arrivée, et définissant le chemin le plus court entre ces sommets.

Configuration du projet du simulateur dans Eclipse

La bibliothèque « fait-maison » *Graph* se télécharge ici. Ajouter cette bibliothèque à votre projet dans Eclipse tel que décrit précédemment pour JGraphT. Configurer votre projet pour utiliser le fichier *graph.jar* tel qu'illustré par la capture d'écran précédente.

Approche d'utilisation de la bibliothèque de graphes

Pour utiliser l'interface graphique *Canvas Viewer*, afin de visualiser votre modèle, vous avez fait implémenter les interfaces de programmation *Canvas* et *Figure* par votre modèle. Ces interfaces définissent un **point de vue** sur votre modèle. Dans le cas de l'algorithme de Dijkstra, vous utiliserez une autre approche, celle de la **transformation de modèles**. Cette approche se définie comme étant la génération automatique d'un ou de plusieurs **modèles cibles** à partir d'un ou de plusieurs **modèles source**, et est essentielle à l'ingénierie dirigée par les modèles, afin de pouvoir utiliser différents outils d'analyse des modèles, dont les formats d'entrée ne sont pas les mêmes que celui du modèle source.

La raison de ce choix, par rapport à celui de point de vue utilisé pour visualiser votre modèle d'usine robotisée, est que pour ce simulateur, on souhaite évaluer différents types d'algorithmes de calcul de trajectoire, ce qui pourra être réalisé simplement en utilisant différentes classes d'implémentations de l'interface *FactoryPathFinder*. Cette dernière permet de minimiser le **couplage** entre les classes du modèle de l'usine robotisée et les classes de recherche de chemin.

Cela n'est pas le cas pour la visualisation de l'usine robotisée, qui n'utilise qu'un seul type de visualisation. En faisant directement implémenter interfaces de canevas par le modèle d'usine robotisée, ce dernier est par conséquent **fortement couplé** à la bibliothèque Canvas Viewer. En revanche, l'utilisation de l'approche point de vue permet de meilleures performances que la transformation de modèles, car il n'est pas nécessaire de reconvertir tous les objets de l'usine en objets de canevas à chaque fois que les données de cette dernière ont changées, ce qui rendrait la visualisation bien moins fluide.

Transformer un modèle d'usine robotisée en graphe

Pour calculer le plus court chemin entre un robot et sa cible, il faudra développer du code Java pour transformer un modèle d'usine robotisée en modèle de graphe. Il faudra d'abord instancier un objet de la classe représentant un graphe de la bibliothèque de graphes que vous avez choisie (voir sections précédentes). Puis, il faudra instancier des objets sommets et arêtes du graphe pour représenter les cases du plan de l'usine.

La classe représentant les sommets devra pouvoir mémoriser la position de la case correspondante du plan de l'usine, afin de pouvoir convertir le chemin calculé en une liste de positions à visiter par les robots, tel que définie par la méthode *findPath* de l'interface *FactoryPathFinder*.

Si vous utilisez JGraphT, vous pouvez utiliser n'importe quelle classe de votre choix pour représenter les sommets, et utiliser la classe *DefaultEdge* de JGraphT pour représenter les arêtes. Si vous utilisez la bibliothèque fait maison Graph, la classe des sommets devra être *GridVertex* ou une sous-classe de cette dernière, et les classes des sommets et arêtes seront respectivement *GridVertex* et *GridEdge*.

Pour chaque case de l'usine, il faudra construire un sommet en instanciant la classe de sommet choisie, qui mémorisera la position de la case associée. Puis, lorsque tous les sommets du graphe auront été créés, il faudra construire les arêtes entre ces sommets. En première approximation, considérer que les robots ne peuvent se déplacer que dans les directions x (largeur) et y (hauteur). Ainsi, pour une case donnée, seules les quatre cases adjacentes à gauche, à droite et en haut et en bas seront reliées à la case centrale par une arête. Pour l'instant, considérer qu'il n'y a pas d'obstacles dans l'usine.

Calculer le plus court chemin

Dans la méthode *findPath* de votre classe implémentant l'interface *FactoryPathFinder*, appeler la méthode de recherche de plus court chemin en utilisant les bonnes classes de la bibliothèque choisie. Il faudra ensuite convertir la liste de sommets retournée en **liste de positions** dans l'usine, que la méthode *findPath* doit retourner tel que requis par l'interface *FactoryPathFinder*.

Tester le bon fonctionnement du calcul de trajectoire, sans obstacles

Le développement de cette partie du simulateur étant plus complexe que les développements des TP précédents, utiliser une approche de développement **incrémentale**. Ainsi, tester d'abord votre algorithme de calcul de trajectoire sans prendre en compte les obstacles de l'usine.

Pour ce faire, instancier une usine contenant un robot, deux machines, chacune d'entre elles étant située dans une aire contenue dans sa propre salle. Chaque salle sera dotée d'une porte. Ajouter une station de recharge dans sa propre salle. Ajouter les deux machines et la station de recharge à la liste des composants à visiter par le robot. Lancer l'interface graphique et vérifier que votre robot se déplace correctement en visitant successivement les deux machines et la station de recharge, sans prendre en compte les obstacles.

Détecter les obstacles

Afin de contourner les obstacles, il ne faudra pas ajouter une arrête si la case adjacente est localisée sur un objet ne devant pas être **traversé** par le robot, tel qu'illustré par cette animation.

Une bonne façon de procéder consiste à déléguer la décision de savoir si une case contient un objet bloquant à l'objet usine, car celui-ci connaît tous les composants. Vous pouvez par exemple ajouter une méthode *overlays* à votre classe *Component*, qui prendra en paramètre un objet de la classe représentant les sommets du graphe et déterminera si ces deux objets se chevauchent. A cette fin, vous pouvez approximer la forme des composants de l'usine en utilisant une forme simple de taille similaire à la forme réelle tel que décrit précédemment.

L'usine, à qui l'on demandera si un sommet chevauche l'un de ses composants, parcourra tous les composants pour appeler la méthode *overlays* et déterminer si un des composants constitue un obstacle, auquel cas on ne construira pas d'arc vers ce sommet.

La méthode *overlays* pourra être redéfinie dans les sous-classes de composants pour prendre en compte leur spécificité. A cette fin considérer qu'une salle est composée de murs, d'une

certaine épaisseur, et de portes comme sous-composants. Un mur constitue un obstacle et ne peut pas être chevauché. Une porte ne constitue un obstacle que si elle est **fermée**.

Considérer que les aires n'ont pas de murs et peuvent donc être traversées. Faire également l'hypothèse simplificatrice que les machines, convoyeurs et stations de recharge peuvent être traversées par les robots.

Ne pas oublier de gérer le cas où aucun chemin n'existe entre un robot et sa cible. Consulter la documentation de la bibliothèque de graphe pour connaître ce qui sera retournée par l'algorithme pour ce cas. Vous pouvez indiquer cet état de **blocage** du robot visuellement dans l'interface graphique en changeant le style de la figure retournée par le robot ou son libellé par exemple.

Gérer également le cas d'exception où un autre robot serait localisé dans la prochaine case où un robot doit se déplacer. Dans ce cas, le robot ne doit pas se déplacer, et ne continuera sa trajectoire que lors du prochain appel de la méthode *behave* si la case a été libérée.

Tester le bon fonctionnement de la détection des obstacles

Relancer le simulateur avec le modèle décrit précédemment et vérifier que votre robot contourne bien les obstacles en entrant dans les salles par leurs portes si elles sont ouvertes. Tester également avec plusieurs robots et vérifier que ceux-ci n'entrent pas en collision.

Pour aller plus loin

Les deux derniers exercices suivants sont optionnels et ne seront pas évalués.

Autoriser des déplacements en diagonale des robots

Modifier votre code pour que les robots puissent se déplacer en diagonale, en plus des orientations le long des axes x et y. Que constatez-vous ? Les robots se déplacent-ils toujours en prenant le chemin le plus court ? Proposer une solution à ce problème.

Gérer l'énergie des robots

Définir un niveau d'énergie pour les robots. Pour un robot, diminuer cette quantité d'énergie d'une valeur fixe à chaque déplacement. Lorsque la quantité d'énergie est basse, le robot calculera une trajectoire vers chaque station de recharge et décidera de continuer son travail si la quantité d'énergie restante est suffisante. Sinon, il se dirigera vers la station de recharge la plus proche. Le comportement de cette station de recharge, lorsqu'elle détectera qu'un robot est présent, sera de recharger le robot jusqu'à ce que la batterie soit pleine, auquel cas le robot pourra poursuivre son travail.

Développer une porte automatisée

Implémenter un comportement pour les portes afin que celles-ci s'ouvrent automatiquement lors qu'un robot arrive et se referment après son passage.