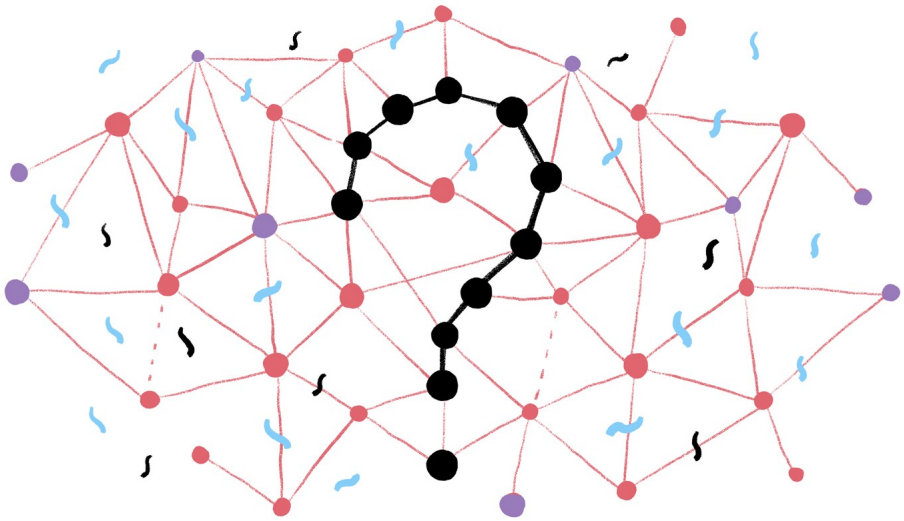
Atelier d'approfondissement en informatique

Graphes et algorithmes A*

E3 - PR3602

Professeur : Michel COUPRIE
Etudiants : Vincent BARBOSA VAZ, Cécile POV
ESIEE Paris - **30/04/2018**



Source : https://medium.com/the-mission/what-is-the-algorithm-for-success-8d820ce167b0

Avant-propos

Vous utilisez un notebook Jupyter. Un document executable et lisible pour les humains.
Le code et le texte sont écrits dans des cellules.

- pour éditer une cellule, double-cliquez ;
- pour executer une cellule, ctrl+entrée

Pour commencer, executez la cellule qui suit :

In [1]:

```
%%HTML
<style>
em {
    color: green;
}
strong {
    color: rgb(53, 70, 180);
}
u {
    text-decoration: underline;
}
</style>
```

N.B. :

Pour simplement visualiser le cours sans le générer manuellement,

- aller dans *Kernel*
- puis cliquer sur *Restart & Run All*

Pour tester l'algorithme,

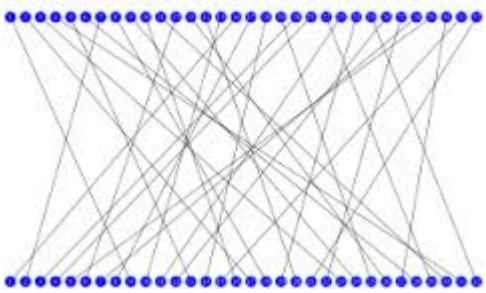
- executer tout le notebook (comme précédement)
- puis aller directement à la section *Paramétrage et test de l'algorithme*

Sujet

Le problème que nous allons résoudre, connu sous le nom de problème de l'affectation, a de nombreuses applications en sciences de l'ingénieur. Voici l'une d'entre elles.

Une administration nationale doit organiser la réaffectation de plusieurs dizaines de ses agents dans des centres répartis dans toute la France. Un nouveau poste doit être proposé à chacun de ces agents, et, en fonction de la personne et du poste, un coût a été calculé, tenant compte de l'indemnité de déménagement, des formations nécessaires, etc. Si N est le nombre d'agents à réaffecter (et donc aussi le nombre de postes), les résultats de ces calculs sont rangés dans une matrice NxN appelée C. Ainsi, C[i,j] représente le coût de l'affectation de l'agent i au poste j.

Notre but est de décider d'une affectation pour chaque agent, de manière à minimiser globalement le coût total de l'opération.



Algorithme naïf

Un premier algorithme consiste à considérer toutes les affectations possibles et à choisir la moins coûteuse. Combien y en a-t-il ? Si l'on suppose qu'une affectation peut être évaluée en une microseconde, et que l'on dispose de trois mois pour faire le calcul, quelle est la valeur maximum de N possible pour envisager d'appliquer cette méthode ?

Nombre de solutions possibles :
Dans le cas du problème de l'affectation et pour une matrice $N \times N$,

- le premier agent peut être affecté à N postes ;
- le second agent peut être affecté à N-1 postes ;
- [...]

Il y a donc N! solutions possibles.

Valeur maximum de N possible :
Pour une durée de calcul de 3 mois,

- trois mois : $3 \times 31 \times 24 \times 60 \times 60 \times 10^6 \approx 8 \times 10^{12} \mu s$;
- la factorielle immédiatement inférieure ou égale à cette valeur est 15!
  - $15! \approx 1 \times 10^{12}$
  - $16! \approx 2 \times 10^{13}$

Valeur de N possible maximale : 15
Il n'est donc pas envisageable d'appliquer cette méthode pour résoudre ce problème.

Automatisation du calcul
On se pose la question dans le cas général de l'efficacité de cette méthode.
Cette fois-ci on estime que l'affectation est prise au sens local de l'algorithme,
c'est à dire une étape dans la recherche de la solution finale, soit un noeud du graphe complet.

Ci-dessous un premier algorithme qui détermine le nombre de noeuds d'un GRP de niveau n, passé en paramètre.

In [2]:

```
# Utilisation de l'algorithme, modifier le(s) paramètre(s)
param = 16 # nombre d'agents à affecter

#----------------------------
def nodeGRP(n):
    # n : niveaux de l'arbre, nombre d'affectations
    P = 1 # racine de l'arbre, noeud initial
    S = n # nombre de noeuds successeurs

    #print("P %d S %d" % (P, S))
    for i in range(n, 0, -1): # on décompose chaque noeud
        P += S
        S = S*(i-1)
        #print("P %d S %d" % (P, S))
    return P

print("Nombre de noeuds pour une matrice {n}X{n} = {s}".format(n=param, s=nodeGR
P(param)))
```

```
Nombre de noeuds pour une matrice 16X16 = 56874039553217
```

nodeGRP(2) : 5
nodeGRP(3) : 16
nodeGRP(4) : 65
nodeGRP(10) : 9864101
nodeGRP(15) : 3554627472076

On voit que le nombre de noeuds croit très rapidement ! L'idée d'explorer tout le graphe parait vite inimaginable.

Ci-dessous un second algorithme qui va lancer itérativement le premier algorithme en comparant le nombre de noeuds retournés à une limite. Cette limite représente le nombre d'instructions maximal executables par l'algorithme pour une période donnée. L'algorithme retourne la profondeur maximale du GRP pouvant être exploré entièrement.

In [3]:

```
# Utilisation de l'algorithme, modifier le(s) paramètre(s)
speed = 1000000 # nombre d'instructions réalisées en 1 seconde
time = 3*31*24*60*60 # durée de calcul en seconde
num_instructions = time*speed # nombre d'instructions réalisables dans le temps
 imparti

#----------------------------
def maxN(num_instructions):
    n = 0
    while(nodeGRP(n) <= num_instructions): # tant que l'on peut encore effectuer
 des instructions
        #print("n",n)
        #print("taille",nodeGRP(n))
        #print("num_instructions",num_instructions)
        #print("")
        n+=1
    return n
print("Nombre d'instructions par seconde :", speed)
print("Durée de calcul : {t}s".format(t=time))
print("Nombre d'instructions réalisables :", num_instructions)
print("Valeur maximale de N possible :", maxN(num_instructions))
```

```
Nombre d'instructions par seconde : 1000000
Durée de calcul : 8035200s
Nombre d'instructions réalisables : 8035200000000
Valeur maximale de N possible : 16
```

Pour un algorithme dont la performance a été donné dans les consignes, en trois mois de calcul sans interruptions avec une vitesse de calcul d'une instruction par microseconde, seulement 16 affectations au maximum sont possibles ! Le résultat est assez surprenant au vu de la vitesse de calcul cependant, il ne faut pas négliger la taille extravagante que peut prendre un GRP, et donc toutes les possibilités à comparer dans le cas d'un algorithme naïf.

Algorithme Glouton

On considère maintenant l'algorithme suivant :

- choisir la valeur minimale de la matrice C ;
- réaliser l'affectation correspondante ;
- retirer la ligne et la colonne contenant cette valeur ;
- et recommencer.

Ce type d'algorithme s'appelle un algorithme glouton. Montrez par un contre-exemple simple que l'algorithme glouton ne trouve pas toujours à la solution optimale pour ce problème, et peut même donner un résultat arbitrairement éloigné de la solution optimale.

Soit les matrices de coûts suivantes : matrice initiale, solution gloutonne et solution optimale.

$$\begin{bmatrix} 5 & 3 & 1 \\ 4 & 2 & 6 \\ 0 & 200 & 0 \end{bmatrix} \begin{bmatrix} 5 & 3 & 1 \\ 0 & 1 & 6 \\ 0 & 200 & 0 \end{bmatrix} \begin{bmatrix} 5 & 3 & 1 \\ 4 & 2 & 6 \\ 0 & 200 & 0 \end{bmatrix}$$

$$Initiale \ - \ Gloutonne \ - \ Optimale$$

On remarque que l'algorithme glouton à fait des choix **optimums locaux** :

- le coût d'affectation le plus faible pour la ligne 1 est 1 ;
- pour la ligne 2 : 0 ;
- pour la ligne 3 : 200 puisque les coûts de valeur 0 ne sont plus disponibles.

Le score pour l'algorithme glouton est de $1 + 0 + 200 = 201$
Le score pour la solution optimale est de $1 + 2 + 0 = 3$

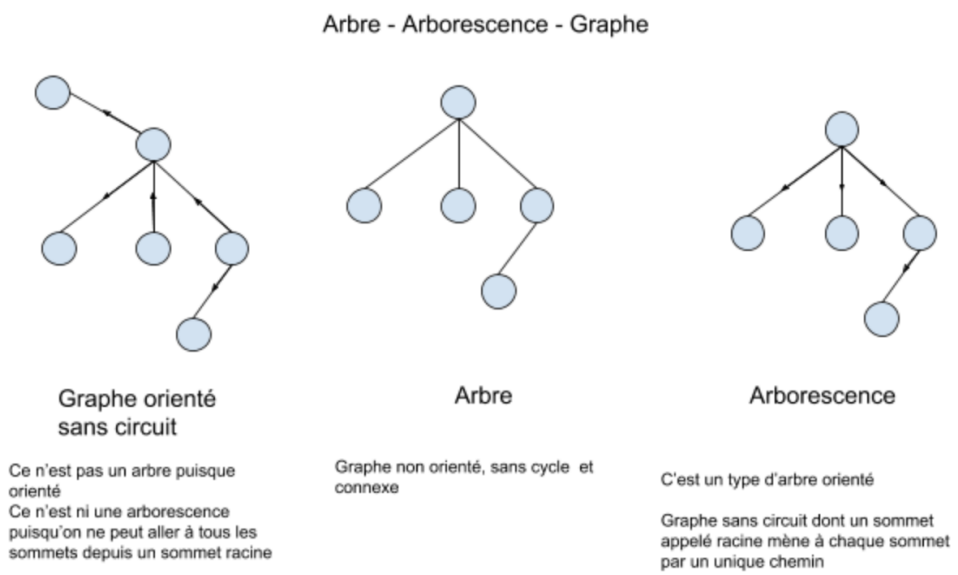
Dans certains cas cette approche permet d'arriver à un optimum global, mais dans le cas général c'est une heuristique.
En choisissant localement la meilleure solution, l'algorithme glouton n'a pas trouvé la solution optimale.

Notions essentielles

Avant d'aborder l'atelier, et au terme de votre préparation, vous devez être en mesure de répondre correctement aux questions suivantes :

- Qu'est-ce qu'un Graphe de Résolution de Problème (GRP), relativement à un problème donné ?
- Quel GRP proposeriez-vous pour le problème de l'affectation ?
- Quel est, schématiquement, le fonctionnement d'un algorithme A* ?
- Que représentent les symboles g, h et f dans l'algorithme ?
- Quelle est la condition sur h pour que l'on parle d'algorithme A* ?

Rappels



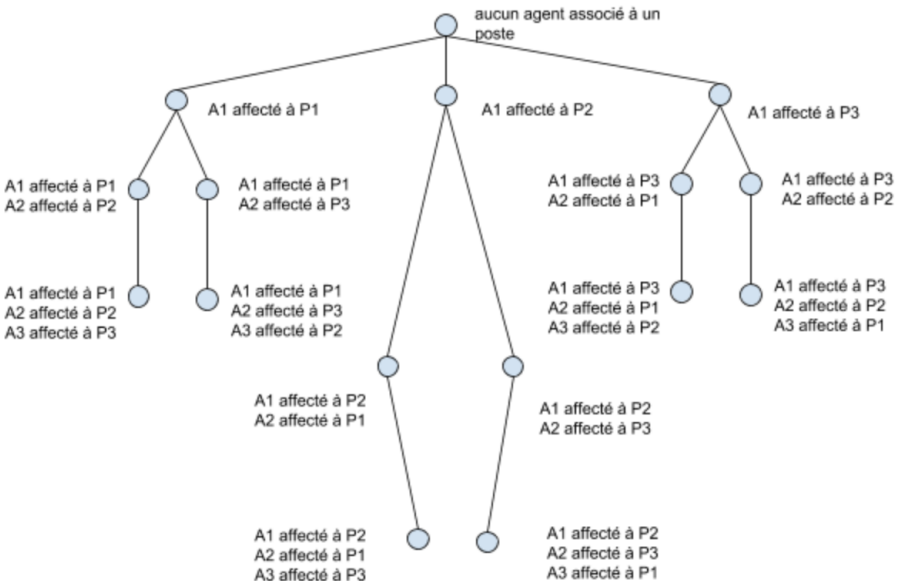
Qu'est-ce qu'un Graphe de Résolution de Problème (GRP), relativement à un problème donné ?

Un GRP est une modélisation d'un problème sous forme de graphe.

Propriétés d'un GRP :

- un graphe connexe et sans cycle : une arborescence ;
- les sommets sont les états possibles du problème ;
- il existe une arc u = (i, j) dans le graphe si une règle permet de passer de l'état i à l'état j ;
- on associe un coût c(u) à cet arc ;
- on distingue l'état initial et les états finaux ;
- un chemin de l'état initial à un état final constitue une solution au problème.

Quel GRP proposeriez-vous pour le problème de l'affectation ?



- chaque sommet représente un état du système : l'affectation des agents aux postes ;
- sommet initial : aucun agent n'est affecté à un poste ;
- sommet finaux (les buts possibles) : tout agent est affecté à un poste unique.

Quel est, schématiquement, le fonctionnement d'un algorithme A* ?

Soit :

- G : un graphe de recherche qui consiste uniquement en un sommet de départ d
- OUVERT : une liste contenant les sommets découverts et non développés
  (les noeuds dans OUVERT sont les feuilles de l'arbre de recherche)
- FERME : une liste contenant les sommets découverts et développés
  (les noeuds dans FERME sont les autres noeuds)

Boucle de l'algorithme :

- Tantque OUVERT non vide
- On sélectionne le premier sommet de la liste OUVERT que l'on met dans FERME et que l'on appelle n
- Si n est un sommet but, FIN : solution optimale trouvée
- Sinon on développe n, produisant l'ensemble M de ses successeurs que l'on memorise comme successeurs de n dans G
- On réordonne la liste OUVERT, soit arbitrairement, soit selon des heuristiques

Que représentent les symboles g, h et f dans l'algorithme ?

Soit n, un sommet pris au hasard dans le GRP, dans l'algorithme, on a :

- d représente le sommet initial ;
- g estime le coût d'un chemin optimal de d au sommet n ;
- h estime le coût d'un chemin optimal de n à un but : c'est la fonction heuristique choisie ;
- f représente la fonction d'évaluation ;
- f(n) estime le coût d'un chemin optimal de d à un but passant par n.

L'application f peut se mettre sous la forme :

$$f(n) = g(n) + h(n)$$

Quelle est la condition sur h pour que l'on parle d'algorithme A* ?

On parle d'algorithme A* si, pour tout sommet n :

$$h(n) \leq h(n^*)$$

Autrement dit, que l'algorithme trouve un chemin optimal du sommet initial à un but.

Heuristiques

Heuristique nulle : $h_0(n)$

$$h_0(n) = 0$$

C'est l'heuristique triviale. Le coût estimé pour chaque sommet est nul. Autrement dit, on estime très mal le coût d'un chemin optimal de n à un but.

Ceci dit il s'agit toujours d'un algorithme A* puisque :

$$h_0(n) = 0$$

$$0 \leq h(n^*)$$

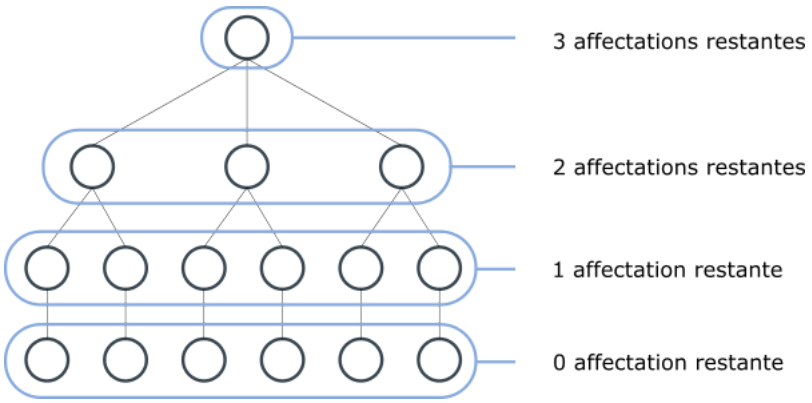
$$h_0(n) \leq h(n^*)$$

C'est l'algorithme de Djisktra.

Heuristique $h_1(n)$: nombre d'affectations restantes

$$h_1(n) = nombre\ d'affectations\ restantes\ (si\ coût >= 1)$$

Pour chaque noeud n, l'estimation $h_1(n)$ correspond aux nombres d'affectations restantes, c'est à dire le nombre d'affectation avant d'atteindre un sommet but.



Cette heuristique fonctionne dans le cas où :

- les coûts sont ≥ 1 *1*
- les coûts sont des entiers *2*

1 : en effet pour respecter la condition $h(n) \leq h(n^*)$. Si les coûts sont inférieurs à 1 on peut surestimer la valeur d'un chemin menant à un but.
2 : imaginons une matrice de coûts de flottants compris entre 1 et 2, les estimations ne prendront pas en compte la différence de coup "decimale" et donc l'heuristique équivaut à l'heuristique nulle.

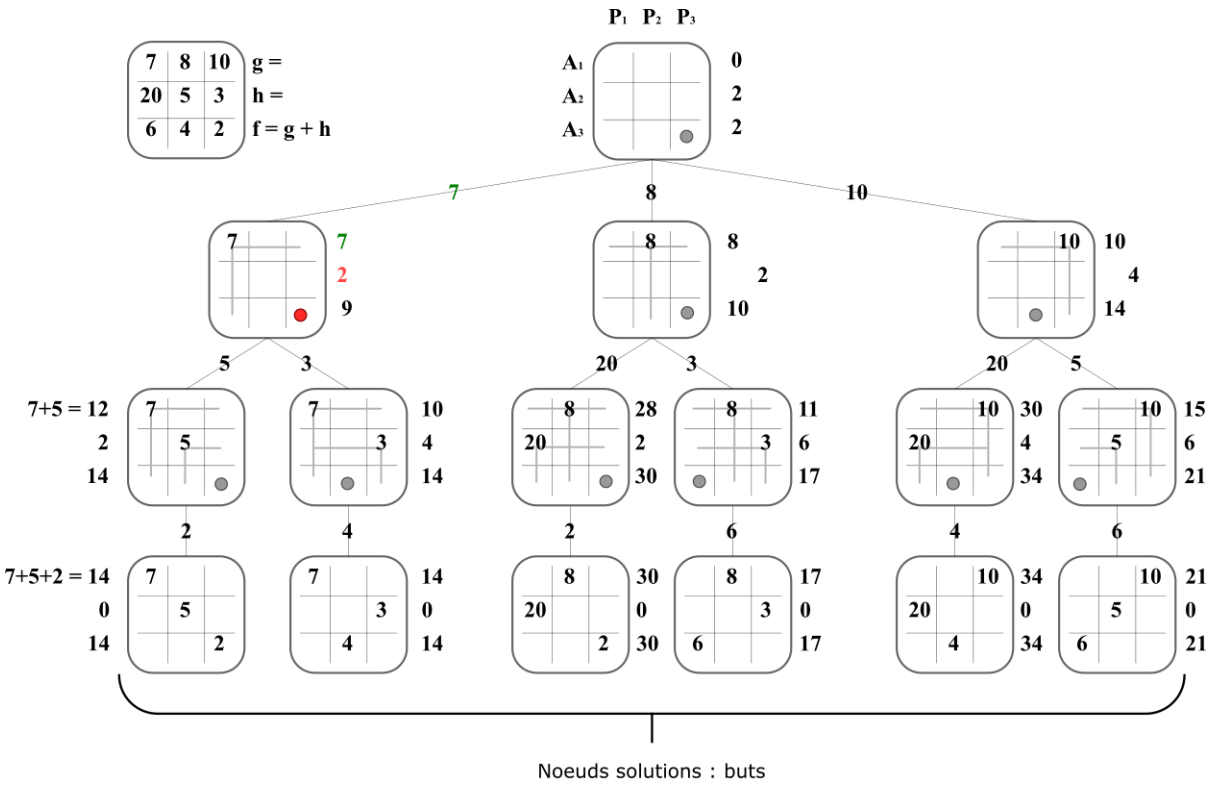
Heuristique $h_2(n)$: coût minimum

$$h_2(n) = min\ des\ coûts\ restants$$

Principe :
L'estimation h(n) d'un noeud à un but correspond au minimum des coûts restants[*1*].
1 : les coûts restants correspondent aux coûts associés aux affectations non réalisées.

En utilisant le coût minimum on s'assure de conserver la condition suivante $h(n) \leq h(n^*)$ tout en améliorant notre estimation en comparaison des heuristiques $h_0(n)$ et $h_1(n)$. En effet, on ne peut surestimer notre évaluation puisqu'on choisi toujours le minimum.

Illustration de l'algorithme :



Heuristique : minimum des coûts restants

N.B. On est passé d'un problème de taille exponentielle à linéaire.

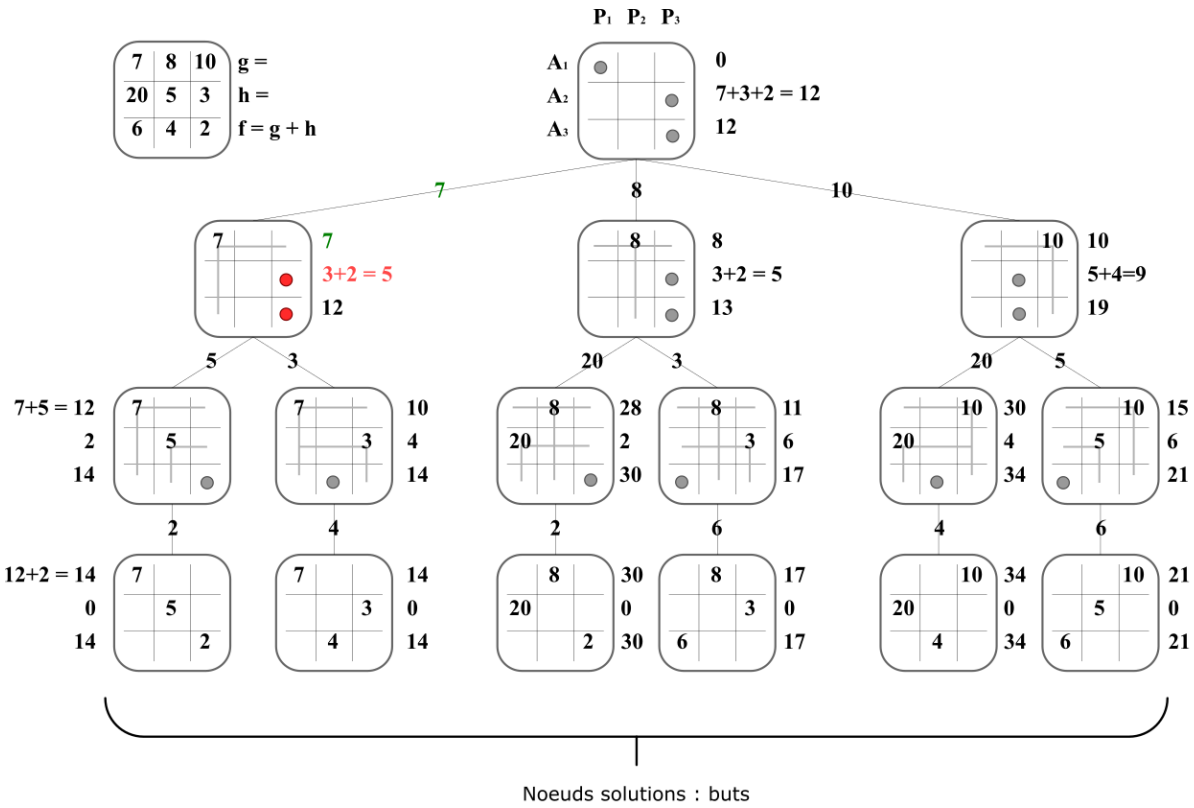
Heuristique $h_3(n)$: somme des coûts minimum restants (par ligne)

$$h_3(n) = somme\ des\ coûts\ minimum\ restants\ par\ ligne$$

Principe :
L'estimation h(n) d'un noeud à un but correspond à la somme des coûts minimums restants[*1*] par ligne[*2*].
1 : les coûts restants correspondent aux coûts associés aux affectations non réalisées.
2 : par ligne puisqu'on somme les coûts minimums pour chaque ligne de la matrice des coûts.

On notera qu'en réalisant cette somme on ne se soucie pas des contraintes d'unicité[*3*] d'affectation.
3 : si l'on ne s'autorise pas à prendre en compte les coûts des affectations réalisées, on peut dans cette somme utiliser des coûts affectant deux agents au même poste (lignes différentes mais même colonne).

Illustration de l'algorithme :



Heuristique : somme des coûts minimum restants par ligne

Heuristique $h_4(n)$: somme des coûts minimum restants (par colonne)

$$h_4(n) = somme\ des\ coûts\ minimum\ restants\ par\ colonne$$

Principe :
Le principe est le même que pour l'heuristique $h_3(n)$ à la différence que la somme se fait par colonne.

Heuristique $h_5(n)$: meilleure estimation de la somme des coûts minimum (lignes et colonnes)

$$h_5(n) = max(h_3(n),\ h_4(n))$$

Principe :
L'estimation $h(n)$ d'un noeud à un but correspond au maximum des deux heuristiques $h_3(n)$ et $h_4(n)$.

On a :

$$\forall n, h_3(n) \leq h(n)^*$$
$$\forall n, h_4(n) \leq h(n)^*$$

Donc :

$$max(h_3(n),\ h_4(n)) \leq h(n)^*$$
$$h_5(n) \leq h(n)^*$$

On a amélioré notre estimation locale en prenant la valeur la plus élevée (donc plus proche du coût réel) entre les heuristiques $h_3(n)$ (par ligne) et $h_4(n)$ (par colonne).

Comparaison des heuristiques

$$h_0(n) \leq h_1(n) \leq h_2(n) \leq h_3(n) \leq h_4(n) \leq h_5(n) \leq h^*(n)$$

L'heuristique nulle $h_0(n)$ et $h_1(n)$ qui utilise le nombre d'affectations restantes sont les moins pertinentes :

- $h_0(n)$ est une procédure aveugle et n'exploite aucune pseudo intelligence dans la recherche de la meilleure solution ;
- $h_1(n)$ a pour défaut de comparer des entiers à des coûts qui peuvent être flottants. Il faudrait normalisés les valeurs ce qui n'est pas pertinent dans des applications manipulant un grand nombre de données ou pour lesquelles la rapiditié de réponse prime sur l'exactitude de la solution.

Un cap important est franchi en terme de performances avec les heuristiques $h_2(n)$, $h_3(n)$ et $h_4(n)$:

- on passe d'un problème de taille exponentielle à linéaire. De plus, on exploite les données fournies avec des estimations très pertinentes.

La dernière heuristiques $h_5(n)$ est particulière puisqu'elle utilise deux heuristiques afin d'affiner l'estimation :

- cette heuristique à l'avantage d'apporter une précision supplémentaire à chaque étape de l'algorithme et donc peut réduire sensiblement des recherches inutiles et arriver à la solution plus rapidement. Toutefois elle met en oeuvre deux heuristiques et multiplie par deux le temps de calcul localement. Une analyse des performance nous dira dans quelles situations son utilisation est recommandée.

Programme

Dans cette partie, nous allons implémenter l'algorithme A-Star précédemment décrit pour résoudre le problème de l'affectation proposé.

Génération des matrices et variables de test

Dans un premier temps, importons les bibliothèques qui seront nécessaires à l'exécution du programme :

In [4]:

```
# On fait les imports nécessaires
import numpy as np # librairie mathématique permettant de manipuler des vecteur
s, matrices et polynômes
import random # générateur de nombre pseudo-aléatoires
import math # fonctions mathématiques définies par le standard C
import pprint # rendu «pretty-print» des structures de données
import time # fonctions relatives au temps, pour évaluer la durée d'un programme
 par exemple
import pandas as pd # manipuler facilement des données à analyser
from IPython.core.display import display # rendus visuels améliorés
```

Pour tester le bon fonctionnement des heuristiques, on prendra pour matrice de test la matrice suivante :

In [5]:

```
m_test = [[7,8,10],
          [20,5,3],
          [6,4,2]]

print("Matrice de test :")
df_m_test = pd.DataFrame(m_test).rename(index={0:'agent n°0', 1:'agent n°1', 2:'a
gent n°2'}) \
                                .rename(columns={0:'job n°0', 1:'job n°1', 2:'job
 n°2'})
display(df_m_test)
```

```
Matrice de test :
```

	job n°0	job n°1	job n°2
agent n°0	7	8	10
agent n°1	20	5	3
agent n°2	6	4	2

Pour chaque heuristique, on vérifiera la cohérence des résultats obtenus avec le programme et en exécutant l'algorithme "à la main".

Nous lancerons également l'algorithme sur plusieurs matrices de coûts générées aléatoirement.
La fonction suivante, random_matrix, nous permettra de retourner une matrice ayant les caractéristiques saisies en paramètre par l'utilisateur.

In [6]:

```
def random_matrix(type, size, bottom_range, top_range):
    """
    Cette fonction crée une matrice carrée ayant les propriétés passées en param
ètres.

    Parametres
    ----------
    type : string
        'float' ou 'int', type des valeurs contenues dans la matrice
    size : int
        dimension de la matrice
    bottom_range : int
        borne inférieure de l'intervalle des valeurs
    top_range : int
        borne supérieure de l'intervalle des valeurs

    Retourne
    --------
    numpy.ndarray
        une matrice carrée.
    """
    if type == 'int':
        return np.random.randint(bottom_range, top_range, (size, size))
    elif type == 'float':
        return np.random.uniform(bottom_range, top_range,(size, size))
```

Créons 2 matrices aléatoires pour tester notre fonction :

In [7]:

```
# Création d'une matrice de flottants, de taille 3x3 et ayant des valeurs compri
ses entre 0 et 10 :
ex_float_3x3 = pd.DataFrame(random_matrix('float', 3, 0,10))
display(ex_float_3x3)

# Création d'une matrice d'entiers, de taille 10x10 et ayant des valeurs compris
es entre 0 et 100 :
ex_int_10x10 = pd.DataFrame(random_matrix('int', 10, 0,100))
display(ex_int_10x10)
```

	0	1	2
0	8.478732	3.175644	0.26778
1	2.375244	0.919639	0.64118
2	7.512194	2.514895	3.56185

	0	1	2	3	4	5	6	7	8	9
0	75	68	44	5	73	84	50	99	80	11
1	79	43	40	30	91	61	3	27	48	93
2	84	50	75	83	23	57	67	70	44	21
3	71	62	76	91	49	22	13	51	13	27
4	1	66	80	89	42	93	45	71	16	51
5	87	32	79	33	72	2	22	80	55	15
6	70	0	43	50	44	4	77	10	35	47
7	60	83	16	13	1	60	87	49	43	57
8	95	54	35	56	53	38	66	17	33	51
9	60	26	75	63	61	12	72	46	96	12

Création de la classe Node

On crée une classe Node pour modéliser un sommet du graphe de résolution de problème. Un objet de cette classe possède comme attribut la liste des affectations partielles, l'estimation g et la valeur de la fonction d'évaluation f pour ce noeud.

In [8]:

```
class Node:
    """
    Un objet de la classe Node modélise un sommet du graphe de résolution de pro
blème.

    Paramètres
    ----------
    assignment_list : list
        Affectation partielle : liste contenant les affectations déjà effectuée
s.
    estim_h : float ou int
        Estimation de la quantité h ; par défaut, ce paramètre vaut l'infini.
    estim_f : float ou int
        Estimation de la fonction d'évaluation f ; par défaut, ce paramètre vaut
 l'infini.
        f(n) = g(n) + h(n), avec h l'heuristique choisie.

    Retourne
    --------
    Un nouveau sommet/noeud du graphe de résolution de problème

    """

    def __init__(self, assignment_list, estim_g = math.inf, estim_f = math.inf):
        """
        Constructeur de la classe Node.

        """
        self.assignment_list = assignment_list  # affectation partielle : liste
 des affectations pour n_aff agents
        self.estim_g = estim_g                  # estimation de la quantité g
        self.estim_f = estim_f                  # estimation de la fonction d'év
aluation f
```

Par exemple, pour le noeud :

In [9]:

```
ex_Node = Node([[1,2],[3,4]])
```

La liste des affectations partielles est :

- agent 1 affecté au poste 2 ;
- agent 3 affecté au poste 4.

Implémentation des heuristiques :

Note : Pour chaque heuristique $h_i(n)$, on ajoute un paramètre de test test_hi qui vaut False par défaut. Lorsque cette variable vaut True, on imprime les affichages de test permettant de vérifier le fonctionnement de la fonction heuristique choisie.

Implémentation de l'heuristique $h_1(n)$: nombre d'affectations restantes

Le nombre d'affectations restantes se calcule en soustrayant le nombre d'agents déjà affectés au le nombre total d'agents à affecter.

- Le nombre total d'agents à affecter est donné par la taille de la matrice des coûts ;
- Le nombre d'agents affectés est donné par la taille de la liste d'affectation partielle du noeud (node.assignment_list).

In [10]:

```
def nb_assignments_left(node,cost_matrix, test_h1 = False):
    """
    Cette fonction calcule l'heuristique estim_h d'un sommet en respectant la règle suivante :
    estim_h = nombre d'affectations restantes.

    Paramètres
    ----------
    node : node
        Noeud pour lequel on veut calculer l'heuristique.
    cost_matrix : list
        Matrice des coûts d'affectation.

    Retourne
    ----------
    estim_h : float ou int
        L'heuristique calculée pour le noeud passé en paramètre.

    """
    estim_h = (len(cost_matrix)-len(node.assignment_list))
    if test_h1 : print(">> Final heuristic :", len(cost_matrix) ,"-", len(node.assignment_list), "=", estim_h)
    return estim_h
```

Test
Calcul d'heuristique pour des noeuds particuliers, avec la matrice de test:

In [11]:

```
print("Noeud 1 :")
nb_assignments_left(Node([[0,1]]), m_test, True)

print("\nNoeud 2 (noeud final):")
nb_assignments_left(Node([[0,1],[1,0],[2,2]]), m_test, True)
```

```
Noeud 1 :
>> Final heuristic : 3 - 1 = 2

Noeud 2 (noeud final):
>> Final heuristic : 3 - 3 = 0
```

Out[11]:

```
0
```

Implémentation de l'heuristique $h_2(n)$: coût minimum

La fonction suivante ne prend pas directement en paramètre le noeud étudié, mais deux listes que l'on a généré grâce aux informations contenues dans la liste des affectations du noeud :

- la liste des numéros des agents déjà affectés ;
- la liste des numéros des postes déjà affectés.

Pour obtenir le minimum des coûts restants, on va travailler sur les élements de la matrice des coûts pour lesquels l'agent et le poste correspondants n'appartiennent pas aux 2 listes passées en paramètre. Parmi les élements restants, on retourne la valeur minimale.

Cependant, on ne pas calculer un minimum lorsque la matrice des coûts restants est vide : cela correspond au cas où le noeud étudié est un sommet but. Or, une des caractéristiques d'un sommet but est que la taille de sa liste des affectations est égale à la taille de la matrice des coûts : en effet, tous les agents et postes ont été affectés. Il faut donc vérifier que cette condition n'est pas satisfaite avant de calculer le minimum. Dans le cas où le noeud étudié est un sommet but, l'heuristique (coût d'un chemin optimal pour atteindre le noeud final) vaut donc zéro.

In [12]:

```
def min_costs_left(cost_matrix, assigned_agents, assigned_jobs, test_h2 = False
):
    """
    Cette fonction calcule l'heuristique estim_h d'un sommet en respectant la rè
gle suivante :
    estim_h = minimum des coûts restants

    Paramètres
    ----------
    cost_matrix : list
        Matrice des coûts d'affectation.
    assigned_agents : list
        Liste contenant les numéros des agents déjà affectés.
    assigned jobs : list
        Liste contenant les numéros des postes déjà affectés.

    Retourne
    ----------
    estim_h : float ou int
        L'heuristique calculée.

    """

    N = len(cost_matrix)


    if len(assigned_agents) != N and len(assigned_jobs) != N :
        min_cost = math.inf # Coût minimal pour la ligne n°agent, initialisé à +i
nfini.
        for agent in range(N): # pour tous les agents
            if agent not in assigned_agents:
                for job in range(N):
                    if job not in assigned_jobs:
                        min_cost = min(min_cost, cost_matrix[agent][job]) # met
 à jour le coût minimal par ligne
        estim_h = min_cost # ajoute ce coût à la quantité estim_h
    else:
        estim_h = 0

    if test_h2 :
        print(">> Final heuristic :", estim_h)

    return estim_h
```

Test

Calcul d'heuristique pour des noeuds particuliers, avec la matrice de test:

In [13]:

```
display(pd.DataFrame(m_test))
print("Noeud 1 :")
min_costs_left(m_test,[0],[1], True)

print("\nNoeud 2 (noeud final):")
min_costs_left(m_test,[0,1,2],[0,1,2], True)
```

	0	1	2
0	7	8	10
1	20	5	3
2	6	4	2

```
Noeud 1 :
>> Final heuristic : 2

Noeud 2 (noeud final):
>> Final heuristic : 0
```

Out[13]:

```
0
```

Heuristique $h_3(n)$: somme des coûts minimum (lignes)

La fonction min_by_row, qui permet de calculer la somme des coûts minimum par ligne, parcourt la matrice des coûts restants "de gauche à droite" et calcule pour chaque ligne le coût minimum. Ce minimum est incrémenté à une variable qui, après avoir parcouru toute la matrice, constitue l'heuristique du noeud étudié.

In [14]:

```
def min_by_row(cost_matrix, assigned_agents, assigned_jobs, test_h3 = False):
    """
    Cette fonction calcule l'heuristique estim_h d'un sommet en respectant la règle suivante :
    estim_h = coût minimum par ligne/agent.


    Paramètres
    ----------
    cost_matrix : list
        Matrice des coûts d'affectation.
    assigned_agents : list
        Liste contenant les numéros des agents déjà affectés.
    assigned jobs : list
        Liste contenant les numéros des postes déjà affectés.

    Retourne
    ----------
    estim_h : float ou int
        L'heuristique calculée.

    """
    N = len(cost_matrix)
    estim_h = 0

    # On calcule le minimum pour TOUS les agents n'ayant pas encore été affectés car h(n) distance estimée entre le noeud
    # intermédiaire étudié et un sommet but.

    for agent in range(N): # pour tous les agents
        if agent not in assigned_agents:
            min_cost = math.inf # Coût minimal pour la ligne n°agent, initialisé à +infini.
            for job in range(N):
                if job not in assigned_jobs:
                    min_cost = min(min_cost, cost_matrix[agent][job]) # met à jour le coût minimal par ligne
            estim_h += min_cost # ajoute ce coût à la quantité estim_h
            if test_h3 : print("Assignment for agent n°", agent, ": minimum cost = ", min_cost)
        else :
            if test_h3 : print("Agent n°", agent, " has already been assigned to a job.")
    if test_h3 : print("\n>> Final heuristic :", estim_h)
    return estim_h
```

Test

Calcul d'heuristique pour des noeuds particuliers, avec la matrice de test :

In [15]:

```
display(pd.DataFrame(m_test))
print("Noeud 1 :")
min_by_row(m_test,[0],[1], True)

print("\nNoeud 2 (noeud final):")
min_by_row(m_test,[0,1,2],[0,1,2], True)
```

	0	1	2
0	7	8	10
1	20	5	3
2	6	4	2

```
Noeud 1 :
Agent n° 0  has already been assigned to a job.
Assignment for agent n° 1 : minimum cost =  3
Assignment for agent n° 2 : minimum cost =  2

>> Final heuristic : 5

Noeud 2 (noeud final):
Agent n° 0  has already been assigned to a job.
Agent n° 1  has already been assigned to a job.
Agent n° 2  has already been assigned to a job.

>> Final heuristic : 0
```

Out[15]:

0

Heuristique $h_4(n)$: somme des coûts minimum (colonnes)

La fonction permettant de calculer la somme des coûts minimum par colonne suit la même logique d'implémentation que son équivalent par ligne. Cependant, on parcoura ici la matrice des coûts restants non pas de gauche à droite mais de bas en haut, de telle sorte à calculer le minimum pour chaque colonne.

In [16]:

```
def min_by_column(cost_matrix, assigned_agents, assigned_jobs, test_h4 = False):

    """
    Cette fonction calcule l'heuristique estim_h d'un sommet en respectant la rè
gle suivante :
    estim_h = coût minimum par colonne/poste.

    Paramètres
    ----------
    cost_matrix : list
        Matrice des coûts d'affectation.
    assigned_agents : list
        Liste contenant les numéros des agents déjà affectés.
    assigned jobs : list
        Liste contenant les numéros des postes déjà affectés.

    Retourne
    ----------
    estim_h : float ou int
        L'heuristique calculée.

    """
    N = len(cost_matrix)
    estim_h = 0

    # On calcule le minimum pour TOUS les agents n'ayant pas encore été affectés
 car h(n) distance estimée entre le noeud
    # intermédiaire étudié et un sommet but.

    for job in range(N): # pour tous les postes
        if job not in assigned_jobs:
            min_cost = math.inf # Coût minimal pour la colonne n°job, initialisé
 à +infini.
            for agent in range(N):
                if agent not in assigned_agents:
                    min_cost = min(min_cost, cost_matrix[agent][job]) # met à jo
ur le coût minimal par colonne
            estim_h += min_cost # ajoute ce coût à la quantité estim_h
            if test_h4 : print("Assignment for job n°", job, ": minimum cost = ",
 min_cost)
        else :
            if test_h4 : print("Job n°", job, " has already been assigned to an a
gent.")
    if test_h4 : print("\n>> Final heuristic :", estim_h)
    return estim_h
```

Test

Calcul d'heuristique pour des noeuds particuliers, avec la matrice de test :

In [17]:

```
display(pd.DataFrame(m_test))
print("Noeud 1 :")
min_by_column(m_test,[0],[1], True)

print("\nNoeud 2 (noeud final):")
min_by_column(m_test,[0,1,2],[0,1,2], True)
```

	0	1	2
0	7	8	10
1	20	5	3
2	6	4	2

```
Noeud 1 :
Assignment for job n° 0 : minimum cost =  6
Job n° 1  has already been assigned to an agent.
Assignment for job n° 2 : minimum cost =  2

>> Final heuristic : 8

Noeud 2 (noeud final):
Job n° 0  has already been assigned to an agent.
Job n° 1  has already been assigned to an agent.
Job n° 2  has already been assigned to an agent.

>> Final heuristic : 0
```

Out[17]:

```
0
```

On constate bien que les valeurs des heuristiques obtenues pour les heuristiques h3 et h4 peuvent être différentes pour un même noeud.

Heuristique $h_5(n)$: meilleure estimation de la somme des coûts minimum (lignes et colonnes)

La fonction max_line_column, qui retourne la meilleure estimation de la somme des coûts minimum, appelle simplement les fonctions min_by_line() et min_by_column puis utilise la fonction max() pour estimer le maximum entre les deux heuristiques calculées.

Remarque :
Au niveau de l'implémentation, nous n'avons pas besoin de passer en paramètre une copie des listes assigned_agents et assigned_jobs car les fonctions min_by_line() et min_by_column() ne modifient pas ces listes. Sinon, il aurait fallu passer des copies de ces listes (avec "list(assigned_agents)" et "list(assigned_jobs)") à min_by_line, de telle sorte à laisser ces deux listes intactes lors de l'appel de min_by_column.

In [18]:

```
def max_line_column(cost_matrix, assigned_agents, assigned_jobs, test_h5 = False
):
    """
    Cette fonction calcule l'heuristique estim_h d'un sommet en respectant la rè
gle suivante :
    estim_h = maximum entre le coût minimum par agent/ligne (h3) et le coût mini
mum par poste/colonne (h4).
    On prend le maximum pour se rapprocher au mieux de h*(n), l'heuristique opti
male.
    En effet, on a : h3(n) < h4(n) < h*(n) ou h4(n) < h3(n) < h*(n)

    Paramètres
    ----------
    cost_matrix : list
        Matrice des coûts d'affectation.
    assigned_agents : list
        Liste contenant les numéros des agents déjà affectés.
    assigned_jobs : list
        Liste contenant les numéros des postes déjà affectés.

    Retourne
    ----------
    estim_h : float ou int
        L'heuristique calculée.

    """
    if test_h5:
        print("\nWe check that lists of assigned agents and assigned jobs are no
t modified between the 2 heuristics calls :\n")
        print("Beginning :\n  assigned_agents :", assigned_agents, "\n  assigned
_jobs   :",assigned_jobs, '\n')

    estim_h_line   = min_by_row(cost_matrix, assigned_agents, assigned_jobs)   #
 exécution de l'heuristique minimum par ligne
    if test_h5: print("After calling min_by_row :\n  assigned_agents :", assigne
d_agents, "\n  assigned_jobs   :",assigned_jobs, '\n')

    estim_h_column = min_by_column(cost_matrix, assigned_agents, assigned_jobs)
# exécution de l'heuristique minimum par colonne
    if test_h5: print("After calling min_by_column :\n  assigned_agents :", assi
gned_agents, "\n  assigned_jobs   :",assigned_jobs, '\n')

    estim_h = max(estim_h_line, estim_h_column) # On prend le maximum entre les
 2 heuristiques calculées.
    if test_h5:
        print("----------------------------------------\n")
        print("Value of heuristic :\n  Minimum by line   :", estim_h_line, "\n
 Minimum by column :", estim_h_column)
        print("\nHeuristic chosen = Maximum(", estim_h_line, ",", estim_h_column
,") = ", estim_h)
        print("\n>> Final heuristic :", estim_h)

    return estim_h
```

Test

Calcul d'heuristique pour des noeuds particuliers, avec la matrice de test :

In [19]:

```
display(pd.DataFrame(m_test))
print("Noeud 1 :")
max_line_column(m_test,[0],[1], True)
print("\n==========================================================================
==\n")
print("Noeud 2 (noeud final):")
max_line_column(m_test,[0,1,2],[0,1,2], True)
```

	0	**1**	**2**
0	7	8	10
1	20	5	3
2	6	4	2

```
Noeud 1 :

We check that lists of assigned agents and assigned jobs are not mod
ified between the 2 heuristics calls :

Beginning :
  assigned_agents : [0]
  assigned_jobs   : [1]

After calling min_by_row :
  assigned_agents : [0]
  assigned_jobs   : [1]

After calling min_by_column :
  assigned_agents : [0]
  assigned_jobs   : [1]

----------------------------------------

Value of heuristic :
  Minimum by line   : 5
  Minimum by column : 8

Heuristic chosen = Maximum( 5 , 8 ) =  8

>> Final heuristic : 8

================================================================
=====

Noeud 2 (noeud final):

We check that lists of assigned agents and assigned jobs are not mod
ified between the 2 heuristics calls :

Beginning :
  assigned_agents : [0, 1, 2]
  assigned_jobs   : [0, 1, 2]

After calling min_by_row :
  assigned_agents : [0, 1, 2]
  assigned_jobs   : [0, 1, 2]

After calling min_by_column :
  assigned_agents : [0, 1, 2]
  assigned_jobs   : [0, 1, 2]

----------------------------------------

Value of heuristic :
  Minimum by line   : 0
  Minimum by column : 0

Heuristic chosen = Maximum( 0 , 0 ) =  0

>> Final heuristic : 0


Out[19]:

0
```

Calcul de la fonction d'estimation f :

Fonction d'estimation de la quantité h (heuristique)

On rappelle que pour un noeud n :

$$f(n) = g(n) + h(n)$$

In [20]:

```
def compute_h(node, cost_matrix, heuristic, assigned_agents, assigned_jobs, test
_cpt_h = False):
    """
    La fonction compute_h calcule l'heuristique estim_h pour le noeud passé en p
aramètre.
    L'heuristique correspond au coût d'un chemin optimal, du noeud passé en para
mètre à un noeud final.

    Paramètres
    ----------
    node : node
        Noeud pour lequel on veut calculer l'heuristique.
    cost_matrix : list
        Matrice des coûts d'affectation.
    heuristic : string
        L'heuristique choisie.
    assigned_agents : list
        Liste contenant les numéros des agents déjà affectés pour le noeud passé
 en paramètre.
    assigned jobs : list
        Liste contenant les numéros des postes déjà affectés pour le noeud passé
 en paramètre.


    Retourne
    ----------
    estim_h : int ou float
        L'heuristique du noeud passé en paramètre.

    """

    # Calcul de h(n), depend de l'heuristique :

    estim_h = math.inf          # initialisation de l'heuristique

    if heuristic == "null":      # Heuristique nulle
        estim_h = 0

    elif heuristic == "nb_assignments_left":    # Nombre d'affectations restante
s
        estim_h = nb_assignments_left(node,cost_matrix,test_cpt_h)

    elif heuristic == "min_by_row":             # Minimum des lignes restantes
        estim_h = min_by_row(cost_matrix, assigned_agents, assigned_jobs,test_cp
t_h)

    elif heuristic == "min_by_column":          # Minimum des colonnes restantes
        estim_h = min_by_column(cost_matrix, assigned_agents, assigned_jobs,test
_cpt_h)

    elif heuristic == "max_line_column":        # Maximum de l'heuristique obten
ue entre  h3 et h4
        estim_h = max_line_column(cost_matrix, assigned_agents, assigned_jobs,te
st_cpt_h)

    elif heuristic == "min_costs_left":         # Coût minimum parmi les coûts r
estants
        estim_h = min_costs_left(cost_matrix, assigned_agents, assigned_jobs,tes
t_cpt_h)
```

```
    elif heuristic == "min_costs_left2":        # Coût minimum parmi les coûts r
estants
        estim_h = min_costs_left2(cost_matrix, assigned_agents, assigned_jobs,te
st_cpt_h)
    else :
        print("Error : please specify a correct heuristic name.")

    return estim_h
```

Fonction d'estimation de la quantité g

In [21]:

```
def compute_g(node, cost_matrix):
    """
    La fonction compute_g calcule la quantité g pour le noeud passé en paramètr
e.
    L'heuristique correspond au coût d'un chemin optimal, du noeud de départ au
 noeud passé en paramètre.

    Paramètres
    ----------
     node : node
        Noeud pour lequel on veut calculer l'heuristique.
     cost_matrix : list
        Matrice des coûts d'affectation.

    Retourne
    ----------
    assigned_agents : list
        Liste contenant les numéros des agents déjà affectés pour le noeud passé
 en paramètre.
    assigned_jobs : list
        Liste contenant les numéros des postes déjà affectés pour le noeud passé
 en paramètre.

    """

    assigned_agents = [] # listes des agents déjà affectés
    assigned_jobs   = [] # listes des postes déjà affectés
    node.estim_g = 0     # initialisation de la quantité g à 0


    # Calcul de g(n) : la quantité h correspond au coût du chemin pour aller du
 noeud initial (pour lequel assignment_list
    # est vide) jusqu'au noeud passé en paramètre de la fonction compute_h.

    # On remplit les listes assigned_agents et assigned_jobs et calcule la quant
ité g.
    for assignment in node.assignment_list:
        agent = assignment[0]
        job   = assignment[1]
        assigned_agents.append(agent)
        assigned_jobs.append(job)

        node.estim_g += cost_matrix[agent][job] # Pour chaque affectation conten
ue dans assignment_list, on ajoute son coût à estim_g.

    return assigned_agents, assigned_jobs
```

Fonction d'estimation de la fonction d'évaluation

In [22]:

```
def compute_f(node, cost_matrix, heuristic, test_cpt_f = False):
    """
    La fonction compute_f calcule la fonction d'évaluation f pour le noeud passé
 en paramètre.
    Cette fonction estime le coût d'un chemin optimal du noeud de départ à un so
mmet but passant par node (noeud passé en paramètre)
    On a f(node) = g(node) + h(node)

    Paramètres
    ----------
    node : node
        Noeud pour lequel on veut calculer l'heuristique.
    cost_matrix : list
        Matrice des coûts d'affectation.
    heuristic : string
        L'heuristique choisie.

    """
    assigned_agents, assigned_jobs = compute_g(node, cost_matrix) # calcul de la
 quantité g
    estim_h = compute_h(node, cost_matrix, heuristic, assigned_agents, assigned_
jobs,test_cpt_f) # calcul de la quantité h

    node.estim_f = node.estim_g + estim_h  # f(node) = g(node) + h(node)
    return
```

Fonction *extract_first_open* qui permet de trouver le noeud minimisant la fonction d'évaluation *estim_f*.

In [23]:

```
def extract_first_open(opened_list):
    """
    La fonction extract_first_open sélectionne dans la liste des ouverts passée
 en paramètre (opened_list)
    un noeud qui minimise la fonction d'évaluation estim_f. Ce noeud est ensuite
 retiré de la liste puis
    retourné par la fonction.

    Paramètres
    ----------
    opened_list : list
        La liste des noeuds ouverts.

    Retourne
    ----------
    optimal_node : node
        Le noeud optimal qui minimise la quantité estim_f.

    """

    # dictionnaire associant le noeud (clé) et sa quantité estim_f (valeur)
    dictionary = {node:node.estim_f for node in opened_list}

    optimal_node = min(dictionary, key = dictionary.get) # on récupère le noeud
 pour lequel estim_f est minimal
    opened_list.remove(optimal_node)

    return optimal_node
```

Fonction *develop_node* qui comme son nom l'indique permet de développer un noeud passé en paramètre, c'est-à-dire retourner la liste de ses noeuds successeurs.

In [24]:

```
def develop_node(node, cost_matrix):
    """
    La fonction develop_node construit la liste des noeuds successeurs du noeud
 passé en paramètre dans
    le graphe de résolution de problème.

    Paramètres
    ----------
    node : node
        Le noeud pour lequel on veut retourner la liste des noeuds successeurs.
    cost_matrix : list
        Matrice des coûts d'affectation.

    Retourne
    ----------
    successors : list
        La liste contenant les noeuds successeurs du noeud passé en paramètre.

    """

    # On mémorise les postes qui ont déjà été affectés
    assigned_jobs = [assignment[1] for assignment in node.assignment_list]

    N = len(cost_matrix)
    agent_to_assign = len(node.assignment_list) # On veut affecter un poste à
 l'agent i
    successors = [] # liste contenant les noeuds successeurs au noeud passé en p
aramètre

    for job in range(N) : # pour tous les postes
        if job not in assigned_jobs:
            new_node_assignment_list = list(node.assignment_list)  # liste des a
ffectations déjà réalisées
            new_node_assignment_list.append((agent_to_assign, job)) # ajoute aff
ectation possible
            new_node = Node(new_node_assignment_list)
            successors.append(new_node)

    return successors
```

Fonction *a_star* qui applique l'algorithme A* dans le cas du problème de l'affectation.

In [25]:

```
def a_star(cost_matrix, heuristic, test_a_star = False):
    """
    La fonction a_star exécute l'algorithme A-Star pour le problème de l'affecta
tion.

    Paramètres
    ----------
    cost_matrix : list
        Matrice des coûts d'affectation, donnée du problème.
    heuristic : string
        L'heuristique choisie

    Retourne
    ----------
    node : node
        Un sommet but avec un coût d'affectation global minimal.
        Un sommet but modélise une solution où tous les agents ont été affectés
 à un poste.
    """
    nb_iterations = 0
    N = len(cost_matrix)
    opened_list = [] # liste des noeuds pas encore étudiés et qui pourraient fai
re partie de la solution optimale
    closed_list = [] # liste des noeuds déjà étudiés et qui pourraient faire par
tie de la solution optimale

    source = Node([]) # Noeud de départ : aucun agent affecté --> liste des affe
ctations vide
    GRP = {None:source}    # GRP : le graphe de recherche un élement de cette li
ste
    opened_list.append(source)

    while (opened_list):
        node = extract_first_open(opened_list) # On réordonne la liste des ouver
ts et extrait le premier élément (celui ayant estim_f le plus faible)
        closed_list.append(node)               # On met le sommet étudié node da
ns fermé.

        if(len(node.assignment_list) != N): # Si node n'est pas un sommet but, c
ontinuer
        #Si le nombre d'affectations est égal à la taille de la matrice, alors t
ous les agents ont été affectés --> le noeud est un sommet but.
            successors = develop_node(node, cost_matrix) # retourne liste des no
euds successeurs

            for successor_node in successors:
                compute_f(successor_node, cost_matrix,heuristic,test_a_star) # c
alcule heuristique des nouveaux noeuds
                opened_list.append(successor_node) # met les nouveaux noeuds dan
s la liste des noeuds qui peuvent être étudiés

            GRP[node] = successors # mémorise la liste successors comme successe
urs du noeud dans le GRP
            nb_iterations += 1
        else:
            return node, GRP, nb_iterations
    else:
        return "pas de solution"
```

__main__ : test de l'algorithme A*

Dans cette partie nous facilitons les tests de l'algorithmes A*.

On définit des matrices de coûts :

In [26]:

```
m_test1 = [[7,8,10],
           [20,5,3],
           [6,4,2]]

m_test2 = [[1, 3, 5, 2],
           [2, 4, 3, 10],
           [7, 1, 8, 5],
           [8, 8, 10, 1]]

m_test3 =  [[5,10,15,1,14],
            [3,15,10,18,32],
            [2,5 ,8,7,3 ],
            [9,10,13,3,2 ],
            [4,5,7,17,9]]

m_random = random_matrix('int', 3, 0, 10)
```

Paramètres à modifier :

In [27]:

```
# choix de la matrice
matrix = m_test3 # à modifier

# choix de l'heuristique
h0 = "null"
h1 = "nb_assignments_left"
h2 = "min_costs_left"
h2p = "min_costs_left2"
h3 = "min_by_row"
h4 = "min_by_column"
h5 = "max_line_column"
heuristic = h1 # à modifier

#----------------------------
start_clock = time.clock()
final_node, GRP,nb_iterations = a_star(matrix, heuristic)
end_clock = time.clock()

def color_matrix(matrix, assignment_list):
    df = pd.DataFrame(matrix)
    df_path = style_matrix(len(matrix), assignment_list)
    return df.style.apply(lambda tab: df_path, axis=None)
def style_matrix(n, assignment_list):
    style = [['']*n for i in range(n)]
    for i in range(0, len(assignment_list)):
        style[i][assignment_list[i][1]] = 'background-color: #A7E52C'
    return pd.DataFrame(style)

print('Matrice de coûts :')
display(pd.DataFrame(matrix))
print('Heuristique choisie :', heuristic, '\n')
print('Temps d\'execution :', end_clock-start_clock, 'seconde(s)', '\n')
print("Chemin optimal :")
print(final_node.assignment_list, '\n')
df_matrix_colored = color_matrix(matrix, final_node.assignment_list)
display(df_matrix_colored)
print('Score total :', final_node.estim_f)
```

```
Matrice de coûts :
```

	0	1	2	3	4
0	5	10	15	1	14
1	3	15	10	18	32
2	2	5	8	7	3
3	9	10	13	3	2
4	4	5	7	17	9

```
Heuristique choisie : nb_assignments_left

Temps d'execution : 0.00046500000000015973 seconde(s)

Chemin optimal :
[(0, 3), (1, 0), (2, 1), (3, 4), (4, 2)]
```

	0	1	2	3	4
0	5	10	15	1	14
1	3	15	10	18	32
2	2	5	8	7	3
3	9	10	13	3	2
4	4	5	7	17	9

```
Score total : 18
```

Performances

Dans cette partie nous testons les performances de chaque heuristique.
On définit d'abord quelques fonctions utiles :

Fonction *gen_test_matrices* qui retourne une liste de listes de matrices.

In [28]:

```
def gen_test_matrices(size_matrices, number_matrices, bottom_range, top_range):
    """
        Génère une liste contenant des listes de matrices.
    """
    matrices = [] # contient la liste des listes des matrices de même taille
    for size in size_matrices: # pour chaque tailles de matrice
        sub = []
        for foo in range(0, number_matrices): # on génère autant de matrices de
 même taille qu'indiqué
            sub.append(random_matrix('int', size, bottom_range, top_range))
        matrices.append(sub)
    return matrices
```

Fonction *sort_by_size* qui ordonne les résultats par matrice pour une meilleure visualisation.

In [29]:

```
def sort_by_size(matrices_size, results):
    """
        Ordonne les résultats par taille de matrice.
    """
    matrices_size = sorted(matrices_size)

    results2 = []
    for size in matrices_size:
        for elt in results:
            if elt[1] == size:
                results2.append(elt)

    df2 = pd.DataFrame(results2)
    df2 = df2.rename(columns={0: "heuristic",
                              1: "matrix size",
                              2: "time(s)",
                              3: "Dev. nodes",
                              4: 'Dev. nodes/All nodes'})
    return df2
```

Fonction *test_performance* qui retourne un tableau contenant le résultat de l'analyse des heuristiques.

In [30]:

```
def test_performance(matrices, heuristics):
    """
        Soit N la taille d'une matrice.

        Pour chaque heuristique:
            Pour plusieurs matrices de taille N:
                Executer l'heuristique
                Mémoriser critères :
                 - le temps d'exécution
                 - le nombre de sommets ouverts (= developpés)
                 - nb exécution/intération
                 - Espace mémoire utilisé
            -> faire moyenne de ces critères + pire cas et meilleur cas d'exécut
ion

        évaluer "à la main" la complexité des algos
        dépend caractéristiques environnement : CPU...
    """
    results = []
    nb_digits = 6


    for h_i in heuristics :
        for sub in matrices:

            mean = math.inf # moyenne
            exec_time_list = []
            nb_developed_nodes = []
            matrix_size = len(sub[0])

            for matrix in sub:
                start_clock = time.clock()
                final_node, GRP,nb_iterations = a_star(matrix, h_i)
                end_clock = time.clock()

                exec_time_list.append(end_clock - start_clock)
                nb_developed_nodes.append(nb_iterations)

            mean_time_exec = np.mean(exec_time_list)
            mean_nb_developed_nodes = np.mean(nb_developed_nodes)
            ratio_developed_nodes = mean_nb_developed_nodes/nodeGRP(len(sub[0]))

            results.append([h_i,matrix_size,round(mean_time_exec,nb_digits), \
                            mean_nb_developed_nodes, round(ratio_developed_nodes
, nb_digits)])

    df = pd.DataFrame(results)
    df = df.rename(columns={0: "heuristic",
                            1: "matrix size",
                            2: "time(s)",
                            3: "Dev. nodes",
                            4: 'Dev. nodes/All nodes'})
    return results
```

Paramètrage de l'analyse des performances

Paramètrer ci-dessous l'analyse des performances des heuristiques :

In [31]:

```
# taille des matrices à tester
size_matrices = (3, 5, 10)
# valeur minimale des coûts
bottom_range = 0
# valeur maximale des coûts
top_range = 20
# nombre de matrices à tester pour une même taille
number_matrices = 2
# heuristiques à tester
h0 = "null"
h1 = "nb_assignments_left"
h2 = "min_costs_left"
h3 = "min_by_row"
h4 = "min_by_column"
h5 = "max_line_column"
heuristics = [h0,h1,h2,h3,h4,h5]

matrices = gen_test_matrices(size_matrices, number_matrices, bottom_range, top_r
ange)
results = test_performance(matrices, heuristics)
results_ordered = sort_by_size(size_matrices, results)
display(results_ordered)
```

	heuristic	matrix size	time(s)	Dev. nodes	Dev. nodes/All nodes
0	null	3	0.000089	8.0	0.500000
1	nb_assignments_left	3	0.002457	8.0	0.500000
2	min_costs_left	3	0.001028	5.0	0.312500
3	min_by_row	3	0.002158	3.0	0.187500
4	min_by_column	3	0.000160	4.0	0.250000
5	max_line_column	3	0.000137	3.0	0.187500
6	null	5	0.000755	40.5	0.124233
7	nb_assignments_left	5	0.000531	29.5	0.090491
8	min_costs_left	5	0.001302	30.0	0.092025
9	min_by_row	5	0.000331	7.5	0.023006
10	min_by_column	5	0.000360	9.0	0.027607
11	max_line_column	5	0.000442	6.0	0.018405
12	null	10	6.795221	3528.0	0.000358
13	nb_assignments_left	10	1.097248	1383.0	0.000140
14	min_costs_left	10	5.915047	3104.0	0.000315
15	min_by_row	10	0.037892	149.5	0.000015
16	min_by_column	10	0.031135	133.5	0.000014
17	max_line_column	10	0.025904	68.0	0.000007

Conclusion :
On constate que pour un problème de taille relativement petit, utiliser des heuristiques peu performantes n'est pas déterminant ; les temps de calculs et le nombre de noeuds developpés varie peu d'une heuristique à une autre.

Cependant, on se rend compte que pour des problèmes de taille plus grands, le temps de calcul et le nombre de noeuds developpés croit de façon très rapide pour les heuristiques les moins performantes (heuristique nulle), alors qu'il croit relativement lentement pour des heuristiques plus performantes (heuristique du maximum entre la somme des coûts minimum par ligne et par colonne).

A partir d'un certain seuil, il devient alors primordial de choisir une heuristique performante.

Bibliographie

- Principes d'intelligence artificielle, Nils J.Nilsson
- Cours et énoncé : https://perso.esiee.fr/~coupriem/PR3602/ (https://perso.esiee.fr/~coupriem/PR3602/), Michel Couprie