CHAPITRE

7

PARCOURS DE GRAPHES

I - Principe général d'un parcours

Une première question que l'on peut se poser est la suivante : en partant d'un sommet s dans un graphe G, quels sont tous le sommets qui peuvent être atteints selon un parcours de G commençant par s ?
Dans cette optique, on dit qu'un sommet u est *atteignable* à partir de s si c'est s lui-même ou s'il existe un chemin de s à ce sommet.
Il faut mettre en place un algorithme qui *garde en mémoire* les possibilités laissées de côté. La mise en mémoire se fait dans une structure de données appelée *conteneur*. De manière générale en informatique les conteneurs sont utilisés pour stocker des objets sous une forme organisée qui suit des règles d'accès spécifiques. Par exemple, les piles, files et listes sont des conteneurs.
On considère alors l'algorithme générique théorique d'exploration suivant qui utilise un conteneur que l'on note $\mathcal{C}$.

---

explorer_th (G, s)

Données : un graphe G, un sommet s de G
Résultat : un sommet est atteignable à partir de s si et seulement s'il est marqué.
début
    création d'un conteneur $\mathcal{C}$ vide
    noter tous les sommets comme non marqués
    mettre s dans $\mathcal{C}$
    tant que $\mathcal{C}$ n'est pas vide **faire**
        enlever un sommet v de $\mathcal{C}$
        si v n'est pas marqué **alors**
            marquer v
            pour chaque voisin w de v **faire**
                mettre w dans $\mathcal{C}$

II - Parcours en profondeur

Il s'agit de l'algorithme obtenu lorsque le conteneur est une **pile**. Cet algorithme a la particularité d'effectuer sa recherche en allant le plus loin possible dans le graphe avant d'être obligé de repartir d'un sommet laissé de côté (dans $\mathcal{C}$ qui est une pile).

DFS_gen (G, s)

Données : un graphe G, un sommet s de G
Résultat : un sommet est atteignable à partir de s si et seulement s'il est marqué.
début
  création d'une pile vide p
  noter tous les sommets comme non marqués
  mettre s dans p
  tant que la pile p n'est pas vide **faire**
    $v \leftarrow$ dépiler(p)
    si v n'est pas marqué **alors**
      marquer v
      pour chaque voisin w de v **faire**
        mettre w dans la pile p si w n'est pas marqué

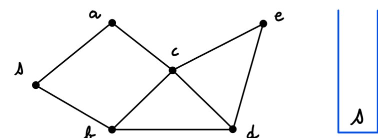
Lorsque les voisins de v sont empilés ils peuvent l'être dans n'importe quel ordre : l'algorithme n'est pas déterministe. On peut le rendre déterministe en empilant les sommets suivant un ordre qui est toujours le même, on peut prendre par exemple l'ordre lexicographique.

```python
def DFS_gen(gr, s):
    p = pile()          # création d'une pile vide
    empiler(p, s)       # s ajouté à la pile
    marque = {}
    for cle in gr:
        marque[cle] = False   # au début, tous les sommets sont à False
    while not(estVide(p)):
        v = depile(p)         # on enlève un sommet v de p
        if not(marque[v]):
            marque[v] = True
            # liste des sommets non marqués adjacents à v
            voisins_non_marques = [w for w in gr[v] if not(marque[w])]
            tri(voisins_non_marques)  # tri par ordre lexicographique
            for w in voisins_non_marques:  # empilement des sommets non marqués
                empiler(p, w)
    return marque
```

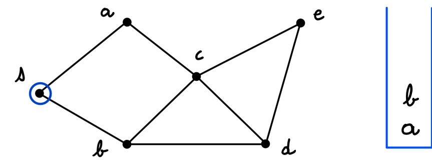
Exemple

Détaillons cet algorithme sur un exemple. On note les corps de boucles par leur numéro d'exécution : CB_n est le n-ième corps de boucle à être exécuté. Les schémas indiquent l'état des variables à la fin de chaque corps de boucle. On entoure dans le graphe les sommets qui sont marqués et on indique par des flèches le parcours effectué lorsque l'on passe d'un sommet marqué à celui qui est marqué juste après lui.

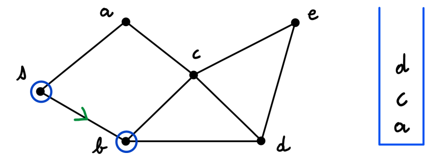
Initialisation :
Une pile vide est créée et le sommet source s est empilé.



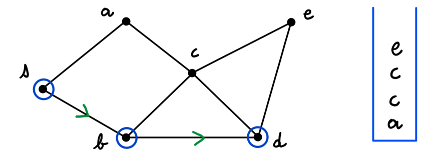
CB$_1$:
La pile n'est pas vide, elle est donc dépilée et on obtient $v = s$ qui est alors marqué : $\underline{s}$. Les voisins de s sont a et b, qui, comme ils sont tous les deux non marqués sont empilés dans l'ordre lexicographique (a en premier et b en second).



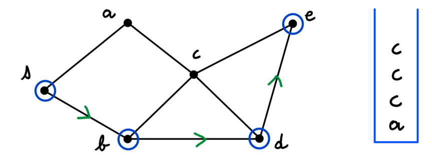
CB$_2$:
On obtient $v = b$ qui est alors marqué : $\underline{b}$. Les voisins de $\underline{b}$ sont $\underline{s}$, c et d. Seuls c et d ne sont pas marqués, donc empilés (on empile d'abord c puis d).



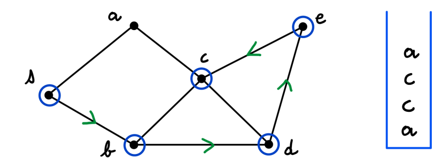
CB$_3$:
On obtient $v = d$ qui est alors marqué : $\underline{d}$. Les voisins de $\underline{d}$ sont $\underline{b}$, c et e. Seuls c et e ne sont pas marqués, donc empilés (on empile d'abord c puis e). On constate que le sommet c se retrouve deux fois dans la pile. La première fois qu'il sera dépilé, ses voisins seront examinés, mais la deuxième fois, rien de plus ne sera fait. On pourrait donc s'affranchir d'empiler des sommets qui sont déjà présents dans la pile. Mais ceci nécessiterait de garder à jour un conteneur indiquant si un sommet est présent ou pas dans la pile. C'est notamment faisable avec un *dictionnaire* dont les temps d'accès sont par construction en O(1). Mais comme l'empilement et le dépilement sont aussi en O(1) on ne changerait pas la classe de complexité ni même la complexité de l'algorithme.



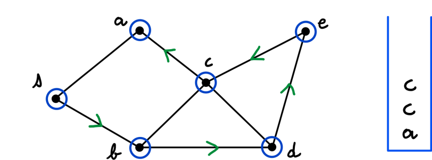
CB$_4$:
On obtient $v = e$ qui est alors marqué : $\underline{e}$. Les voisins de $\underline{e}$ sont c et $\underline{d}$. Seul c n'est pas marqué, donc empilé.



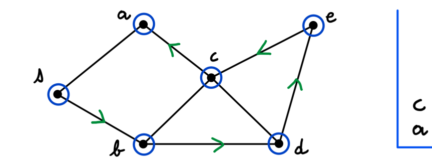
CB$_5$:
On obtient $v = c$ qui est alors marqué : $\underline{c}$. Le seul voisin de $\underline{c}$ qui n'est pas marqué est a qui est alors le seul sommet à être empilé.



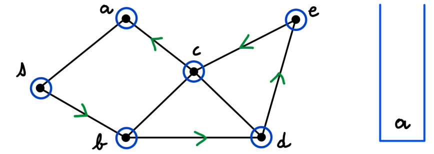
CB$_6$:
On obtient $v = a$ qui est alors marqué : $\underline{a}$. Le sommet a n'a aucun voisin qui n'est pas marqué, donc rien de plus n'est fait. On constate qu'arrivé à cette étape, tous les sommets sont marqués. Les autres corps de boucle ne vont donc que dépiler les autres sommets de la pile jusqu'à ce qu'elle soit vide.



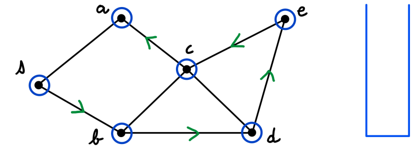
CB$_7$:
Le sommet c est dépilé.



CB$_8$:
Le sommet c est à nouveau dépilé.



CB$_9$:
Le sommet a est dépilé et la pile est maintenant vide.



L'algorithme d'exploration en profondeur est de nature fondamentalement récursive. Les piles utilisées précédemment sont inhérentes au fonctionnement des fonctions récursives, elles ont simplement été mises en évidence dans le cadre d'une programmation impérative. C'est pourquoi cet algorithme s'écrit récursivement de manière concise.

DFS_rec (G, s)

début
- marquer s
- **pour** chaque voisin v de s **faire**
  - **si** v n'est pas marqué **alors**
    - **DFS_rec** (G, v)

```python
def DFS_rec(graphe, sommet):
    marque[sommet] = True
    for v in graphe[sommet]:
        if not(marque[v]):
            DFS_rec(graphe, v)
```

III - Algorithme d'exploration en largeur

L'idée principale de cet algorithme est de découvrir les sommets par couches successives, les sommets de chaque couche étant éloignés d'un même nombre minimal d'arêtes du sommet source s.

Il s'agit de la particularisation de l'algorithme générique d'exploration dans le cas où le conteneur est une file.

BFS (G, s)

début
- création d'une file vide f
- noter tous les sommets comme non marqués
- mettre s dans f
- **tant que** la file f n'est pas vide **faire**
  - $v \leftarrow$ défiler(f)
  - **si** v n'est pas marqué **alors**
    - marquer v
    - **pour** chaque voisin w de v **faire**
      - mettre w dans la file f si w n'est pas marqué

Voici par exemple le programme permettant d'obtenir la classe de connexité s'un sommet.

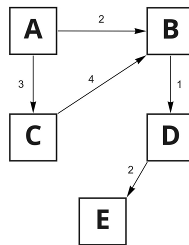
```
def BFS(gr, s):
    """
    Entrée : gr (type : dict), s (type : clé de gr)
    Sortie : liste de clés de gr
    """
    F = File()     # création d'une pile vide
    F.enfiler(s)   # s ajouté à la pile
    marque = {}
    for cle in gr:
        marque[cle] = False    # False : non marqué
    while not(F.empty()):
        v = F.defiler()  # défiler v de la file
        if not(marque[v]):
            marque[v] = True
            # liste des sommets non marqués adjacents à v
            voisins_non_marques = [w for w in gr[v] if not(marque[w])]
            # tri par ordre lexicographique croissant
            voisins_non_marques.sort()
            # emfilement des sommets non marqués dans la pile
            for w in voisins_non_marques:
                F.enfiler(w)

    classe_connexite = []
    for u in gr:
        if marque[u]:
            classe_connexite.append(u)

    return classe_connexite
```

IV - Plus courts chemins dans un graphe pondéré - algorithme de Dijkstra

On a essentiellement manipulé jusqu'à présent des graphes pour lesquels il n'y a pas d'orientation des arêtes, ni de valeurs associées aux arêtes. Ces deux dernières caractéristiques peuvent être utiles pour certaines modélisations. Par exemple, si l'on cherche une représentation d'un réseau de rues d'une ville, on peut utiliser un graphe dont les sommets sont les intersections de chaque rue avec une autre et les arêtes, les portions de rues entre chaque intersection. Il faut cependant tenir compte du fait que certaines rues sont à sens unique et que la distance entre deux intersections n'est pas partout la même. Ainsi on peut ajouter aux arêtes initiales un sens et une longueur. On obtient alors un graphe orienté et pondéré.

Exemple

Pour représenter un graphe pondéré en Python, il suffit dans les listes des voisins d'un sommet `s` d'utiliser un couple `(a, n)` où `a` est un voisin de `s` et `n` est le poids de l'arête ou de l'arc entre `s` et `a`.
Par exemple, considérons le graphes suivant :



Une façon de représenter la liste d'adjacence de ce graphe orienté pondéré est d'utiliser le dictionnaire suivant :

```
G = {'A' : [('B',2), ('C', 3)],
     'B' : [('D',1)],
     'C' : [('B',4)],
     'D' : [('E',2)],
     'E' : [] }
```

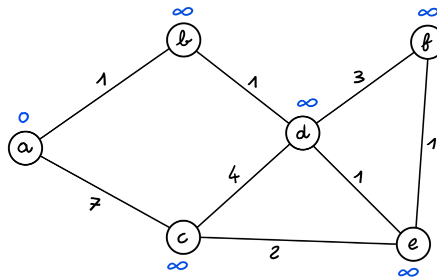
Pour un graphe pondéré donné, et un de ses sommets appelé *racine* et noté r, on cherche pour tout sommet u du graphe un plus court chemin d'extrémités r et u. Ce chemin n'est pas forcément unique.

Algorithme de Dijkstra

Données : G un graphe pondéré, S_A le sommet de départ, S_B le sommet d'arrivée
Sorties : un nombre représentant le poids minimal pour un chemin reliant S_A à S_B
début
  P ← liste vide
  pour chaque sommet s de G **faire**
    on pondère s par ∞
  on pondère S_A par 0
  tant que S_B n'est pas dans P **faire**
    S_{min} ← sommet de poids minimal qui n'est pas dans P
    on ajoute S_{min} à P
    pour chaque sommet s de G relié à S_{min} et qui n'est pas dans P **faire**
      on met à jour la pondération de s
  retourner la pondération de S_B

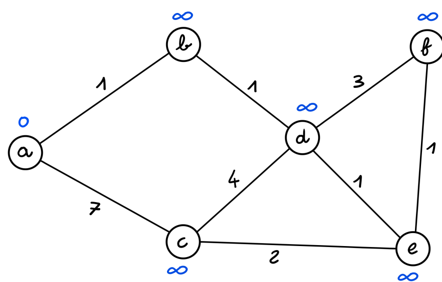
Exemple

Considérons le graphe (non orienté et pondéré) suivant :



Le sommet racine choisi est a. On va tenir à jour un tableau qui donne à la fin de chaque étape d pour chaque sommet ainsi que son parent éventuel. L'initialisation est considérée comme l'étape 0.

Étape 0 :

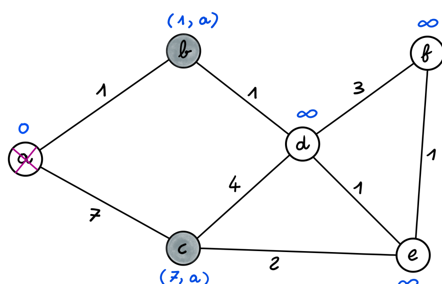


Étape	a	b	c	d	e	f
0	0	∞	∞	∞	∞	∞

Étape 1 :

On part du sommet source a, car c'est celui qui a la plus petite valeur de δ (on le souligne dans le tableau), et on essaie d'améliorer la valeur de δ de ses voisins qui sont b et c. Les poids des arêtes ab et ac étant 1 et 7, ce sont tout simplement les valeurs de δ pour b et c puisque pour le moment on n'a pas de chemins plus courts que (a, b) et (a, c) pour atteindre b et c à partir de a. On colorie alors a en noir pour le considérer comme exploré, et b et c en gris, sommets desquels on pourra repartir pour continuer à explorer le graphe. En pratique on placera juste une croix sur le sommet a comme convention d'un coloriage en noir. En ce qui concerne le coloriage en gris, on pourra considérer qu'un sommet qui n'est pas en noir et dont la distance n'est pas infinie, est en fait gris. On place ensuite une croix pour le sommet a dans le tableau car la valeur de d ne sera plus jamais remise en question (considération valable pour tous les sommets qui sont coloriés en noir).

On indique à côté de chaque sommet u le couple $(\delta(u), \text{parent}(u))$ si $\delta(u) \neq \infty$, de même dans la case du tableau associée au sommet u.

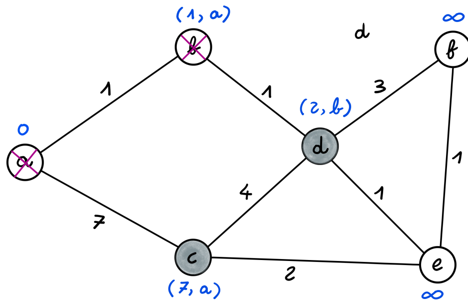


Étape	a	b	c	d	e	f
0	$\underline{0}$	∞	∞	∞	∞	∞
1	$\times$	$(1, a)$	$(7, a)$	∞	∞	∞

Étape 2 :

Pour continuer à progresser dans le graphe on repart d'un sommet qui minimise la fonction δ et dont les voisins n'ont pas encore été considérés, c'est donc parmi les sommets gris que l'on cherche le minimum de δ. Il s'agit du sommet b car $1 < 7$. On examine alors tous les voisins de b qui ne sont pas noirs afin

d'améliorer éventuellement leur valeur de distance δ. Le seul voisin de b qui n'est pas noir est le sommet d. On a $\delta(d) = \infty$, que l'on peut améliorer car en venant de b on a déjà un chemin venant de a de longueur $\delta(b) = 1$, auquel il faut ajouter le poids de l'arête bd, c'est-à-dire 1. On a ainsi amélioré $\delta(d)$ qui passe de l'infini à $1 + 1 = 2$. on note aussi que le sommet d a pour parent b. On termine en coloriant b en noir et d en gris.



Étape	a	b	c	d	e	f
0	0	∞	∞	∞	∞	∞
1	×	$(1, a)$	$(7, a)$	∞	∞	∞
2	×	×	$(7, a)$	$(2, b)$	∞	∞

Étape 3 :

Parmi les sommets gris, c'est le sommet d qui a la plus petite valeur de δ, c'est donc celui duquel on va repartir. Ses voisins non noirs sont : c, e et f. Le sommet b, voisin de d a déjà été exploré (il est colorié en noir), donc on ne le considère pas. On examine l'amélioration éventuelle de la fonction δ pour chacun de ces trois sommets :

sommet c :

On a $\delta(c) = 7$, mais en venant du sommet d il suffit d'ajouter l'arc dc pour obtenir un chemin reliant a à c de longueur $\delta(d) + 4 = 2 + 4 = 6$ qui est strictement inférieur à 7, longueur minimale d'un chemin d'extrémités a et c que l'on avait à notre disposition. On actualise alors les informations sur c :

- nouvelle valeur de d : $\delta(c) = 6$
- nouveau parent de c : d

L'ancien chemin (a, c) est plus long que le nouveau chemin (a, b, d) (au sens de la longueur définie pour un graphe pondéré).

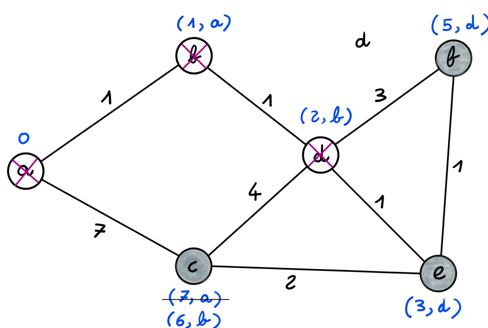
sommet e :

On a $\delta(e) = \infty$, on actualise alors là aussi les informations du sommet e : $(2 + 1, d)$, soit $(3, d)$.

sommet f :

On a $\delta(f) = \infty$, on actualise les informations du sommet f : $(2 + 3, d)$, soit $(5, d)$.

Le sommet d est colorié en noir, les sommets e et f en gris.



Étape	a	b	c	d	e	f
0	0	∞	∞	∞	∞	∞
1	×	$(1, a)$	$(7, a)$	∞	∞	∞
2	×	×	$(7, a)$	$(2, b)$	∞	∞
3	×	×	$(6, d)$	×	$(3, d)$	$(5, d)$

Étape 4 :

Le prochain sommet à considérer est le sommet e (valeur de δ minimale parmi les sommets gris). Ses voisins non noirs sont c et f. On vérifie si on peut diminuer ou pas les valeurs de $\delta(c)$ et $\delta(f)$.

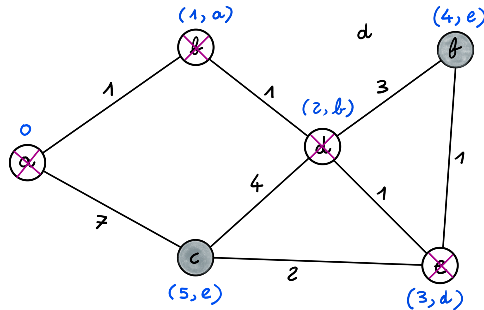
sommet c :

On a $\delta(c) = 6$. En venant e, la longueur du chemin serait de $\delta(e) + 2 = 3 + 2 = 5$, qui est plus petit que 6, meilleure valeur de δ trouvée jusqu'à présent pour c. On a donc trouvé un chemin de plus petit longueur pour c : (a, b, d, e, c). On actualise la valeur de δ et le parent pour c : $(5, e)$.

sommet f :

On a $\delta(f) = 5$. Ici encore, on venant de e on peut améliorer la longueur du chemin en $\delta(e) + 1 = 3 + 1 = 4$ qui est plus petit que 5. Actualisation des données pour le sommet f : $(4, d)$.

On colorie e en noir, c et e sont déjà gris.



Étape	a	b	c	d	e	f
0	0	∞	∞	∞	∞	∞
1	$\times$	$(1, a)$	$(7, a)$	∞	∞	∞
2	$\times$	$\times$	$(7, a)$	$(2, b)$	∞	∞
3	$\times$	$\times$	$(6, d)$	$\times$	$(3, d)$	$(5, d)$
4	$\times$	$\times$	$(5, e)$	$\times$	$\times$	$(4, e)$

Étape 5 :

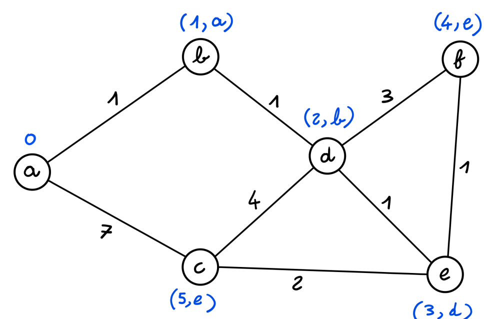
Le prochain sommet à traiter est f. Il n'a pas de voisins non traités, donc on ne peut améliorer leur valeur de δ. En effet, une fois qu'un sommet est noir, on ne revient plus sur sa valeur de δ. Le tableau reste inchangé dans ses valeurs, on note simplement que f est traité (croix sur la case de coordonnées $(5, f)$).

Étape	a	b	c	d	e	f
0	0	∞	∞	∞	∞	∞
1	$\times$	$(1, a)$	$(7, a)$	∞	∞	∞
2	$\times$	$\times$	$(7, a)$	$(2, b)$	∞	∞
3	$\times$	$\times$	$(6, d)$	$\times$	$(3, d)$	$(5, d)$
4	$\times$	$\times$	$(5, e)$	$\times$	$\times$	$(4, e)$
5	$\times$	$\times$	$(5, e)$	$\times$	$\times$	$\times$

Étape 6 :

Le sommet c est le seul sommet non traité qui reste, on ne peut donc améliorer la valeur de δ d'aucun sommet, on colorie c en noir et l'exécution de l'algorithme est terminée car tous les sommets ont été traités (tous les sommets sont noirs).

Étape	a	b	c	d	e	f
0	0	∞	∞	∞	∞	∞
1	$\times$	$(1, a)$	$(7, a)$	∞	∞	∞
2	$\times$	$\times$	$(7, a)$	$(2, b)$	∞	∞
3	$\times$	$\times$	$(6, d)$	$\times$	$(3, d)$	$(5, d)$
4	$\times$	$\times$	$(5, e)$	$\times$	$\times$	$(4, e)$
5	$\times$	$\times$	$(5, e)$	$\times$	$\times$	$\times$
6	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$



```python
def Dijkstra(gr, r):
    """
    Renvoie le tuple (d, parent) :
    - d est le dictionnaire dont les clés sont les sommets
      de gr et les valeurs les distances minimales au sommet r.
    - parent est le dictionnaire donnat pour chaque sommet,
      son parent dans le chemin le plus court le menant à r.
    """
    # Initialisation de d, couleur et parent
    d = {}
    couleur = {}
    parent = {}
    for u in gr.keys():
        d[u] = float('inf')
        couleur[u] = 'blanc'
    d[r] = 0
    couleur[r] = 'gris'
    # Début de l'exploration
    while gris_non_vide(couleur):
        u = gris_min(couleur, d)
        couleur[u] = 'noir'
        for v, poids in gr[u]:
            if couleur[v] != 'noir':
                d_test = d[u] + poids
                if d_test < d[v]:
                    d[v] = d_test
                    parent[v] = u
                    couleur[v] = 'gris'
    # Renvoi des résultats
    return (d, parent)
```

Algorithme A*

L'algorithme A* est un algorithme de recherche d'un chemin dans un graphe orienté et pondéré entre deux sommets fixés à l'avance : le sommet de départ et le sommet d'arrivée. Il a été découvert en 1968 par Nilsson et ses collaborateurs afin d'aider un robot à se déplacer dans une pièce contenant des obstacles. Il est un des algorithmes les plus répandus en intelligence artificielle et programmation des jeux.

Le but est de trouver le chemin de coût minimal entre un sommet source s et un sommet t qui doit terminer le chemin. L'algorithme de Dijkstra répond déjà à la question. Cependant, les graphes sur lesquels on est amené à travailler sont souvent issus de problèmes qui contiennent des informations qui ne sont pas représentées par le graphe. On peut alors parfois se servir de ces informations pour obtenir une estimation, notée $h(u)$, du coût du chemin de coût minimum qui reste à parcourir d'un sommet u à t. C'est cette estimation $h(u)$ qu'exploite l'algorithme A* pour accélérer la recherche. La fonction h est appelée **heuristique**. Cette accélération de la recherche peut parfois ne pas donner un chemin de coût minimum au profit de la vitesse d'exécution.

L'algorithme de Dijkstra utilise pour chaque sommet u, le coût pour aller de la source s au sommet u. Il progresse de manière gloutonne en essayant d'augmenter d le moins possible au cours de son exploration. Avec la connaissance de $h(u)$ on obtient une *estimation du coût total* d'un chemin de s à t, sous la forme de $f(u)$ s'écrivant $f(u) = d(u) + h(u)$ et c'est bien la version exacte de cette grandeur que l'on cherche à minimiser, d'où l'idée de privilégier lors de l'exploration du graphe, les sommets de plus petite valeur de f, plutôt que de d. On obtient ainsi l'**algorithme A***. Cet algorithme est donc très proche de celui de Dijkstra, à la différence qu'au lieu d'utiliser d seul, on utilise f.