Heuristiques, algorithme de Dijkstra et algorithme A*

1) Algorithme de Dijkstra (il s'agit d'un algorithme optimal de type glouton)

On considère un graphe orienté $G = (S, A)$ valué **par des réels positifs**, c'est-à-dire muni d'une fonction $\omega : A \to \mathbb{R}^+$, où A est l'ensemble des arêtes du graphe G. Ainsi, on note $\omega(x, y)$ le poids de l'arête $(x, y) \in A$.
Le coût d'un chemin est par définition la somme des poids des arêtes du chemin.

On fixe un sommet initial $s \in S$ et un sommet final $t \in S$.
On cherche à calculer le coût minimal d'un chemin reliant s à t.

• A chaque étape de l'algorithme, on dispose :
- d'une liste L de sommets pour lesquels **on connaît le coût minimum d'un chemin** reliant s à x
- de la bordure B de L dans S, c'est-à-dire des sommets admettant un prédecesseur dans L mais qui ne sont pas dans L (il s'agit assez souvent de tous les sommets n'appartenant pas à L)
- d'un tableau d de sorte que
$\begin{cases} \forall x \in L,\ d[x] \text{ est le coût minimum d'un chemin reliant } s \text{ à } x \\ \forall x \in B,\ d[x] \text{ est le coût minimum d'un chemin reliant } s \text{ à } x \text{ et } \textbf{dont les sommets intermédiaires} \text{ sont dans } L \end{cases}$
L est souvent appelée "liste fermée" (*closed list*) et B est souvent appelée "liste ouverte" (*open list*).

Remarque : En fait, les sommets de B sont plus éloignés de s (en coût) que les sommets de L.

• On détermine alors le sommet de B minimisant $d[x]$, on l'ajoute à L, et on met à jour le tableau d, car il faut tenir compte des successeurs y de x : il convient d'ajouter y à B s'il n'est pas déjà et de modifier la valeur de $d[y]$ en tenant compte des chemins $s \to ... \to x \to y$, avec l'instruction $d[y] = \min(d[y], d[x] + \omega(x, y))$.

Code Python de l'algorithme :

```
# On suppose en amont le tableau d initialisé à +∞ et le tableau visite initialisé à 0.
d[s] = 0
B = FileVide() ; ajouter(s,B,d)          # B est une file de priorité selon d
L = [] ; visite[s] = 1     # la valeur 1 indique l'appartenance à B
while estNonVide(B) :
    x = defiler(B)
      # x est choisi comme l'élément de B minimisant d
      # d[x] est le coût minimum d'un chemin de s à x : on peut donc ajouter x à L
    L.append(x) ; visite[x] = 2    # la valeur 2 indique l'appartenance à L
    if x == t :  break
    # On met ensuite à jour le tableau d et la bordure B suite au passage de x de B à L
    # Les chemins considérés dans d peuvent donc passer par x (en plus des anciens sommets de L)
    for y in successeurs(x) :
        c = d[x] + w(x,y)
        if visite[y] == 1 and c < d[y] :
            d[y] = c ; modifier(B,y,d) ; pere[y] = x
```

```
            # autrement dit, si y est dans B, on prend d[y] = min(d[y], d[x] + ω(x,y))
        elif visite[y] == 0 :
            d[y] = c ; ajouter(B,y,d) ; pere[y] = x
```

A la fin de l'algorithme, en cas de succès (c'est-à-dire si x vaut t), on pourrait reconstruire le chemin à partir de la liste close. Mais il est plus efficace d'itérer sur `pere`, en remontant depuis le sommet final z.

Remarque : La liste L peut contenir des sommets n'appartenant pas au chemin final, dans le cas où on a été amené à rebrousser chemin. Par exemple, L peut être de la forme $[s, x_1, x_2, x_3, y, t]$ avec :

$$\begin{array}{lllllll} s & \xrightarrow{1} & x_1 & \xrightarrow{1} & x_2 & \xrightarrow{1} & x_3 \\ \downarrow_4 & & & & & & \\ y & \xrightarrow{1} & t & & & & \end{array}$$

Complexité : La file de priorité B selon d consiste à stocker les éléments selon la valeur de d.

Une bonne structure est le tas (ajout, modification et suppression en $O(\log n)$). Une structure plus élémentaire utilise une liste mais est de complexité plus élévée (ajout et modification en $O(n)$).

2) Algorithme A*

On souhaite déterminer un chemin allant d'une position initiale s à une position finale t.

On note $d(x)$ le coût minimal pour aller de s à x. On se donne **une heuristique** : pour tout sommet x, $h(x)$ donne **une estimation** du coût d'un chemin pour aller de x à t.

Détermination d'un chemin reliant s à t par l'algorithme A^* : à chaque étape, on choisit dans **la file de priorité** B le sommet (ou l'un de ceux) qui minimise le **coût heuristique du chemin total : le coût heuristique en x est la somme** du coût du chemin de s à x et de l'heuristique en x

$$f(x) = d(x) + h(x)$$

Dans la suite, le coût du chemin de s à x est codé par le tableau `d` et l'heuristique par le tableau `h`.

De plus, si on souhaite expliciter le chemin optimal, le plus efficace plus rapide d'utiliser un dictionnaire `pere` donnant le père de chaque sommet visité. Ainsi, à la fin, en partant de t, on peut remonter au sommet s en itérant sur le tableau `pere`.

Algorithme A en pseudo-code* PYTHON :

```
# On suppose en amont le tableau d initialisé à +∞ et les tableaux visite et f initialisés à 0.
# On suppose connue l'heuristique h de sorte que h(x)évalue δ(x,z)
01      d[s] = 0 ; f[s] = h(s)
02      B = FileVide() ; ajouter(s,B,f) # B file de priorité f
03      L = [] ; visite[s] = 1     # la valeur 1 indique l'appartenance à B
04      while estNonVide(B) :
05          x = defiler(B) # x est choisi parmi les éléments de open minimisant f
06          L.append(x) ; visite[x] = 2
07          if x == t :  break
```

```
    # On met ensuite à jour les tableaux d et f, et la bordure B
08          for y in successeurs(x) :
09           c = d[x] + w(x,y)
10          if visite[y] == 1 and c < d[y] :
11                d[y] = c ; f[y] = d[y] + h[y]
12                modifier(B,y,f) ; pere[y] = x
                # autrement dit, si y est dans B, on prend d[y] = min(d[y], d[x] + w(x,y))
13          elif visite[y] == 0 :
14                d[y] = c ; f[y] = d[y] + h[y]
15                ajouter(B,y,f) ; visite[y] = 1 ; pere[y] = x
```

Remarque : La liste L n'est pas nécessaire, car elle est redondante avec le tableau de marquage `visite`.

3) Heuristique

Dans de nombreux problèmes d'optimisation, on cherche une solution minimisant une certaine grandeur. Une heuristique est une méthode de calcul qui fournit rapidement une solution réalisable, non nécessairement optimale ou exacte, pour un problème d'optimisation difficile. Un algorithme de recherche qui garantit de trouver toujours une meilleure solution est dit **algorithme optimal**.

a) Pour accélérer la résolution, on utilise souvent une heuristique et une fonction d'évaluation associée qui permet de limiter la recherche et ainsi d'accélérer l'obtention d'une solution.

Dans la suite, on considère le problème caractéristique d'**un plus-court chemin dans un graphe valué (par une fonction ω) d'un sommet initial fixé s à un sommet final fixé t.**

On suppose connue une heuristique $h(x)$ qui étant donné un sommet x, évalue la distance de x à t.
En particulier, on prend $\boxed{h(t) = 0}$.

- **Algorithme A***. Lorsqu'un sommet est visité lors du parcours, on dispose de $d(x)$ le coût du chemin entre s et x.

On considère alors la fonction d'évaluation, appelé **coût heuristique** de x, définie par

$$f(x) = d(x) + h(x)$$

- **Algorithme de Dijkstra** : il s'agit d'un cas particulier de A* où on prend $h(x) = 0$.

Lorsque tous les poids des arêtes (par exemple égaux à 1), l'algorithme correspond alors à un parcours en largeur.

- **Méthode gloutonne** : on prend $f(x) = h(x)$. Autrement dit, on ne tient pas compte de $g(x)$.

b) Heuristique consistante

On dit que h est consistante ssi pour tout descendant y de x, on a $h(x) - h(y) \leq \omega(x, y)$.

Prop : **Si A* utilise une heuristique consistante, alors le chemin renvoyé par A* est optimal.**

Preuve : L'algorithme A* correspond exactement à l'algorithme de Dijkstra appliqué au poids

$$\omega'(x, y) = \omega(x, y) + h(y) - h(x)$$

Lorsque h est consistante, $\omega'(x,y) \geq 0$, et ainsi la preuve de Dijkstra (qui nécessite que les poids soient positifs) s'applique. En particulier, l'algorithme renvoie le chemin optimal reliant s à t pour la valuation ω'.

Le coût d'un chemin reliant s à t pour la valuation ω' vaut $\delta'(s,t) = \delta(s,t) - h(s)$, car $h(t) = 0$.
Donc comparer les $\delta'(s,t)$ revient à comparer les $\delta(s,t)$.
On obtient donc bien un chemin optimal pour la valuation ω.

Remarque : C'est le cas dans l'algorithme de Dijkstra avec $h(x) = 0$: h est bien consistante.

Remarque : Par le même argument, pour tout sommet x appartient à la liste close L, $d[x]$ est le coût minimal d'un chemin reliant s à x. Cette propriété peut être fausse si h n'est pas consistante. On serait alors amener au cours de l'algorithme à retrancher à certains moments des éléments de L pour les remettre des B (et la terminologie "liste close" pour L n'est plus vraiment judicieuse, ...).

c) Heuristique admissible (★) (complément culturel difficile)

On note $d(x,t)$ le coût minimal pour aller de x à t.

On dit que h est admissible ssi elle ne surestime jamais le coût : pour tout x, on a $h(x) \leq d(x,t)$.

Remarque : Toute heuristique consistante est admissible.
En effet, $d(s,t)$ est atteint pour un chemin $s = x_0 \to x_1 \to ... \to x_p = t$, et on a alors :
$d(x,t) = \sum_{k=1}^{p} \omega(x_{k-1}, x_k) \geq \sum_{k=1}^{p} h(x_{k-1}) - h(x_k) = h(x) - h(t) = h(x)$

Important : Avec une heuristique admissible, on n'obtient pas nécessairement le chemin optimal : c'est pourquoi, pour avoir A* optimal, il faut modifier le code de la façon suivante : chaque fois qu'on revient sur un sommet x appartenant à la liste close avec un coût heuristique plus faible, il faut sortir x de la liste fermée pour le réinsérer dans la liste ouverte avec ce nouveau coût $f(x)$.

Exemple d'heuristique admissible :

$$\begin{array}{ccccc} & \nearrow & x & \searrow & \\ s & & & & z \to t \\ & \searrow & y & \nearrow & \end{array} \text{ , avec } \begin{cases} \omega(s,x) = 2 \text{ et } \omega(x,z) = \omega(s,y) = \omega(y,z) = 1 \text{ et } \omega(z,t) = 3 \\ h(x) = 0 \text{ et } h(y) = 4 \text{ et } h(z) = h(t) = 0 \\ \text{on a bien } h(x) \leq d(x,t) = 4 \text{ et } h(y) \leq d(x,t) = 4 \end{cases}$$

C'est l'écart entre $h(x)$ et $h(y)$ qui va conduire à modifier l'algorithme initial :

En effet, une fois s placé dans L, x et y sont ajoutés à B, avec $f(x) = 2$ et $f(y) = 5$.
Donc x est ajouté à L, et y et z sont dans la liste B, avec $f(y) = 5$ et $f(z) = 3$.
Donc z est placé dans L, et y et t sont dans la liste B, avec $f(y) = 5$ et $f(t) = 6$.
Pour aboutir à la solution optimale $s \to y \to z \to t$, il faut pouvoir sortir z de la liste L afin de pouvoir le remettre dans B (comme successeur de y avec le nouveau coût $f(z) = 2$).

C'est pourquoi, on modifie le code en ajoutant les lignes :

```
16          if visite[y] == 2 and c < d[y] :
17               d[y] = c ; f[y] = d[y] + h[y]
18               ajouter(B,y,f) ; visite[y] = 1 ; pere[y] = x
```

On peut alors prouver : Si **A* modifié** utilise **une heuristique admissible,** alors il est **optimal.**

d) Exemples d'heuristiques

- Exemples d'heuristiques admissibles dans le jeu du taquin :

partant de

11	14	6	12
3	15	7	8
4	9	1	2
13	10	5	

, on veut obtenir

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

On peut prendre $h_1(x)$ = nombre de pièces mal placées, où x est l'état en cours

Ou bien prendre $h_2(x)$ = somme des distances de chaque pièce à sa position finale.

On dit que h_2 domine h_1 lorsque h_1 et h_2 sont admissibles et que $h_2(x) \geq h_1(x)$.

C'est le cas ici. C'est pourquoi h_2 est meilleure pour la recherche du plus court-chemin.

- Exemples d'heuristiques dans le problème des reines : il s'agit de placer des reines sur un échiquier de sorte qu'elles appartiennent à des lignes, des colonnes et des diagonales différentes.

On peut prendre $h(x)$ = nombre de conflits entre les reines.