Graphes

I Vocabulaire des graphes

Un graphe est une construction visant à formaliser ou modéliser des relations (les arêtes ou arcs) entre des données (les sommets). Ce peut être utile dans des domaines variés :

— une **carte routière** décrit les axes routiers (arêtes) reliant des villes (sommets). Ordre de grandeur : en France, il y a environ 35 000 communes.

— un **arbre généalogique** décrit la relation de descendance (arcs) entre différents membres d'une même famille (sommets).

— un **réseau social** peut être décrit par les liens d'amitié virtuelle ou d'abonnement (arêtes ou arcs) reliant des utilisateurs (sommets). Ordre de grandeur : Facebook compte presque 3 milliards d'utilisateurs actifs chaque mois, Twitter 400 millions.

— le **graphe du web** décrit les liens hypertextes (arcs) entre les sites internet (sommets). Ordre de grandeur : 2 milliards de sites internet dans le monde, 30 millions d'articles sur Wikipédia (toutes langues confondues).

(ces ordres de grandeur datent de 2020 ou 2021)

1 Graphes non orientés

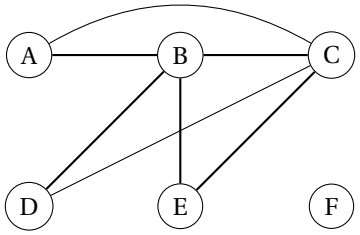
Définition (Graphe non orienté, sommet, arête, degré)**.** Un **graphe non orienté** G est un objet du type $G = (S, A)$ où :

1. S est un ensemble fini appelé ensemble des **sommets** (ou **nœuds**) du graphe G,
2. $A \subset \{\{s,t\} | (s,t) \in S \times S \text{ et } s \neq t\}$ est appelé ensemble des **arêtes** du graphe G.

On appelle **degré** d'un sommet $s \in S$ et on note $d(s)$ le nombre d'arêtes contenant ce sommet.

Remarque. Soit $x \in S$. Il faut autoriser les arêtes de type $\{x\}$ pour autoriser les **boucles**, c'est-à-dire permettre de se rendre de x à x.

Exemple. On considère le graphe non orienté $G = (S, A)$ donné par la représentation graphique :

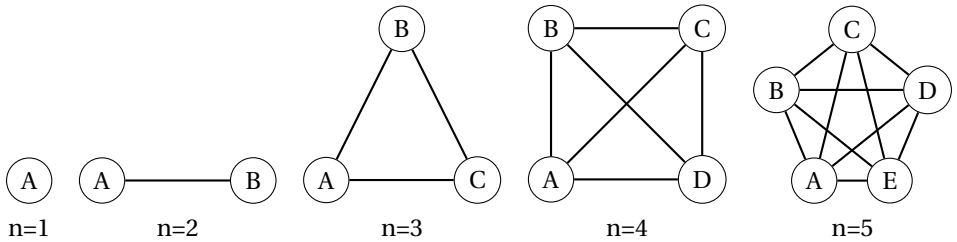


On a ici $S = \{A, B, C, D, E, F\}$, $A = \big\{\{A,B\}, \{A,C\}, \{B,C\}, \{B,D\}, \{B,E\}, \{C,D\}, \{C,E\}\big\}$.
Par ailleurs, $d(A) = 2$, $d(B) = 4$, $d(C) = 4$, $d(D) = 2$, $d(E) = 2$, $d(F) = 0$.

Définition. On dit que le graphe G est complet lorsque tout couple de sommets disjoints est relié par une arête.

Remarque. Si $G = (S, A)$ est un graphe complet et si $n = \text{Card}(S)$ alors $A = \{\{s,t\} | (s,t) \in S \times S \text{ et } s \neq t\}$ (tout couple de sommets définit une arête). On en déduit donc en dénombrant : $\text{Card}(A) = \binom{n}{2} = \frac{n(n-1)}{2}$.

Exemples.



2 Graphes orientés

Définition (Graphe orienté, sommet, arc)**.** Un **graphe orienté** G est un objet du type $G = (S, A)$ où :

1. S est un ensemble fini appelé ensemble des **sommets** (ou nœuds) du graphe G,
2. $A \subset S \times S$ est appelé ensemble des **arcs** du graphe G.

Remarque. Dans un graphe orienté, les arcs ont un sens de parcours : l'existence d'un arc (x, y) signifie que l'on peut passer du sommet x au sommet y mais pas forcément du sommet y au sommet x.

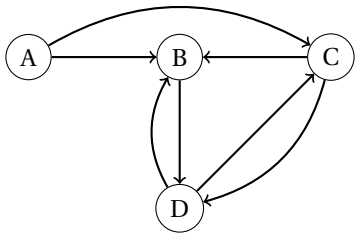
Remarque. Les arcs de type (x, x) sont toujours appelés **boucles**.

Définition (Degré entrant, degré sortant)**.** Soit $s \in S$ un sommet.

— On appelle **degré entrant** de s, noté $d^-(s)$, le nombre d'arcs entrant vers s, c'est-à-dire de la forme (x, s) avec $x \in S$.

— On appelle **degré sortant** de s, noté $d^+(s)$, le nombre d'arcs sortant de s, c'est-à-dire de la forme (s, x) avec $x \in S$.

Exemple. On considère le graphe orienté $G = (S, A)$ donné par la représentation graphique :



On a ici $S = \{A, B, C, D\}$, $A = \{(A, B), (A, C), (B, D), (C, B), (C, D), (D, B), (D, C)\}$.
Par ailleurs, $d_+(A) = 2$, $d_-(A) = 0$, $d_+(B) = 1$, $d_-(B) = 3$, $d_+(C) = 2$, $d_-(C) = 2$, $d_+(D) = 2$, $d_-(D) = 2$.

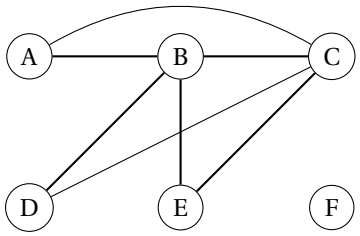
3 Implémentation

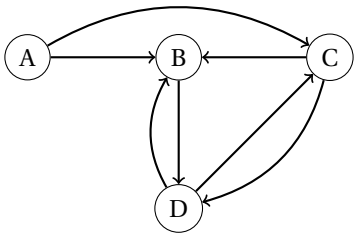
Définition (Liste d'adjacence)**.** Soit $G = (S, A)$ un graphe. On appelle **liste d'adjacence** du graphe G la liste, pour chaque sommet s, des sommets accessibles depuis s par un arc ou une arête.

Remarque. Les listes d'adjacentes sont particulièrement pratiques pour décrire les graphes où il y a peu d'arcs ou d'arêtes, ou pour les approches qui se basent sur les sommets du graphe.

Remarque. Certaines variantes proposent de mettre dans la liste d'adjacence les sommets qui permettent d'accéder à s, plutôt que les sommets auxquels s permet d'accéder. Le fonctionnement reste globalement le même.

Exemple. Reprenons les cas des exemples précédents et établissons les listes d'adjacence associées :





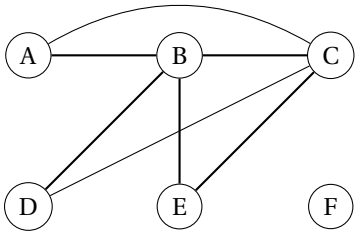
Définition (Matrice d'adjacence)**.** Soit $G = (S, A)$ un graphe. Si on note $n = \text{Card}(S)$, il existe une bijection ϕ entre $[\![1, n]\!]$ et S.

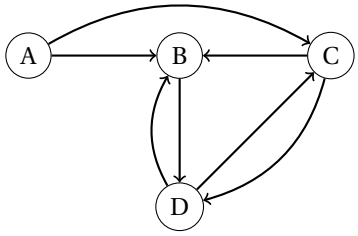
On appelle **matrice d'adjacence** du graphe G la matrice carrée $M = (m_{i,j}) \in \mathcal{M}_n(\mathbb{R})$ définie par :

$$M_{i,j} = \begin{cases} 1 \text{ s'il existe une arête/un arc de } \phi(i) \text{ à } \phi(j) \\ 0 \text{ sinon} \end{cases}.$$

Remarque. Les matrices d'adjacentes sont particulièrement pratiques pour décrire les graphes où il y a beaucoup d'arcs ou d'arêtes, ou pour les approches qui se basent sur les arcs ou arêtes du graphe.

Exemple. Reprenons les cas des exemples précédents et établissons les matrices d'adjacence associées :





Remarque. Les matrices d'adjacence de graphes non orientés sont symétriques.

Remarque. Les diagonales des matrices d'adjacentes sont constituées uniquement de 0, sauf dans le cas où le graphe contient une boucle.

4 Chemins, cycles et connexité

Définition (Chemin)**.** Soit $G = (S, A)$ un graphe. On appelle **chemin** toute suite finie non vide de sommets telle que chaque paire de sommets consécutifs de la suite soit une arête ou un arc du graphe.

La **longueur** d'un chemin est le nombre d'arêtes qui constituent ce chemin.

Remarque. Le terme « chaîne » est parfois utilisé à la place de « chemin » dans les graphes non-orientés, mais le programme ne fait pas cette distinction.

Remarque. Un chemin de longueur $m \in \mathbb{N}$ possède $m+1$ sommets.

Propriété. *Soit $G = (S, A)$ un graphe. On note $n = \text{Card}(S)$, ϕ une bijection entre $[\![1, n]\!]$ et S et M la matrice d'adjacence de G.*

Le coefficient (i, j) de la matrice M^p contient le nombre de chemins de longueur p reliant $\phi(i)$ à $\phi(j)$.

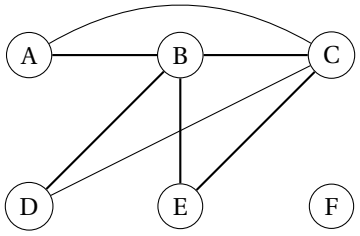
Définition (Cycle)**.** Soit $G = (S, A)$ un graphe.

Un **cycle** est un chemin contenant au moins une arête, dont le sommet de départ et d'arrivée sont les mêmes et qui n'utilise jamais deux fois la même arête ou le même arc.

On dit que G est **acyclique** s'il ne contient pas de cycle.

Remarque. Un cycle a quand même le droit de passer plusieurs fois par le même sommet, c'est sur les arêtes/arcs que porte la contrainte d'unicité.

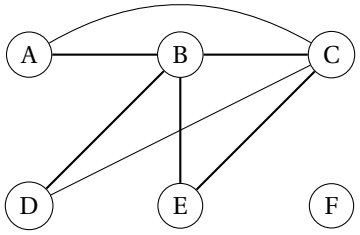
Exemple. Sur le graphe suivant, le chemin $A \to B \to C \to A$ est un cycle :



Définition (Graphe connexe)**.** Soit $G = (S, A)$ un graphe non orienté. On dit qu'il est **connexe** s'il existe un chemin entre tout couple de sommets.

Remarque. Autrement dit, un graphe G est connexe si tout sommet est accessible depuis tous les autres sommets.

Exemple. Le graphe suivant n'est pas connexe :



En effet, il n'existe (par exemple) pas de chemin entre A et F. Le sous-graphe auquel on a retiré le sommet F est par contre connexe : il existe alors un chemin entre tout couple de sommets.

II Parcours en largeur

1 Principe et files

Définition (Distance entre deux sommets)**.** Soit $G = (S, A)$ un graphe, $(s, t) \in S^2$ et $L_{s,t} \subset \mathbb{N}$ l'ensemble des longueurs des chemins de s à t. On appelle distance de s à t la valeur :

$$\begin{cases} \mathcal{M}in(L_{s,t}) \text{ si } L_{s,t} \neq \emptyset \\ +\infty \text{ sinon} \end{cases}.$$

Remarque. Dans le cas d'un graphe connexe, les distances entre sommets sont toutes finies.

Le parcours en largeur d'un graphe suit le principe suivant :

1. On choisit un sommet de départ, noté r.
2. On visite les sommets situés à une distance 1 de r, puis ceux à une distance 2, puis ceux à une distance 3…
3. On s'arrête après avoir visité tous les sommets accessibles depuis r.
4. On recommence avec un nouveau sommet de départ non visité (s'il en reste), jusqu'à avoir exploré tous les sous-graphes connexes de G.

Définition (File, enfiler, défiler)**.** On appelle **file** toute suite finie d'objets à laquelle on peut appliquer les opérations suivantes :

— renvoyer la longueur de la file,

— ajouter un objet à la fin de la file (**enfiler**),

— retirer le premier élément de la file et renvoyer sa valeur (**défiler**).

Remarque. On parle de modèle en « premier arrivé, premier servi » ou « first in, first out » (FIFO).

Un parcours en largeur d'une composante connexe s'implémente facilement à l'aide d'une file (qui stocke les sommets à traiter). Initialement, la file ne contient que le sommet de départ r. À chaque itération de l'algorithme, on défile le premier sommet de la file, puis pour chacun de ses voisins s non encore exploré :

— on enfile s,

— on marque s comme vu,

— éventuellement, on étiquette sur s sa distance à r : il s'agit de la distance de son père à r incrémentée de 1.

L'algorithme s'arrête lorsque la file est vide. On le relance alors sur un nouveau sommet non marqué (s'il en reste), jusqu'à avoir exploré toutes les composantes connexes du graphe.

2 Implémentation de l'algorithme

En pseudo-code, pour l'exploration d'une composante connexe du graphe G depuis un sommet r, l'algorithme donne :

```
Exploration_composante(G,r):
  Marquer r comme vu (et l'étiqueter à 0)
  Enfiler r
  Tant que la file n'est pas vide:
    Défiler un sommet v
    Pour tout sommet s adjacent à v et non encore marqué:
      Marquer s comme vu (et étiqueter sa distance à r)
      Enfiler s
```

Ce qui permet de parcourir le graphe G en entier avec le code suivant :

```
Parcours_largeur(G):
  On initialise tous les sommets comme non marqués
  Pour tout sommet u non marqué:
    Exploration_composante(G,u)
```

III Parcours en profondeur

1 Principe et piles

Le parcours en profondeur d'un graphe suit le principe suivant :

1. On choisit un sommet de départ, noté r, qu'on explore et marque.
2. Tant qu'il reste des sommets non marqués et accessibles, on en choisit un (en privilégiant les sommets reliés par une arête aux derniers sommets visités). On marque le sommet choisi et on poursuit l'exploration depuis ce nouveau sommet.
3. On s'arrête après avoir visité tous les sommets accessibles depuis r.
4. On recommence avec un nouveau sommet de départ non marqué (s'il en reste), jusqu'à avoir exploré tous les sous-graphes connexes de G.

Remarque. On dit que l'on procède en profondeur car la visite privilégie toujours un sommet qui est relié par une arête au précédent sommet visité.

Définition (Pile, empiler, dépiler)**.** On appelle **pile** toute suite finie d'objets à laquelle on peut appliquer les opérations suivantes :

— renvoyer la longueur de la file,

— ajouter un objet à la fin de la pile (**empiler**),

— retirer le dernier élément de la pile et renvoyer sa valeur (**dépiler**).

Remarque. On parle de modèle en « dernier arrivé, premier servi » ou « last in, first out » (LIFO).

Un parcours en profondeur d'une composante connexe s'implémente facilement à l'aide d'une pile (qui stocke les sommets à traiter). Initialement, la pile ne contient que le sommet de départ r. À chaque itération de l'algorithme, tant que la pile est non vide :

— si le dernier sommet de la pile possède un voisin non marqué s, on empile et marque s.

— sinon, on dépile le dernier sommet de la pile.

L'algorithme s'arrête lorsque la pile est vide. On le relance alors sur un nouveau sommet non marqué (s'il en reste), jusqu'à avoir exploré toutes les composantes connexes du graphe.

2 Implémentation de l'algorithme

En pseudo-code, pour l'exploration d'une composante connexe du graphe G depuis un sommet r, l'algorithme donne :

```
Exploration_composante(G,r):
  Marquer r comme vu
  Enpiler r
  Tant que la pile n'est pas vide (on note v le sommet du haut de la pile):
    Si v possède un sommet s adjacent et non encore marqué:
      Marquer s comme vu
      Empiler s
    Sinon:
      Dépiler v
```

On peut aussi en proposer une variante sans pile mais avec récursivité :

```
Exploration_composante(G,r):
  Marquer r comme vu
  Pour tout sommet s adjacent à v et non encore marqué:
    Exploration_composante(G,s)
```

Ce qui permet de parcourir le graphe G en entier avec le code suivant :

```
Parcours_Profondeur(G):
  On initialise tous les sommets comme non marqués
  Pour tout sommet u non marqué:
    Exploration_composante(G,u)
```

IV Algorithme de Dijkstra

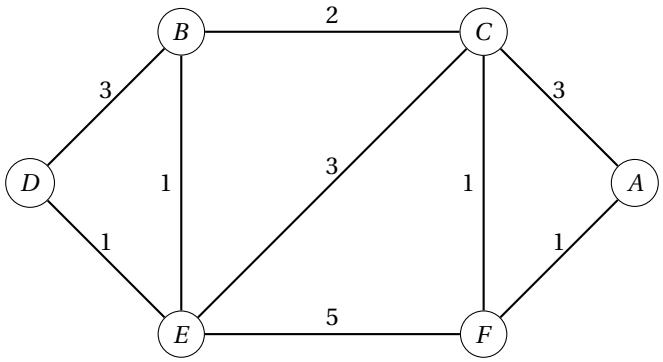
IV.1 Graphes pondérés

Définition (Graphe pondéré)**.** Un graphe pondéré $G = (S, A)$ est un graphe muni d'une fonction de pondération g définie de A dans $\mathbb{R}$. La valeur $g(s,t)$ s'appelle le **poids** de l'arc (s,t) ou de l'arête $\{s,t\}$.

Le poids d'un chemin est le somme des poids des arcs par lesquels il passe.

Exemple. Dans le cas d'une carte routière, l'arête reliant deux villes peut être pondérée par la distance qui les sépare, par le temps mis pour se rendre de l'une à l'autre, par le prix du trajet...

Exemple.
On considère le graphe $G = (S, A)$ de représentation graphique :



IV.2 Présentation de l'algorithme

Soit $G = (S, A)$ un graphe (orienté ou non) muni d'une fonction de poids g. L'algorithme de Dijkstra recherche les plus courts chemins menant d'un sommet $s \in S$ fixé et se rendant à chaque autre sommet.

Remarque. On suppose les poids de G positifs, ce qui empêche d'avoir des cycles de poids strictement négatifs.

Notons $\delta(t)$ la longueur du plus court chemin menant de s à t, on va chercher d'une part à déterminer cette valeur, d'autre part à isoler le chemin associé.

Pour cela, on munit chaque sommet $v \in$ § d'une étiquette $e(v)$, initialisée à $+\infty$ (sauf $e(s)$ qui vaut 0) et modifiée au fur et à mesure de l'algorithme (de sorte à contenir $\delta(v)$ à la fin). On fait en sorte d'avoir à tout moment et pour tout sommet $v \in S$ que $\delta(v) \leq e(v)$.

À chaque étape :

— On sélectionne un sommet u d'étiquette $e(u)$ est la plus faible (parmi les sommets non encore traités).

— Pour chaque sommet v voisin de u et non encore traité, on teste s'il est possible d'améliorer le plus court chemin de s à v en passant par u..

On s'arrête une fois tous les sommets sélectionnés et visités.

En pseudo-code, cela donne :

```
Dijkstra(G,s):
  Initialiser les étiquettes (à 0 ou infini suivant les sommets)
  Tant qu'il reste des sommets non visités:
    Choisir un sommet non visité u d'étiquette la plus faible
    Marquer u comme visité
    Pour tout sommet v voisin de u et non visité:
      Si e(u) + g(u,v) < e(v):
        e(v) = e(u) + g(u,v)
```

Proposition (Correction).

L'algorithme de Dijkstra lancé à partir d'un sommet s étiquette tous les sommets atteignables à partir de s. La valeur finale de l'étiquette d'un sommet v est la distance théorique minimale entre s et v.

Démonstration. Soit $n \in \mathbb{N}^*$ le nombre de sommets du graphe. On note :

- $v_0, \ldots, v_{n-1}$ les sommets du graphe.
- $g_{u,v}$ la distance entre deux sommets voisins u et v (donnée par la fonction de poids).
- $\delta(v)$ la distance théorique minimale entre s et v.
- $a_0, \ldots, a_{n-1}$ les sommets du graphe rangés par ordre croissant de distance à s. Notamment, $a_0 = s$ et $\delta(a_0) = 0 \leqslant \delta(a_1) \leqslant \delta(a_2) \leqslant \ldots \leqslant \delta(a_{n-1})$.
- $\tilde{a}_0, \ldots, \tilde{a}_{n-1}$ les sommets du graphe rangés dans l'ordre où ils sont traités par l'algorithme de Dijkstra. Notamment, $\tilde{a}_0 = s$ et $\tilde{a}_1$ sera un voisin de s.

De plus, pour tout entier $i \in [\![0, n-1]\!]$ et tout sommet v, on note $e_i(v)$ la valeur de l'étiquette de v après l'étape i de l'algorithme. Notamment, pour tout sommet $v \neq s$, $e_0(v) = +\infty$. Si $i \in [\![0, n-1]\!]$, on a aussi $e_i(s) = 0$.

Soit $i \in [\![0, n-1]\!]$, on souhaite prouver l'invariant de boucle suivant :

$$H(i) : \begin{cases} \tilde{a}_i = a_i \\ e_i(a_i) = \delta(a_i) \\ \forall v \in S, e_i(v) \geqslant \delta(v) \end{cases}.$$

- $\tilde{a}_0 = s = a_0$, $e_0(s) = 0 = \delta(s)$ et $\forall v \neq s$, $e_0(v) = +\infty \geqslant \delta(v)$. Donc $H(0)$ est vraie.
- Soit $i \in [\![0, n-2]\!]$, on suppose que $\forall k \in [\![0, i]\!]$, $H(k)$ est vraie (c'est-à-dire que $H(0)$, $H(1)$, $\ldots$, $H(i)$ sont vraies). Montrons que $H(i+1)$ est vraie.
  On s'intéresse au sommet a_{i+1} et au chemin optimal qui y mène partant de s, qu'on note :

  $$s \to t_1 \to \ldots \to t_{j-1} \to t_j \to a_{i+1}$$

  L'avant dernier sommet visité t_j est par définition plus proche de s que ne l'est a_{i+1}, donc $\exists p \in [\![0, i]\!]$ tel que $t_j = a_p$ (c'est-à-dire t_j est l'un des sommets $a_0, \ldots, a_i$). Donc a_{i+1} est voisin d'un sommet déjà traité. De plus, toujours d'après ce chemin optimal, on a :

  $$\delta(a_{i+1}) = \delta(a_p) + g_{a_p, a_{i+1}}.$$

  En effet, la distance parcourue lors du chemin $s \to t_1 \to \ldots \to t_{j-1} \to a_p$ doit elle-même être optimale entre s et a_p, sans quoi le chemin complet de s à a_{i+1} ne serait pas optimal. Il s'agit donc bien de $\delta(a_p)$.
  Par hypothèse de récurrence $H(p)$, $\delta(a_p) = e_p(a_p)$, donc $\delta(a_{i+1}) = e_p(a_p) + g_{a_p, a_{i+1}}$.
  Or la valeur $e_p(a_p) + g_{a_p, a_{i+1}}$ a été proposée comme étiquetage de a_{i+1} lorsque a_p a été traité, et les valeurs des étiquettes ne peuvent que diminuer. On en déduit que $e_i(a_{i+1}) \leqslant \delta(a_{i+1})$. L'inégalité contraire étant assurée par hypothèse de récurrence $H(i)$, on a :

  $$e_i(a_{i+1}) = \delta(a_{i+1}).$$

  Considérons maintenant les sommets non encore traités par l'algorithme. Pour tout $k \in [\![i+2, n-1]\!]$, l'hypothèse de récurrence $H(i)$, l'inégalité $k \geqslant i+1$ et la définition des a_j donnent :

  $$e_i(a_k) \geqslant \delta(a_k) \geqslant \delta(a_{i+1}) = e_i(a_{i+1}).$$

  Cela signifie que $e_{i+1}(a_{i+1})$ est le minimum des étiquettes des sommets non encore traités. Autrement dit, le sommet $\tilde{a}_{i+1}$ choisi par Dijkstra (en prenant le minimum des étiquettes) est a_{i+1} :

  $$\boxed{\tilde{a}_{i+1} = a_{i+1}}.$$

L'étiquette du sommet a_{i+1} ne sera donc plus modifiée après l'étape i (l'algorithme ne touche plus aux étiquettes des sommets traités ou en cours de traitement). On en déduit que $e_{i+1}(a_{i+1}) = e_i(a_{i+1})$, et donc :

$$\boxed{e_{i+1}(a_{i+1}) = \delta(a_{i+1})}.$$

On se place maintenant après avoir traité a_{i+1}, c'est-à-dire avoir actualisé les étiquettes de ses voisins. Pour $v \in S$, les $e_{i+1}(v)$ sont toujours supérieurs aux $\delta(v)$. En effet :

— soit l'étiquette de v n'a pas été changée et donc on a $e_{i+1}(v) \geqslant \delta(v)$ par hypothèse de récurrence $H(i)$.

— soit l'étiquette de v a été changée et $e_{i+1}(v) = e_{i+1}(a_{i+1}) + g_{a_{i+1},v} = \delta(a_{i+1}) + g_{a_{i+1},v}$. Cette quantité peut être vue comme la distance du chemin $s \to \ldots \to a_{i+1} \to v$ (où $s \to \ldots \to a_{i+1}$ est le chemin optimal entre s et a_{i+1}). Comme tout chemin entre s et v, il a une longueur supérieure à $\delta(v)$.

En résumé,

$$\boxed{\forall v \in S, \quad e_{i+1}(v) \geqslant \delta(v)}.$$

Donc $H(i+1)$ est vraie.

En conclusion, si on laisse l'algorithme de Dijkstra traiter tous les sommets, il étiquette correctement chacun de ces sommets, dans l'ordre croissant de leur distance à s. □

Remarque. Si l'on ne s'intéresse qu'à un sommet d'arrivée spécifique, cette preuve montre que l'on peut stopper l'algorithme dès que l'on traite ce sommet.

VI Algorithme A^*

1 Principe

L'algorithme A^* est une variante de l'algorithme de Dijkstra qui cherche à l'accélérer en tirant profit de connaissances sur le graphe :

— Intuitivement, A^* oriente le parcours de l'algorithme de Dijkstra en privilégiant les directions qui semblent plus prometteuses.

— Mathématiquement, A^* consiste à appliquer l'algorithme de Dijkstra sur un graphe auxiliaire, dont les distances ont été modifiées par une pénalisation. Dans l'idéal, ces pénalisations ne changent pas le résultat de l'algorithme, elles permettent juste d'aboutir plus rapidement.

Définition (Heuristique)**.** Soit $G = (S, A)$ un graphe. On appelle heuristique une fonction $h : S \to \mathbb{R}$.

Dans le cas où on s'est donné un point de départ $s \in S$ et un point d'arrivée $a \in S$, l'heuristique associe à chaque sommet v un nombre idéalement d'autant plus petit que v est proche du sommet arrivée.

Exemple. Soit $C \in \mathbb{R}$. On peut poser $\forall v \in S,\ h(v) = C\|v - a\|$.

Si x et y sont deux sommets voisins et si on note $g_{x,y} \geq 0$ la distance qui les sépare, on définit une pseudo-distance $g'_{x,y}$ par :

$$g'_{x,y} = g_{x,y} + h(y) - h(x)$$

L'algorithme A^* revient à utiliser l'algorithme de Dijkstra avec ces pseudo-distances (plutôt qu'avec les distances d'origine).

Remarque. Ces pseudo-distances sont d'autant plus petites que la quantité $h(y) - h(x)$ est négative, c'est-à-dire que l'arête mène à un sommet pour lequel la valeur de h est plus faible. On pénalise donc les arêtes, en diminuant le poids de celles qui « rapprochent » de a au sens de la fonction h, et en augmentant le poids de celles qui « éloignent » de a.

L'utilisation d'une heuristique est censée diminuer le nombre de chemine à tester en "orientant" l'algorithme vers les chemins les plus prometteurs. Il existe cependant des cas-comme celui de la figure 1, où ces chemins ne seront finalement pas les meilleurs !

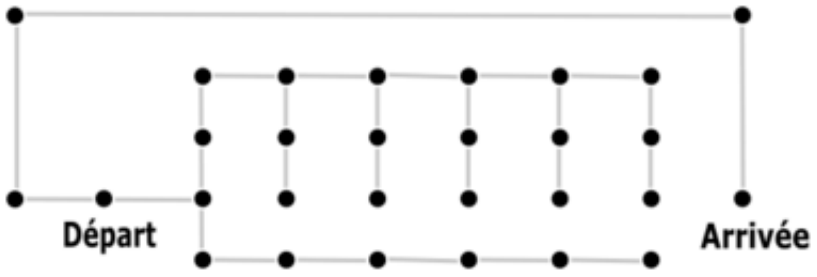


FIGURE 1 – Un exemple où l'heuristique décrite ci-dessus sera inefficace