**Projet « Nono le robot »**

1. **Présentation et données de base :**

Dans ce projet, nous allons faire un programme de calcul d’itinéraire pour un robot convoyeur dans un atelier de fabrication. Le robot permet de déplacer des pièces entre poste dans l’atelier. Nous devons concevoir ici un programme permettant de calculer un itinéraire entre deux points de l’atelier. Nous allons pour cela utiliser un algorithme de recherche combinatoire, c’est-à-dire un algorithme évaluant les différentes possibilités de mouvements pas par pas jusqu’à l’objectif.

Le robot dispose d’une carte de l’atelier modélisée par une matrice de caractères (voir exemple sur la figure 1) qui sera récupérée dans un fichier texte (map.txt). Le robot tient une case et peut se déplacer soit vers le haut, la gauche, la droite ou le bas à l’intérieur de la matrice (s’il n’y a pas d’obstacle dans les cases correspondantes bien sûr). Nous connaissons le point de départ et d’arrivée par leurs coordonnées dans la matrice (couple d’entier représentant le numéro de ligne et de colonne) également chargée depuis le fichier texte.

Départ : (Ld, Cd) Arrivée : (La, Ca)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 |
| 1 |  |  |  |  |  |
| 2 |  |  |  | 1 | A |
| 3 |  | 1 | 1 | 1 |  |
| 4 |  |  |  | 1 |  |
| 5 |  | D |  |  |  |
| 6 |  |  |  |  |  |

***Figure 1 :*** Matrice représentant l’atelier et déplacement du robot

Nous allons du coup utiliser une structure pour récupérer les informations dans le fichier texte. Nous auront donc un nom de problème, les positions d’arrivée et de départ, le nombre de ligne et le nombre de colonnes de la carte et la carte elle-même. La structure d’un problème à résoudre pourrait être la suivante :

struct coordonnee{

unsigned int num\_ligne,num\_col;

};

struct problem{

char nom[20];

struct coordonnee arrivee, depart;

int nb\_ligne,nb\_colonne;

char \*\*carte;

};

Rem : nous utiliserons une matrice de caractère pour pouvoir l’afficher directement. Du coup, une case vide est une case contenant un espace et tout autre symbole est un obstacle. Pour faciliter les choses, j’indiquerai le nombre de lignes et de colonnes bien qu’il puisse être calculé directement à la lecture du fichier. Pour finir, vous utiliserez une allocation dynamique pour mémoriser la carte qui doit pouvoir être manipulée comme une matrice…

1. **Recherche d’itinéraires :**

La recherche peut être représentée par un arbre des déplacements possibles partant du départ puis essayant de trouver l’arrivée en testant de proche en proche, c’est-à-dire en générant l’arbre au fur et à mesure jusqu’à trouver l’arrivée. Nous pouvons utiliser pour cela un algorithme de parcours en largeur. Les nœuds de l’arbre correspondent chacun à une position dans la matrice et un chemin pour y arriver (liste des positions depuis le départ). Le nœud racine est la position de départ et une liste vide. L’avantage d’un parcours en largeur est que la première fois que l’on trouvera l’arrivée, on aura le plus court chemin avec la liste des positions pour y arriver.

**1) Fonction « déplacements possibles » :**

Nous avons besoin d’une fonction qui à partir d’une position et de la matrice donne la liste des positions atteignables en un mouvement (haut, bas, gauche ou droite). Le robot ne revient pas en arrière. Vous avez donc également sa position au pas précédant. Vous utiliserez l’entête suivant pour réaliser la fonction Deplacement :

Fonction Liste de coordonnee **Deplacement** ( E/ Atelier, E/ Pos\_cour)

**Paramètres** :

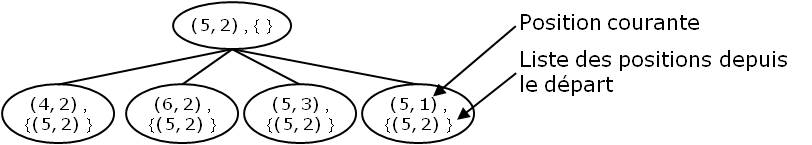
Atelier (problem) données du problème à résoudre

Pos\_cour (coordonnee) position courante

Rem : notation algorithmique, à vous de choisir l’implémentation en C.

**2) Recherche par parcours en largeur**

Nous allons maintenant réaliser l’algorithme de parcours en largeur. Par exemple, voici l’arbre correspondant aux déplacements possibles depuis la position de départ de la figure précédente.



Remarque importante : l’arbre n’est en fait jamais généré. Il représente juste la recherche.

Comme nous l’avons vu, le parcours en largeur correspond à l’utilisation d’une file de nœuds à visiter. Après initialisation de la file avec la position de départ, le fonctionnement global sera donc :

* Enlever le nœud en tête de file
* Traiter le nœud, c’est-à-dire ici tester si c’est l’arrivée,
* Sinon générer et mettre dans la file ses enfants (déplacements possibles depuis ce nœud).
* Si la file est vide, c’est que l’arrivée est inaccessible.

Vous aurez besoin des structures suivantes :

Type structuré **Etape**

**Début**

pos (coordonnee) position courante

chemin (liste de coordonnee) chemin pour atteindre ce noeud

**Fin**

Ecrire la fonction Parcours\_Larg() qui réalise le comportement décrit précédemment et retourne la liste des positions correspondant au chemin trouvé.

Fonction Liste de coordonnee Parcours\_Larg(E/ Atelier)

**Paramètres** :

Atelier (problem) données du problème à résoudre

3) Première amélioration : éviter les cycles

Nous voulons améliorer notre algorithme en évitant de générer les branches qui correspondent à des cycles (et il y en a énormément !). Quelle opération doit être réalisée pour éviter les cycles, c’est-à-dire de repasser par des positions précédentes ? Réalisez pour cela une fonction dont vous définirez l’entête et dites où l’insérer dans votre algorithme.

1. Autres améliorations pour la suite

Vous pouvez ensuite utiliser d’autres méthodes que la recherche en largeur :

* + Recherche bidirectionnelle, (recherche du départ vers le but et inversement)
  + meilleur d’abord, (exploration prioritaire du nœud le plus proche du but)
  + A\*, (exploration prioritaire du nœud de plus faible coût (heuristique + longueur du chemin actuel))

Une autre amélioration possible sera de rendre générique votre méthode de recherche c.à.d. de ne pas lui donner les données du problème mais uniquement les fonctions permettant de les calculer tel que le tri de la bibliothèque standard (sort).