Projet – MOGPL

Résolution d'un casse-tête :

Rush Hour™

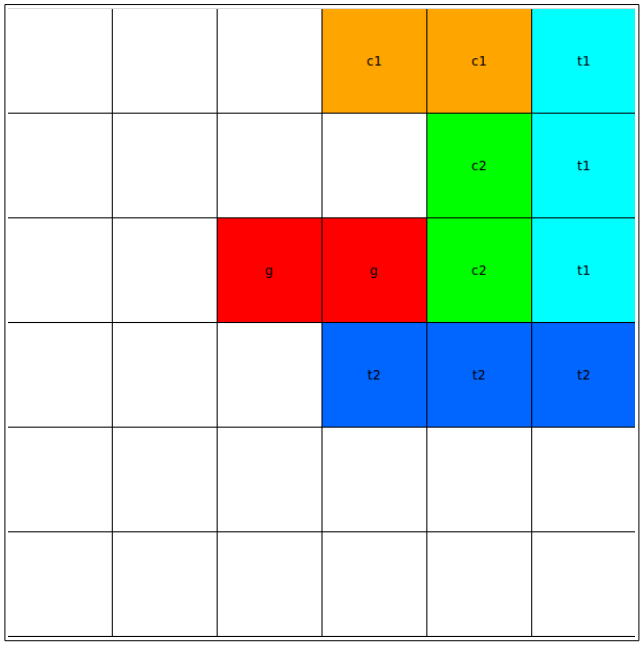
Coumba SALL – Emilie FARJAS

2016-2017

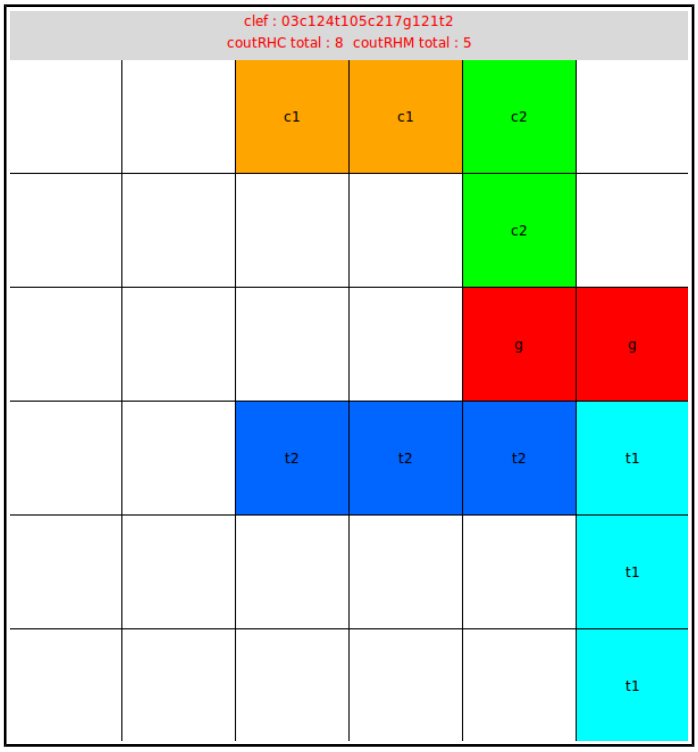
Question 1.

Nous exhibons un exemple permettant de voir qu'une solution optimale pour RHC n'est pas nécessairement optimale pour RHM :

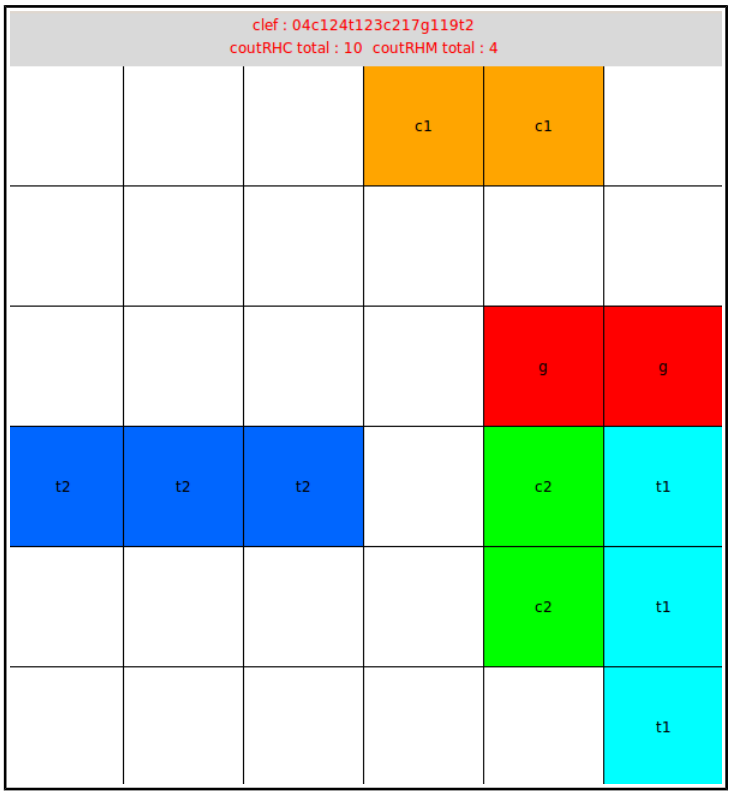
Prenons une configuration initiale contenant 3 voitures (dont la voiture rouge) et 2 camions :



Si nous effectuons une recherche de la solution optimale pour RHC nous obtenons 8 cases, alors que la solution RHM est de 5 mouvements.



En revanche, lorsque nous effectuons la recherche pour trouver une solution optimale pour RHM, nous obtenons 4 mouvements, alors que le coût RHC correspondant est de 10 cases.



Cette exemple montre bien que pour une même configuration de départ, une solution optimale pour RHC, n'est pas forcément optimale pour RHM.

Nous avons développé ce projet en python.

Question 2.

Coder une fonction de lecture d'un fichier d'entrée, qui génère une structure de données représentant la configuration de départ. La structure choisie devra permettre d'accéder en (1) à n'importe quelle case de la grille (ainsi qu'à l'identifiant du véhicule qui l'occupe si la case est occupée).

Nous avons développé la fonction lectureFichier(strFileName), qui a pour paramètre le chemin du fichier texte de configuration initiale.

Voici l'algorithme pour la partie Gurobi :

- Nous ouvrons le fichier dont le chemin est passé en paramètre.
- Nous lisons la première ligne et nous obtenons un 2-tuples correspondant aux dimensions de la grille.
- Nous initialisons à 0 une matrice à 2 dimensions de la même taille que les dimensions de la grille.

- nous initialison les compteurs de lignes et de colonnes à 0.
- Pour chaque ligne du fichier
    Pour chaque case(séparée par des espaces) de la ligne
        Nous enregistrons dans ma matrice(à la position I,j) la valeur lue à la position I,j correspondante
        Nous augmentons la valeur de la colonne i de 1
    Nous affectons 0 à la valeur de la colonne I
    Nous augmentons de 1 la valeur de la ligne j
- Nous retournons la matrice de configuration ainsi que le tuple de dimensions de la grille.

Pour la partie Dijkstra, nous avons modifié la fonction de lecture.
En effet, dans le cadre d'une optimisation de l'exécution du code, nous avons créé 2 structures supplémentaires :
    => un dictionnaire de voitures : répertorie toutes les voitures présentent dans le jeu.
    Clef : idVoiture ; Value : objet voiture
    => un dictionnaire de coordonnées : clef : coordonnée d'un case :tuple (i,j) ; Value : objet voiture.

Nous avons ajouté les fonctionnalités suivantes :

- Pour chaque case lue :
    - tester si le vehicule est la voiture rouge
        - si oui, vérifier que son emplacement est sur la bonne ligne
            - si l'emplacement est mauvais, on sort du système
    - si la case est occupée par un véhicule,
        - si le dictionnaire de véhicules ne contient pas ce dernier.
            => on crée l'objet vehicule et on l 'insère dans le dictionnaire de voitures.
        - sinon, il existe déjà et nous pouvons alors calculer son orientation
        => on crée l'objet vehicule dans le dictionnaire de coordonnee
            avec pour clef (i,j)
    - sinon, la case est vide et nous créons la coordonnée dans le dictionnaire de coordonnées et nous lui affectons une valeur Null.
    => nous mettons à jour la matrice de configuration initiale avec la valeur de la case lue.
=> nous incrémentons le compteur de ligne et mettons à 0 celui des colonnes

Pour l'algorithme Dijkstra , dicoVoiture et dicoCoordonnees sont les structures importantes pour l affichage. La matrice 2D n'est pas utilisée par la suite. Elle ne sert qu'à refleter la grille provenant du fichier afin de générer la clef de configuration initiale. Nous avons représenté chaque configuration par une clef, qui est une chaine de caractères de la forme : 02c103t105c213g1 qui représente la sequence de position-idVehicule de tous les véhicules présents sur la grille à une configuration donnée. Ici nous avons 02c1 correspondant au marqueur de la voiture 1 présent sur la case 2 ; nous avons dû modifier l'identifiant de la voiture rouge de g à g1 afin de respecter la lecture d'une

nouvelle information tous les 4 caractères.
L'avantage de cette représentation (au lieu d'une matrice à 2 dimensions) est la recherche de la position de chaque véhicule dans la configuration donnée est en O(n) avec n représentant le nombre de voitures , alors que dans la matrice à 2 dimensions nxn, la recherche est pour le pire des cas en O(n2) correspondant au nombre total de cases dans la matrice. De plus la recherche de voisins est plus rapide étant donné que nous ne ciblons que la partie de la chaine de caractères
L'inconvénient est un algorithme de traduction des clefs pour l'affichage plus complexe.

Dans les 2 cas, l'accès à n'importe quelle case de la grille se fait en Θ(1)
- dans le 1er cas, matrice à 2 dimensions ayant comme index x et y
- dans le 2d cas, la structure est un dictionnaire ayant pour clef le tuple (x,y)

Question 3.

Coder une fonction d'affichage qui prend en entrée la structure de données représentant une conguration quelconque et qui affiche la grille.
L'affichage peut se faire dans la console tant qu'il reste lisible.

Nous avons également développé deux fonctions d'affichage différentes : une pour Gurobi et l'autre pour Dijkstra.

Pour Gurobi, l'affichage s'effecture directement dans la console.
La fonction d'affichage prend en paramètre le dictionnaire Y de marqueurs de chaque véhicule pour chaque mouvement.
Le dictionnaire Y a donc une clef constituée d'un 4-tuple : (idVoiture, case début, case fin, cout) et a une valeur à 1 si le déplacement correspondant fait parti du résultat optimum.

- initialiser un compteur de déplacement à 0
- Pour chaque clef du dictionnaire Y :
    Si la valeur du dictionnaire pour cette clef est égale à 1
        nous augmentons le compteur et nous affichons le déplacement du véhicule
=> afficher la valeur totale du nombre de déplacement effectué

Pour Dijkstra, l'affichage est en mode graphique. Nous utilisons la bibliothèque Tkinter.
Nous passons donc en paramètre le dictionnaire de Coordonnées.

- Nous créons une fenetre, racine de l'interface.
- Nous créons 2 label permettant d'afficher du texte dans la fenetre
    - la clef
    - les résultats des couts totaux pour RHC et RHM
- Nous créons une liste de rectangles (pour le fond de la case) ainsi qu'une

liste de textes (pour le texte de chaque case), ayant comme dimension la largeur de la grille.
- Nous créons un cadre dans la fenetre qui contiendra les éléments (rectangles et textes qui nous allons créer).
- Pour chaque ligne i
    - Pour chaque case j en rangée
        => récupérer le nom du véhicule depuis le dictionnaire de coordonnées si celui-ci n'est pas Null
        => creer le rectangle associe avec comme une dimension proportionnelle au nombre de cases et une position calculée en fonction de i et j, ainsi qu'une couleur de fond paramétrée dans l'objet voiture accessible par le dictionnaire de coordonnées.
        => creer un objet texte et le positionner sur le rectangle associé.
- Nous créons les boutons charger, résoudre et quitter. Dans notre version actuelle, seul le bouton quitter est fonctionnel.
- Nous démarrons la boucle Tkinter qui s'interompt quand on ferme la fenêtre.

Nous avons une seconde methode : affiche_error(message) qui prend en paramètre une chaine de caractère et qui est appelée lorsque la voiture rouge se trouve sur une mauvaise ligne.

Question 4.

Donner l'expression de la fonction objectif :
a. si l'on souhaite résoudre RHM (minimiser le nombre de mouvements) ;

$\min \sum_i \sum_k y_{i,j,l,k} \qquad \forall j;k$

b. si l'on souhaite résoudre RHC (minimiser le nombre total de cases déplacement).

$\min \sum_i \sum_k [\ |l-j|\%6]\ y_{i,j,l,k} \quad \forall j;k$

Question 5.

Donner les contraintes, impliquant les variables xi;j;k et yi;j;l;k, qui assurent que :

a. la voiture rouge est positionnée devant la sortie au terme du dernier mouvement ;

$x_{g,17,k} = 1$
$y_{g,j,17,k} = 1$

b. au plus un véhicule est déplacé par tour ;

$\sum_i \sum_j \sum_l y_{i,j,l,k} \quad \forall k$

c. la position du marqueur d'un véhicule i est bien mise à jour si le véhicule i se déplace.

$X_{i,j,k-1} + x_{i,l,k} + y_{i,j,l,k} = 3$

Question 6.

Expliquer pourquoi se un sous-ensemble de toutes les variables binaires potentielles doivent être introduites dans le modèle.

Pour chaque déplacement nous ne regardons que la ligne ou la colonne impliquée dans ce mouvement. Nous ne regardons pas toute la grille.
De plus nous pouvons enlever la dernière colonne pour une voiture ainsi que les 2 dernières colonnes pour les camions car les marqueurs de ces véhicules ne pourront jamais se positionner sur ces cases.

Question 7.

Coder la fonction qui fait appel un solveur de programmation linéaire en variables binaires (Gurobi) pour résoudre Rush Hour. La solution devra être affichée sous la forme d'une séquence de déplacements à réaliser afin de résoudre le casse-tête

Apres la lecture du fichier, nous générons les structures de données nécessaires à la résolution du programme linéaire :
- un dictionnaire “dicoVehicInfo”, regroupant les informations des vehicules
clef :id du vehicule
valeur: liste de triplets : (orientation, coordoneeVariable en fonction de l orientation, taille vehicule)

- un dictionnaire “dicoConfigInit” : donne la position d'un vehicule
clef : id du vehicule
valeur : numero de la case où le véhicule se trouve (1 a 36)

Nous définissons un modèle Gurobi, auquel nous associons les variables de décisions définies grâce à la fonction genereVarDecision(…).
Les variables sont définies grâce aux 3 dictionnaires : x(i,l,k) - y(i,j,l,k) - z(i,j,k) :
* dicoX , qui définit l'emplacement des marqueurs de tous les vehicules pour

chaque tour.
  * dicoY , qui définit les mouvements de tous les vehicules pour chaque tour
  * dicoZ , qui définit toutes les cases occupees par tous les vehicules pour chaque tour.
A chaque valeur valides de ces dictionnaires est créée une variable de type binaire, le nom de ces variables suit une nommenclature : nom du dictionnaire ainsi que les valeurs de ses clefs valides correspondantes.

Une fois les variables ajoutées dans Gurobi, nous mettons à jour le modèle pour intégrer ces variables.

Nous générons alors les contraintes associées à notre PL.
Nous mettons à nouveau à jour le modele pour intégrer les contraintes.
Nous définissons un paramètre de limite d'exécution de 2 minutes.
Nous définissons l'objectif qui est ici de Minimiser le nombre total de déplacement de tous les véhicules à tous les tours du jeu. Il est représenté par le dictionnaire dicoY (dictionnaire de mouvements de tous les véhicules à toutes les étapes du jeu).

Et enfin nous lançons la résolution du problème. Nous affichons le résultat de la séquence des mouvements optimales dans la console.

PARTIE DIJKASTRA

Question 8.

Coder une méthode de résolution de RHC fondée sur l'utilisation de l'algorithme de Dijkstra dans le graphe des configurations.
Quelle est la condition d'arrêt de l'algorithme ?

Nous avons représenté le graphe des configurations par un couple :
- liste triplets (clef de configuration valide, clef de configuratioin voisine valide, cout de deplacement entre les 2)
- liste de clefs buts (configurations valides buts caractérisées par la séquence de caractères : “17g1”) sans doublons

- nous initialisons
    - une variable représentant le resultat trouvé pour RHC : couple de valeurs (clef de la configuration but, chemin trouvé correspondant)
    - une variable contenant le résultat de l'algorithme Dijkstra à None
    - une variable de cout minimum de RHM à l'infini
    - une variable de cout minimum de RHC à l'infini
- Pour chaque clef but de la liste des clefs de configurations but valides,
    => nous obtenons grâce à la fonction Dijkstra (ayant pour paramètres la liste des triplets(arc entrant, arc sortant, coût), un triplet (coût RHC, coût RHM, cheminOptimal RHC)
    - Si le coût minium RHC trouvé jusqu'à présent > à celui nouvellement calculé par l'appel de Dijkstra, alors
      => on remplace la valeur du coût minimum RHC par la nouvelle

valeur de Dijkstra
=> on enregistre dans le couple de resultat RHC les valeurs (clefBut, triplet (coût RHC, coût RHM, cheminOptimal RHC) associé)

Nous obtenone alors une chaine de caractère correspondant au meilleur chemin pour RHC (séquence de clefs de configuration séparées par une virgule, ainsi que la clef de configuration but associée.

La condition d'arrêt de l'algorithme est la position du marqueur de g devant être en 17, sinon La condition d'arrêt est tous les noeuds ont été visités.

Question 9.

Coder une fonction qui permet d'afficher la séquence de déplacements à réaliser afin de résoudre le casse-tête depuis une configuration de départ donnée en entrée (l'affichage en mode texte dans la console est accepté), ainsi que la valeur correspondante de la fonction objectif.

Pour chaque clef de configuration appartenant à la liste de configurations du chemin optimal trouvé,

- appeler une fonction mettant à jour le dictionnaire de coordonnées basée sur la clef de configuration (chaine de caractères) passée en paramètre
- appeler la fonction d'affichage qui prend en paramètre le dictionnaire de coordonnées et qui génère une fenêtre graphique représentant une étape de la séquence de déplacement à réaliser pour résoudre le casse-tête.

Nous pouvons donc visualiser la séquence complète des déplacements à réaliser pour résoudre le casse-tête.

Question 10.

Expliquer comment adapter la méthode de la question 8 si l'on souhaite résoudre RHM (minimiser le nombre de mouvements). Implanter cette fonctionnalité dans le code.

Nous pouvons adapter la méthode de la question 8 en changeant la valeur de tous les arcs dans l'algorithme dijkstra. Ces coût, initialement représentaient le coût de déplacement en nombre de cases, vont à présent représenter le coût d'un mouvement. Nous allons donc les mettre tous à 1 car un déplacement correspond à un mouvement.

Question 11.

Expliquer comment adapter la méthode de la question 8 si l'on souhaite compter le nombre de configurations réalisables (en un nombre quelconque de mouvements de véhicules) depuis la configuration de départ. Implanter cette fonctionnalité dans le code.

Pour arriver à une configuration nous avons découpé l'algorithme de méthode de résolution en 2 parties :
- la première nous donne, à partir du graphe ????, tous les chemins possibles réalisables jusqu'à arriver à un état but.
- la seconde applique Dijkstra sur tous les chemins possibles trouvés à la première étape ci-dessus.

Question 12.

Fournir les temps de résolution obtenus pour chaque méthode implantée et chaque instance ou groupe d'instances, sous la forme d'un tableau ou d'une courbe.

Pour la méthode fondée sur l'algorithme de Dijkstra, il pourra être intéressant également d'étudier le comportement en fonction du nombre de configurations réalisables depuis la configuration de départ.

Tableau du temps de résolution pour chaque instance :

	Dijkstra		Gurobi	
Nom Fichier	Nombre instance	Temps execution	simplex iterations	Temps execution
puzzles/test/test2.text	1	9.39E-05	3879	0.35
puzzles/expert/jam32.txt	115	0.5833921433		
puzzles/expert/jam36.txt	198	10.10232687		
puzzles/expert/jam35.txt	271	17.500287056		
puzzles/expert/jam37.txt	22	0.5658948421		
puzzles/expert/jam40.txt	199	11.1918308735		
puzzles/expert/jam34.txt	56	3.6109409332		
puzzles/expert/jam39.txt	110	5.7317960262		
puzzles/expert/jam33.txt	30	2.0316081047		
puzzles/expert/jam31.txt	1478	126.703253984		
puzzles/expert/jam38.txt	575	39.5294561386		
puzzles/debutant/jam3.txt	104	1.2256379128		
puzzles/debutant/jam9.txt	192	18.7931468487		
puzzles/debutant/jam6.txt	116	5.2585730553		
puzzles/debutant/jam2.txt	1084	644.538795948		
puzzles/debutant/jam8.txt	2	0.0184950829		
puzzles/debutant/jam1.txt	172	3.2833428383		
puzzles/debutant/jam10.txt	380	26.41437006		
puzzles/debutant/jam5.txt	86	4.6109929085		
puzzles/debutant/jam7.txt	849	164.240821838		
puzzles/debutant/jam4.txt	355	2.9302859306		
puzzles/avance/jam21.txt	392	1.9436089993		
puzzles/avance/jam29.txt	20	1.3968441486		
puzzles/avance/jam25.txt	253	47.4717769623		
puzzles/avance/jam22.txt	4255	786.004912138		
puzzles/avance/jam27.txt	242	10.5406241417		
puzzles/avance/jam23.txt	423	25.954985857		
puzzles/avance/jam30.txt	115	1.5252270699		
puzzles/avance/jam26.txt	90	6.6751630306		
puzzles/avance/jam28.txt	168	5.5968639851		
puzzles/avance/jam24.txt	1730	250.436683893		
puzzles/intermediaire/jam12.txt	27	0.4297418594		
puzzles/intermediaire/jam20.txt	878	66.1797149181		
puzzles/intermediaire/jam13.txt	2475	794.677351952		
puzzles/intermediaire/jam16.txt	434	30.1501009464		
puzzles/intermediaire/jam17.txt	925	39.1613409519		
puzzles/intermediaire/jam15.txt	216	1.2737441063		
puzzles/intermediaire/jam19.txt	156	1.1842000485		
puzzles/intermediaire/jam11.txt	689	10.5749890804		
puzzles/intermediaire/jam18.txt	2644	157.830566883		
puzzles/intermediaire/jam14.txt	-1	-1		

Nota Bene :

Les valeurs à -1 signifient que l'execution dépasse 60 minutes.

Nous n'avons malheureusement pas pu tester Gurobi sur de plus grandes instances dû à l'impossibilité d'utiliser la licence etudiant en dehors de la faculté. La licence personnelle ne permet de tester que des modèles très petits.

Comportement en fonction du nombre de configurations réalisables depuis la configuration de départ :