Étudiants

Informatique générale

I.U.T.A LYON 1

Alex Mehdi ZAHID zahid.alexmehdi@gmail.com

Antoine GRÉA grea09@gmail.com

Blon THO tho_blon@hotmail.com

Dossier de concurrence

Optimisation de trajet pour un robot

DiRIGe

Alex Mehdi ZAHID zahid.alexmehdi@gmail.com
Antoine GRÉA grea09@gmail.com
Blon THO tho_blon@hotmail.com

Sommaire

Alex Mehdi ZAHID zahid.alexmehdi@gmail.com
Antoine GRÉA grea09@gmail.com
Blon THO tho_blon@hotmail.com

1) Objectif

Ce dossier développe l'étude des logiciels concurrents, présente les différentes technologies, langages de programmations, algorithme et environnements potentiellement utilisables pour réaliser le projet.

Il n'existe pas à proprement parler de logiciel sur la marché ayant les mêmes fonctionnalités que l'application de notre projet. Cependant, de nombreux logiciels présentent le même objectif : trouver le chemin le plus court entre deux points. Ces logiciels couvrent souvent le domaine des transports (routiers, marins, aériens). Pour atteindre cet objectif, il existe plusieurs algorithmes.

Le fonctionnement de chaque algorithme sera décrit par des pseudo-codes.

Alex Mehdi ZAHID **Antoine GRÉA** **Blon THO**
zahid.alexmehdi@gmail.com grea09@gmail.com tho_blon@hotmail.com

2) Les algorithmes

2.1) Algorithme de Bellman-Ford

L'algorithme de Bellman-Ford permet de trouver des plus courts chemins depuis un sommet source donné. Il autorise la présence d'arcs de poids négatif, et détecte l'existence d'un circuit absorbant(1) dans un graphe orienté pondéré(2), accessible depuis le sommet source.

(1)Chemin formée par un ou plusieurs sommets qui a un poids total négatif.

(2)Graphe qui contient des arcs ayant un sens et un poids.

2.1.1) Pseudo-code :

```
procédure Bellman-Ford (G : graphe, départ : sommet de départ)

initialisation (G, s) ;  // les poids de tous les sommets sont fixés à
              //+infini
              // le poids du sommet initial à 0

pour i allant de 1 jusqu'à (Nombre de sommets – 1) faire
|     pour chaque arc (u, v) du graphe faire
|     |     p = poids(u) + poids(arc(u, v));
|     |     si p < poids(v) alors
|     |     |    pred(v) = u;
|     |     |    poids(v) = p;
|     |     finsi
|     finpour
finpour
pour chaque arc (u, v) du graphe faire
|     si poids(u) + poids(arc(u, v)) < poids(v) alors
|     |     retourner faux;
|     finsi
finpour
retourner vrai;

fin
```

2.1.2) Complexité :

Ici, la complexité est O(N*M) au pire des cas avec N le nombre de sommets du graphe et M le nombre d'arcs dans le graphe.

Alex Mehdi ZAHID **Antoine GRÉA** **Blon THO**
zahid.alexmehdi@gmail.com grea09@gmail.com tho_blon@hotmail.com

2.2) Algorithme de Floyd-Warshall

L'algorithme de Floyd-Warshall détermine tous les plus courts chemins dans un graphe orienté pondéré. Cet algorithme prend en entrée le graphe sous la forme d'une matrice d'adjacence(3) qui donne le poids d'un arc s'il existe ou l'infini sinon. Le poids d'un chemin entre deux sommets est la somme des poids de chaque arc qui constitue ce chemin. L'algorithme peut gérer les poids négatifs mais il ne doit pas exister de chemin absorbant. L'algorithme trouve les chemins les plus courts entre chaque paire de sommet en calculant le poids minimal parmi tous les chemins.

(3)Matrice ayant les mêmes dimensions en x et en y.

2.2.1) Pseudo-code :

```
1 procédure Floyd-Warshall (G : matrice x * y)
2
3 W = G;
4 pour k allant de 1 jusqu'à (Dimension x de W) faire
5 |    pour i allant de 1 jusqu'à (Dimension x W) faire
6 |    |    pour j allant de 1 jusqu'à (Dimension y W) faire
7 |    |    |    W[i][j] = min (W[i][j], W[i][k] + W[k][j]);
8 |    |    finpour
9 |    finpour
10finpour
11
12fin
```

2.2.2) Complexité :

Ici, la complexité est O(N3) au pire des cas avec N le nombre de sommets du graphe.

Alex Mehdi ZAHID **Antoine GRÉA** **Blon THO**
zahid.alexmehdi@gmail.com grea09@gmail.com tho_blon@hotmail.com

2.3) Algorithme de Dijkstra

Dans la théorie des graphes, cet algorithme permet de résoudre le problème du plus court chemin. Il s'applique à un graphe dont les arcs vont dans les deux sens et dont le poids lié à chaque arêtes est positif ou nul. L'algorithme de Dijkstra construit petit à petit un sous-graphe à partir des données initiales, dans lequel les différents sommets sont classés par ordre croissant de leur distance totale minimale au sommet de départ.

2.3.1) Pseudo-code :

```
13procédure Dijkstra (G : graphe, départ : sommet de départ)
14
15Initialisation(G, sdeb);         // les poids de tous les sommets sont fixés
à
16                     //+infini
17                     // le poids du sommet initial à 0
18Q = ensemble de tous les nœuds;
19tant que Q n'est pas un ensemble vide faire
20|    s1 = min(Q);          // s1 devient le nœud du plus petit poids
21|    Q = Q – s1;
22|    pour chaque nœud s2 voisin de s1 faire
23|    |    si d[s2] > d[s1] + poids(s1,s2) alors
24|    |    |    d[s2] = d[s1] + poids(s1,s2);
25|    |    finsi
26|    finpour
27finpour
28
29fin
```

2.3.2) Complexité :

L'algorithme de Dijkstra a pour complexité $O(N^2)$ dans le pire des cas avec N le nombre de sommet.

Alex Mehdi ZAHID **Antoine GRÉA** **Blon THO**
zahid.alexmehdi@gmail.com grea09@gmail.com tho_blon@hotmail.com

*2.4) Algorithme d'A **

L'algorithme d'A * est un algorithme de recherche du plus court chemin dans un graphe. Il ne donne pas toujours la meilleure solution mais est un algorithme assez rapide. L'algorithme reçoit la direction dans laquelle le point d'arrivée se trouve. Ensuite, il se rapprochera de plus en plus à la destination à chaque itération. Les possibilités plus proches de la destination seront privilégiées, les autres mises de côtés. Ainsi, si la liste des possibilités privilégiées amenait à une impasse, les possibilités mises de côtés seront retraitées. Cela garantira que l'algorithme trouvera toujours une solution (si elle existe).

2.4.1) Pseudo-code :

```
/*
Soit x un point.
g_score[x] distance parcourue pour arrivé au point étudié x
h_score[x] stocke le poids du chemin estimé pour arrivé au point x
f_score[début] somme des distances g_score[x] et h_score[x]
pred[] tableau qui contient les prédécesseurs. Exemple pred[x] contient le prédécesseur du point x
nœud_le_plus_intéressant : booléen permettant d'annoncer que l'on étudie le point potentiellement le plus intéressant parmi la liste ouverte, pour se déplacer à la prochaine étape.
cout_entre(x,y) : fonction qui détermine le cout pour se déplacer du point x au point y
*/

fonction A* (début : sommet de départ, arrivée : sommet d'arrivée, nb_nœud : nombre de nœuds)

liste_fermée = vide //contient l'ensemble des nœuds déjà évalués
list_ouverte = début //contient l'ensemble des nœuds à évaluer

pour i allant de 0 à nb_nœud faire
|    g_score[i] = 0;
|    h_score[i] = 0;
|    f_score[i] = 0;
|    pred[i] = vide ;
finpour

g_score[début] = 0;
h_score[début] = distance_estimée(début, arrivée);

//La fonction retourne la distance à vol d'oiseau entre les points passés en paramètre.
//Elle pourra également faire intervenir des coûts supplémentaires exemple : rotation

f_score[début] := h_score[début]

tant que liste_ouverte n'est pas vide faire
|    x := nœud de la liste ouverte qui a le score f_score[] le plus petit;
|    si x = arrivée
```

```
63|    |    retourner reconstruire_chemin(pred,arrivée);
64|    finsi
65|    supprimer x de la liste_fermée;
66|    ajouter x à la liste_ouverte;
67|    pour chaque y nœud_voisin(x) faire
68|    |    si y est dans la liste_fermée
69|    |    |    tentative_g_score = g_score[x] + cout_entre(x,y);
70|    |    finsi
71|    |    si y n'est pas dans la liste ouverte
72|    |    |    ajouter y à la liste_ouverte;
73|    |    |    nœud_le_plus_intéressant = vrai;
74|    |    sinon
75|    |    |    si tentative_g_score < g_score[y]
76|    |    |    |    nœud_le_plus_intéressant = vrai;
77|    |    |    sinon
78|    |    |    |    nœud_le_plus_intéressant = faux;
79|    |    |    finsi
80|    |    finsi
81|    |    si nœud_le_plus_intéressant = vrai
82|    |    |    pred[y] = x;
83|    |    |    g_score[y] = tentative_g_score;
84|    |    |    h_score[y] = distance_estimée(y, arrivée);
85|    |    |    f_score[y] = g_score[y] + h_score[y];
86|    |    finsi
87|    finpour
88fintantque
89retourner échec;
90
91fin
92
93fonction reconstruire_chemin (pred, nœud_courant)
94
95si pred[nœud_courant] <> vide
96|    p = reconstruire_chemin(pred,pred[nœud_courant]);
97|    retourner( p + nœud_courant );
98sinon
99|    retourner chemin_vide;
100   finsi
101
102
103   fin
```

2.4.2) Complexité :

La complexité de l'algorithme d'A * est de O(N + M * log(N)) avec N le nombre de sommets et M et nombre d'arcs présents dans le graphe.

Alex Mehdi ZAHID **Antoine GRÉA** **Blon THO**
zahid.alexmehdi@gmail.com grea09@gmail.com tho_blon@hotmail.com

3) Choix de l'algorithme

Pour réaliser notre projet, la solution la plus adaptée est l'algorithme de Dijkstra. En effet, nous ne recherchons qu'un seul chemin dans le graphe : le chemin le plus court. Étant donné que l'environnement contiendra différents obstacles, le robot ne pourra pas s'orienter dans la direction dans laquelle le point d'arrivée se trouve. De plus, les poids ne pourront pas être négatifs. Le programme doit être le plus efficace possible et le moins coûteux en terme de mémoire, tout en restant rapide. L'algorithme de Dijkstra est parfait pour ces contraintes.

4) Langages de programmation

4.1) Les langages existants :

Il est possible de réaliser le projet dans plusieurs langage de programmation qu'ils soient procédurales ou orienté objet. En effet, les contraintes imposées par le projet n'impliquent pas l'utilisation d'un langage précis.

4.2) Choix du type de langage :

Il existe deux grands types dans la programmation : le procédural et l'orienté objet.

Le procédural : chaque fonction ou procédure est associée à un traitement particulier pouvant être décomposé en sous-traitements jusqu'à obtenir des fonctions basiques. Globalement, le procédural travaille sur l'action ou le verbe.

Exemple : pour calculer l'aire d'un carré, le procédural conduira à faire :

```
104  calculAire(carre) //Le code faisant référence à un carré est donc "fondu"
et dispersé dans l'ensemble des programmes.
```

L'orienté objet : on manipule uniquement des objets c'est-à-dire des ensembles groupés de variables et de méthodes associées à des entités intégrant naturellement ces variables et ces méthodes.

Exemple : le sujet est prépondérant : disposant d'un objet Carre, on effectuera spontanément : Carre.calculAire() . Tout le code touchant à l'objet Carre se trouve ainsi regroupé.

Pour notre projet, nous utiliserons le type orienté objet. En effet, il présente un énorme avantage sur le procédural : les objets que l'on représente dans le code sont des objets réels qui existent dans le monde réel. Cela permettra de mieux visualiser les objets qui seront pris en charge (robot, obstacles, etc…). Par ailleurs, étant donné que

zahid.alexmehdi@gmail.com grea09@gmail.com tho_blon@hotmail.com

nous travaillons en groupe, l'orienté objet offre une meilleure organisation du code. Nous pouvons aussi se concentrer sur un problème à la fois avec des modules indépendants. Ainsi, chacun de nous comprendra plus facilement les implémentations réalisées et la réutilisation de l'application sera plus facile.

4.3) Choix du langage :

Parmi les différent langages orienté objet existant, nous ferons un choix entre le langage Java et C++. En effet, nous connaissons les bases de ces langages. Cela nous permettra de gagner du temps dans la réalisation du projet.

Un des critères majeur de notre projet est la rapidité. En effet, l'algorithme que nous allons utiliser devra être capable de travailler sur très grand nombre de nœud. Le programme devra en conséquence être le plus rapide possible. Il faut aussi que le langage permette de créer facilement des interface. Le langage C++ pourra plus facilement répondre à ce critère que le Java.

En effet, le langage C++ offre une bibliothèque très puissante : la bibliothèque Qt[1]. Cette bibliothèque offre de très nombreuses fonctionnalités pour réaliser des interfaces graphiques et faire du dessin .

5) Environnement

5.1) Niveau développement

Afin de pouvoir travailler efficacement et d'éviter tout problème d'interopérabilité, notre groupe travaillera à la fois pour les systèmes d'exploitation Windows et Linux. Cela permettra de vérifier la compatibilité du logiciel à réaliser sur différentes plateformes.

Nous utiliserons les conventions d'écriture décrite ici. Nous utiliserons Qt Creator pour réaliser le plus gros du projet.

1 Qt (prononcez *Quioute*) est un ensemble de programme et de classes qui permettent de faire des interface graphique et bien plus encore....

Alex Mehdi ZAHID zahid.alexmehdi@gmail.com
Antoine GRÉA grea09@gmail.com
Blon THO tho_blon@hotmail.com

5.2) Niveau fonctionnement

L'algorithme de Dijkstra travaille sur un graphe. Dans notre cas, le graphe sera représenté par une image. Chaque pixel de l'image représentera un nœud. De cette manière, nous n'aurons pas à découper la carte pour obtenir des cases représentant les nœuds.

Dans un premier temps, nous travaillerons sur une image simple dans un format sans perte comme BMP[2] ou PNG[3]. L'algorithme travaillera pixel par pixel. Cependant, cela impose des inconvénients d'ergonomie et de fonctionnalité.

Ensuite pour simplifier le travail du GUI nous utiliserons une autre technologie. Le langage XML[4] étant un standard très souple, on peut décrire l'environnement de manière plus naturelle. Il existe un format d'image qui utilise XML pour décrire des formes simples, le SVG[5].

Ce format présente de nombreux avantages :

- ✔ Il permet d'associer à n'importe quelles structures des mécanismes performants qui permettent de gérer les coûts et leurs calculs dans le SVG.
- ✔ On peut utiliser SVG pour écrire un script de transformation de l'image pour le rendre utilisable par Dijkstra.
- ✔ On peut modifier la qualité de l'image suivant la puissance matérielle utilisée.
- ✔ Le programme pourra bénéficier d'une interface simple et efficace.
- ✔ Tout est transparent pour l'utilisateur.
- ✔ Le SVG est peu gourmand espace mémoire et plus optimisé au niveau calcul.
- ✔ Et bien d'autre avantages qui vont bien aider à la phase de conception et de réalisation.

2 BMP : *Bitmap*. L'image est stocké sous forme brute, c'est à dire que chaque pixel a sa couleur stocké. Ceci est très gourmand en espace mémoire

3 PNG : *Portable Network Graphics*. Format d'image compresse sans pertes. Le PNG est connus pour son excellent support de la transparence et pour le fait qu'il soit libre.

4 XML : e*Xtensible Markup Language*. Il utilise des balises personnalisées pour décrire des données.
Ex: <Definition text='' blabla''/>
Il est à la base du SVG.

5 SVG : *Scalable Vector Graphics*. signifie « graphique vectoriel adaptable », et définit une image à l'aide de formes simples. Peut contenir toute sorte d'informations XML. Il suporte la transparence et est extrèmement léger.