

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

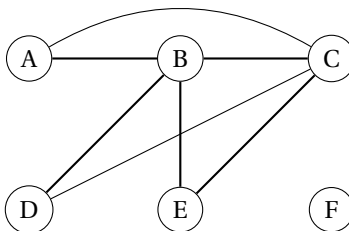
Définition (Graphe non orienté, sommet, arête, degré). Un **graphe non orienté** G est un objet du type $G = (S, A)$ où :

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 .

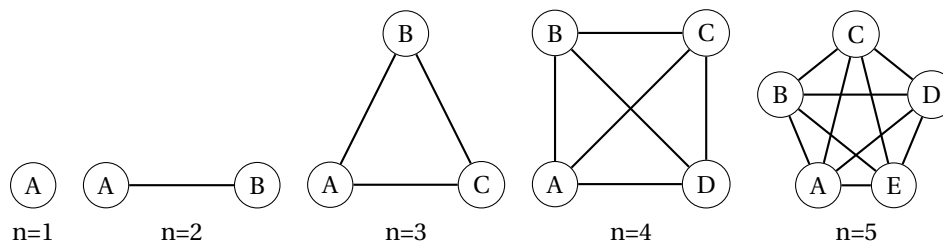
Exemple. On considère le graphe non orienté $G = (S, A)$ donné par la représentation graphique :



On a ici $S = \{A, B, C, D, E, F\}$, $A = \{\{A, B\}, \{A, C\}, \{B, C\}, \{B, D\}, \{B, E\}, \{C, D\}, \{C, E\}\}$.

Par ailleurs, $d(A) = 2$, $d(B) = 4$, $d(C) = 4$, $d(D) = 2$, $d(E) = 2$, $d(F) = 0$.

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



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 : $\text{Card}(A) = \binom{n}{2} = \frac{n(n-1)}{2}$.

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 .

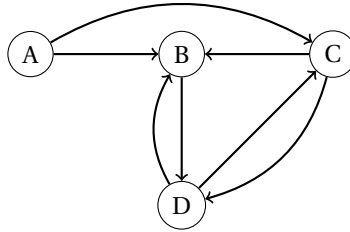
Remarque. Dans un graphe orienté, les arcs ont un sens de parcours : l'existence d'un arc (x, y) signifie que l'on peut passer du sommet x au sommet y mais pas forcément du sommet y au sommet x .

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, $d_+(A) = 2$, $d_-(A) = 0$, $d_+(B) = 1$, $d_-(B) = 3$, $d_+(C) = 2$, $d_-(C) = 2$, $d_+(D) = 2$, $d_-(D) = 2$.

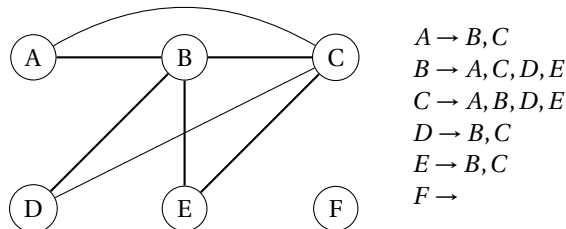
3 Implémentation

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.

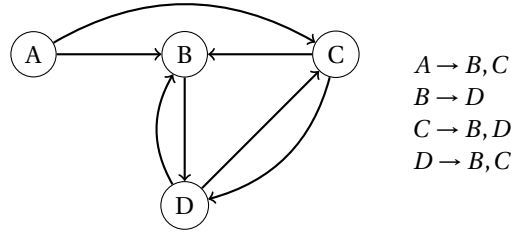
Remarque. Les listes d'adjacentes sont particulièrement pratiques pour décrire les graphes où il y a peu d'arcs ou d'arêtes, ou pour les approches qui se basent sur les sommets du graphe.

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 :



$A \rightarrow B, C$
 $B \rightarrow A, C, D, E$
 $C \rightarrow A, B, D, E$
 $D \rightarrow B, C$
 $E \rightarrow B, C$
 $F \rightarrow$



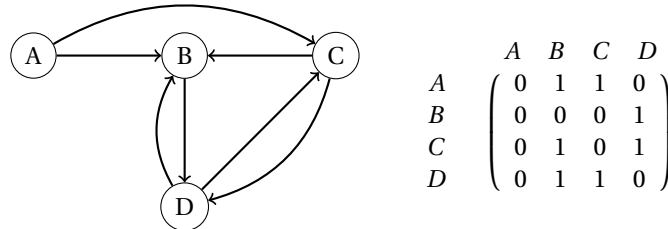
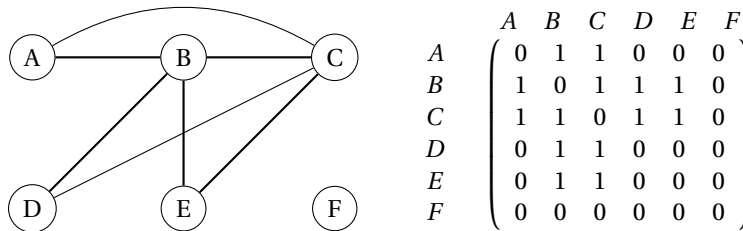
Définition (Matrice d'adjacence). Soit $G = (S, A)$ un graphe. Si on note $n = \text{Card}(S)$, il existe une bijection ϕ entre $[[1, n]]$ et S .

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 $\llbracket 1, n \rrbracket$ 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 $\phi(j)$.

Démonstration. (À FAIRE SAUTER ÉVENTUELLEMENT SI MANQUE DE TEMPS?)

Soit $p \in \mathbb{N}$. Pour tout couple $(i, j) \in \llbracket 1, n \rrbracket \times \llbracket 1, n \rrbracket$, on note :

- $c_{i,j}(p)$ le nombre de chemins de longueur p reliant les sommets $\phi(i)$ et $\phi(j)$.
- $m_{i,j}(p)$ le coefficient située à la i -ème ligne et j -ième colonne de M^p .

On pose $P(p) : \llcorner \forall (i, j) \in \llbracket 1, p \rrbracket \times \llbracket 1, p \rrbracket, c_{i,j}(p) = m_{i,j}(p) \rceil$.

- Les seuls chemins de longueur 0 sont ceux constitués seulement d'un sommet. Donc $\forall (i, j) \in \llbracket 1, n \rrbracket \times \llbracket 1, n \rrbracket$,

$$c_{i,j}(0) = \begin{cases} 1 & \text{si } i = j \\ 0 & \text{si } i \neq j \end{cases}.$$

Par ailleurs, $M^0 = I_n$. Ainsi, $\forall (i, j) \in \llbracket 1, n \rrbracket \times \llbracket 1, n \rrbracket$, $c_{i,j}(0) = m_{i,j}(0)$. Donc $P(0)$ est vraie.

- Soit $p \in \mathbb{N}$, on suppose que $P(p)$ est vraie. Soit $(i, j) \in \llbracket 1, n \rrbracket \times \llbracket 1, n \rrbracket$. $c_{i,j}(p+1)$ représente le nombre de chemins entre $\phi(i)$ et $\phi(j)$ de taille $p+1$, donc en sommant sur les sommets adjacents à $\phi(j)$,

$$\begin{aligned} c_{i,j}(p+1) &= \sum_{\substack{k=1 \\ \phi(k) \text{ est relié à } \phi(j)}}^n c_{i,k}(p) \\ &= \sum_{k=1}^n c_{i,k}(p) \times m_{k,j}(1) \\ &= \sum_{k=1}^n m_{i,k}(p) \times m_{k,j}(1) \text{ par hypothèse de récurrence} \\ c_{i,j}(p+1) &= m_{i,j}(p+1) \text{ par définition du produit de matrices} \end{aligned}$$

Donc $P(p+1)$ est vraie.

On a donc démontré la propriété annoncée. □

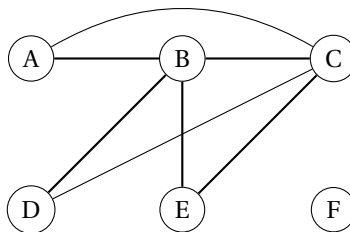
Définition (Cycle). Soit $G = (S, A)$ un graphe.

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.

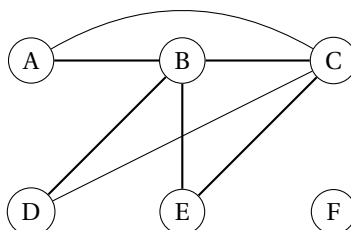
Exemple. Sur le graphe suivant, le chemin $A \rightarrow B \rightarrow C \rightarrow A$ est un cycle :



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 :

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

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 enfiler 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
  Tant qu'il existe un sommet u non marqué:
    Exploration_composante(G,u)
```

Exemple. On détaille l'algorithme d'exploration du graphe suivant à partir du sommet A . Initialement, aucun sommet n'est marqué. Les sommets marqués lors de cette exploration apparaîtront en rouge sur le graphe. La colonne de gauche indique l'évolution de la file d'appel au fur et à mesure de l'algorithme.

A

 On marque et étiquette A , puis enfile A .

B
G

 On défile A , marque, étiquette et enfile les sommets adjacents à A .

G
C
D

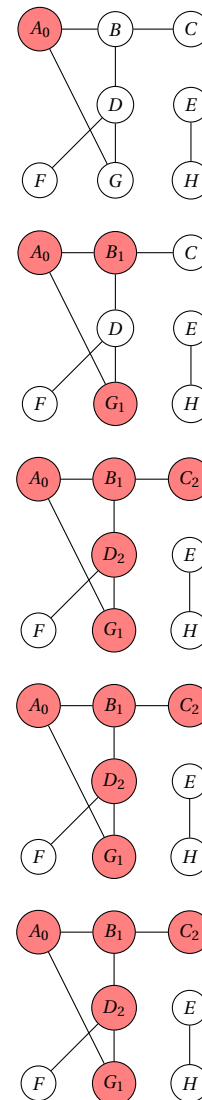
 On défile B , marque, étiquette et enfile les sommets adjacents à B et non encore marqués.

C
D

 On défile G , qui n'a pas de sommet adjacent non marqué.

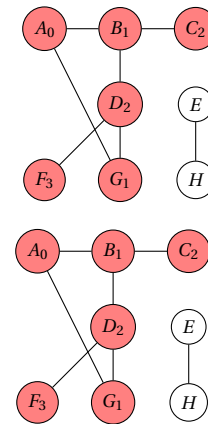
D

 On défile C , qui n'a pas de sommet adjacent non marqué.



☐ F On défile D , marque, étiquette et enfile le seul sommet adjacent à D et non encore marqué.

☐ On défile F , qui n'a pas de sommet adjacent non marqué.

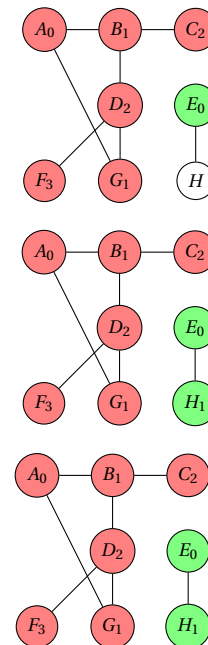


La file est alors vide : on a exploré toute la composante connexe de A . On recommence avec un nouveau sommet non encore marqué, par exemple E . Les sommets marqués lors de cette nouvelle exploration apparaîtront en vert sur le graphe.

☐ E On marque et étiquette E , puis enfile E .

☐ H On défile E , marque, étiquette et enfile le seul sommet adjacent à E et non encore marqué.

☐ On défile H , qui n'a pas de sommet adjacent non marqué.



La file est alors vide : on a exploré toute la composante connexe de E , et tous les sommets du graphe ont été marqués.
Exemple. (À PASSER EN TD?)

Ajouter un exemple de parcours en largeur sur un arbre (pour mieux illustrer ensuite la différence entre parcours en largeur et en profondeur), par exemple sur l'arbre des puissances (issu de *The Art of Computer Programming*, Knuth) pour donner une application pratique :

3 Terminaison, complexité et correction

Propriété (Terminaison). *L'algorithme de parcours en largeur termine.*

Démonstration. (À PASSER EN TD?)

On sait que le nombre de sommets est fini. Un sommet qui a été enfilé une fois par le parcours en largeur est marqué, il ne peut donc pas être enfilé de nouveau. La file ne peut donc recevoir aucun sommet plus d'une fois.

Or un sommet est défilé à chaque passage de boucle : la file finit donc nécessairement par se vider.

(On peut aussi utiliser comme variant de boucle la somme du nombre de sommets non encore marqués et du nombre de sommets dans la file). □

Propriété (Complexité). *La complexité d'un parcours en largeur est en $O(n + a)$, où n et a représentent les nombres de sommets et d'arêtes du graphe.*

Remarque. On suppose pour ce résultat que chaque opération sur la file est de coût constant, ce qui peut ne pas être le cas suivant l'implémentation.

Démonstration. (À PASSER EN TD?)

Soit E l'ensemble contenant tous les sommets qui sont dans la file et tous ceux qui n'y ont pas encore été. À chaque passage de boucle, E perd un élément et $ACard(E)$ décroît strictement de 1. Donc il y a $O(n)$ passages dans la boucle « tant que ».

La boucle « pour » coûte le nombre de sommets adjacents à v . Les v sont tous différents à chaque fois (c'est l'élément qui quitte E) donc tout le travail effectué dans tous les passages de la boucle « pour » coûte la somme sur v du nombre de sommets adjacents, c'est-à-dire $O(a)$. $\square$

Propriété (Correction). *L'algorithme de parcours en largeur explore toute la composante connexe du sommet de départ.*

Démonstration. Notons r le sommet choisi comme racine du parcours et C sa composante connexe dans le graphe. On veut démontrer que chaque sommet de C apparaît dans la file (et est donc exploré).

On raisonne par l'absurde. Soit $s \in C$, supposons que s n'apparaît jamais dans la file. Comme $s \in C$, il existe un chemin $r = v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_p = s$ menant de r à s . Soit $m \in \llbracket 1, p \rrbracket$ l'indice minimal pour lequel v_m n'apparaît pas dans la file (m existe puisque v_0 apparaît dans la file, mais pas v_p).

Par construction, v_{m-1} apparaît dans la file. Donc ce sommet est traité avant la fin de l'algorithme, et tous ses voisins sont ajoutés dans la file à cette occasion. Donc v_m est apparu dans la file : absurde. Donc s a bien été ajouté à la file et exploré par l'algorithme. $\square$

Propriété (Correction dans le cas étiqueté). *On considère l'exploration par un parcours en largeur partant de r de la composante connexe C . Pour tout $t \in C$, l'étiquette attribuée au sommet t correspond à la longueur du plus court chemin de r à t .*

Démonstration. Pour $t \in C$, on note $e(t)$ l'étiquette attribuée à t et $\delta(t)$ la longueur du plus court chemin de r à t (cette longueur existe bien puisque $t \in C$).

Soit $n \in \mathbb{N}$. On pose P_n : « Il existe un moment dans l'exécution tel que, pour tout $t \in S$, $(\delta(t) \leq n \Rightarrow e(t) = \delta(t))$ et $(\delta(t) = n \text{ ssi } t \text{ est dans la file})$. »

— À l'instant initial, r est le seul sommet de la file. Or $\delta(r) = 0$ et r est le seul sommet situé à une distance 0 de lui-même. Donc $P(0)$ est vraie.

— Soit $n \in \mathbb{N}$, on suppose que $P(n)$ est vraie. La file ne contient donc que les sommets situés à une distance n (et elle les contient tous). On se place au moment où tous ces sommets auront été défilé, et où la file contient leurs successeurs. Soit $t \in S$ tel que $\delta(t) \leq n + 1$. Si $\delta(t) \leq n$, $e(t) = \delta(t)$ par $P(n)$ (puisque une étiquette n'est jamais modifiée après avoir été attribuée).

Si $\delta(t) = n + 1$, notons c un plus court chemin de r à t et u le prédécesseur de t dans c . Le sous-chemin $r \rightarrow u$ de c reste un plus court chemin et $\delta(u) = n$ donc, comme $e(t) = e(u) + 1$, $e(t) = n + 1 = \delta(t)$. De plus t est un successeur de u , donc t se trouve dans la file à l'instant considéré.

Réciproquement, si un sommet s se trouve dans la file à l'instant considéré, alors il existe un sommet u pour lequel $\delta(u) = n$ dont il est le successeur. Donc $\delta(t) = e(t) = e(u) + 1 = n + 1$.

Donc $P(n + 1)$ est vraie.

On a donc bien montré la correction de l'algorithme. $\square$

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  
  Empiler  $r$   
  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  $r$  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

Tant qu'il existe un sommet u non marqué:

Exploration_composante(G,u)

Exemple. On détaille l'algorithme d'exploration du graphe G suivant à partir du sommet A . Initialement, aucun sommet n'est marqué. Les sommets marqués lors de cette exploration apparaîtront en rouge sur le graphe. La colonne de gauche indique l'évolution de la pile d'appel au fur et à mesure de l'algorithme.

A

 On marque et empile A .

B
A

 On marque et empile B , voisin de A .

C
B
A

 On marque et empile C , voisin de B .

B
A

 On dépile C , qui n'a pas de voisin non marqué.

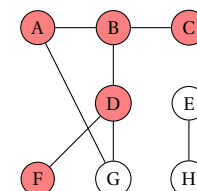
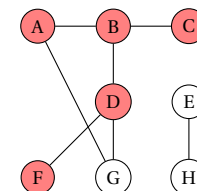
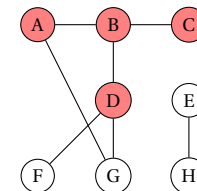
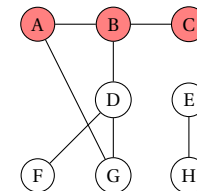
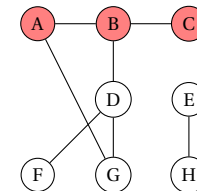
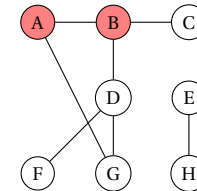
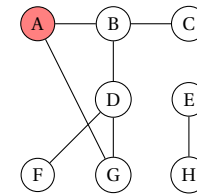
D
B
A

 On marque et empile D , voisin de B .

F
D
B
A

 On marque et empile F , voisin de D .

D
B
A

 On dépile F , qui n'a pas de voisin non marqué.


G
D
B
A

On marque et empile G , voisin de D .

D
B
A

On dépile G , qui n'a pas de voisin non marqué.

B
A

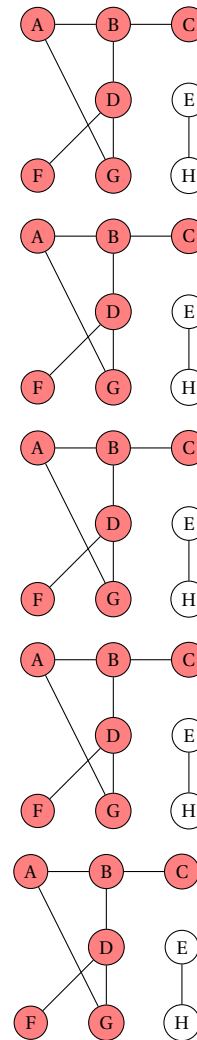
On dépile D , qui n'a pas de voisin non marqué.

A

On dépile B , qui n'a pas de voisin non marqué.

--

On dépile A , qui n'a pas de voisin non marqué.



La pile est alors vide : on a exploré toute la composante connexe de A . On recommence avec un nouveau sommet non encore marqué, par exemple E . Les sommets marqués lors de cette nouvelle exploration apparaîtront en vert sur le graphe.

E

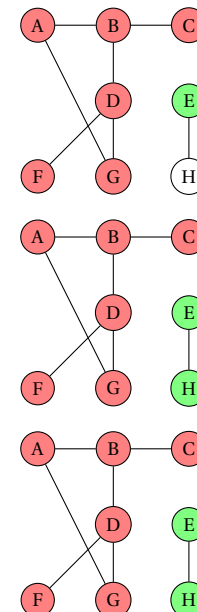
On marque et empile E .

H
E

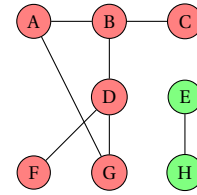
On marque et empile H .

E

On dépile H , qui n'a pas de voisin non marqué.



□ On dépile E , qui n'a pas de voisin non marqué.



La pile est alors vide : on a exploré toute la composante connexe de E , et tous les sommets du graphe ont été marqués.

Exemple. (À PASSER EN TD?)

Traiter l'arbre de l'exemple ??exarbre, cette fois-ci à l'aide d'un parcours en profondeur.

3 Terminaison et complexité

Propriété (Terminaison). *L'algorithme de parcours en profondeur termine.*

Démonstration. (À PASSER EN TD?)

On sait que le nombre de sommets est fini.

— Version avec pile :

Un sommet qui a été empilé une fois par le parcours en largeur est marqué, il ne peut donc pas être empilé de nouveau. La pile ne peut donc recevoir aucun sommet plus d'une fois. De plus, à chaque passage de boucle, soit un sommet est marqué, soit un sommet est dépilé.

Un variant de boucle qui convient est donc le nombre de sommets non marqués comptés deux fois auquel on ajoute le nombre de sommets dans la pile.

— Version récursive :

La fonction `Exploration_composante` n'est lancée que sur des sommets non marqués, et est donc lancée au plus une fois sur chaque sommets (puisque la fonction marque le sommet sur lequel elle est lancée). Le nombre d'appels total est donc fini, ce qui garantit la terminaison.

□

Propriété (Complexité). *La complexité d'un parcours en profondeur est en $O(n + a)$, où n et a représentent les nombres de sommets et d'arêtes du graphe.*

Démonstration. (À PASSER EN TD?)

Soit E l'ensemble contenant tous les sommets qui sont dans la pile et tous ceux qui n'y ont pas encore été, comptés deux fois (ou l'ensemble des sommets non encore marqués pour la version récursive). À chaque itération, E perd un élément et $ACard(E)$ décroît strictement de 1. Donc il y a $O(n)$ passages dans la boucle « tant que » (ou appels de fonctions pour la version récursive).

La suite nécessite d'explorer les sommets adjacents à v non encore marqués. Étudions d'un bloc tous les appels depuis le sommet v (c'est immédiat pour le cas récursif, mais v peut revenir plusieurs fois en haut de la pile dans l'autre version). Chaque voisin de v est étudié au plus une fois au total (s'il n'a pas convenu la première fois, on ne le réétudie pas : il ne conviendra pas davantage les fois suivantes). Le travail total effectué coûte donc la somme sur v du nombre de sommets adjacents, c'est-à-dire $O(a)$.

□

Propriété (Correction). *L'algorithme de parcours en profondeur explore toute la composante connexe du sommet de départ.*

Démonstration. Notons r le sommet choisi comme racine du parcours et C sa composante connexe dans le graphe. On veut démontrer que chaque sommet de C apparaît dans la pile (et est donc exploré).

On raisonne par l'absurde. Soit $s \in C$, supposons que s n'apparaît jamais dans la pile. Comme $s \in C$, il existe un chemin $r = v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_p = s$ menant de r à s . Soit $m \in \llbracket 1, p \rrbracket$ l'indice minimal pour lequel v_m n'apparaît jamais dans la pile (m existe puisque v_0 apparaît dans la pile, mais pas v_p).

Par construction, v_{m-1} apparaît à un moment dans la pile. Donc ce sommet est dépilé avant la fin de l'algorithme (qui ne s'interrompt que quand la pile est vide). Au moment où il est dépilé, il l'est parce que tous ses voisins (et donc v_m) sont déjà marqués. Comme le marquage n'a lieu qu'au moment de l'ajout à la pile, v_m est apparu dans la pile : absurde. Donc s a bien été ajouté à la pile et exploré par l'algorithme.

La preuve fonctionne de même dans le cas récursif en remplaçant « ajouté à la pile » par « appelé par la fonction récursive » □

IV Applications des algorithmes de parcours

1 Étude de la connexité

Les algorithmes de parcours explorent toute la composante connexe d'un sommet donné. Un graphe G est donc connexe si et seulement si le premier appel de la fonction d'exploration marque tous les sommets de G .

Dans le cas contraire, les appels successifs à la fonction d'exploration permettent de repérer les composantes connexes du graphe G .

2 Détection de cycles

Propriété. Soit $G = (S, A)$ un graphe non orienté et $r \in S$.

Dans le cas d'un parcours en largeur, G possède un cycle si et seulement si au cours d'un parcours partant de r , un sommet que l'on défile a parmi ses voisins un sommet dans la file.

Dans le cas d'un parcours en profondeur, G possède un cycle si et seulement si au cours d'un parcours partant de r , un sommet que l'on dépile a au moins deux de ses voisins dans la pile.

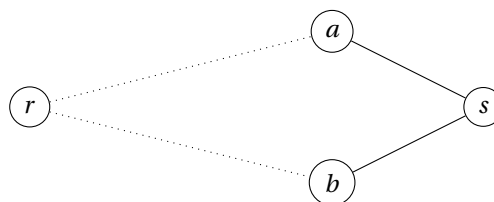
Exemple.

- Dans l'exemple 2, on détecte le cycle $A \rightarrow B \rightarrow D \rightarrow G \rightarrow A$ au moment de défiler G puisque son voisin D se trouve dans la file.
- Dans l'exemple 2, on détecte le cycle $A \rightarrow B \rightarrow D \rightarrow G \rightarrow A$ au moment de dépiler G , puisque ses voisins A et D se trouvent dans la pile.

Démonstration. (À FAIRE SAUTER ÉVENTUELLEMENT SI MANQUE DE TEMPS?)

La preuve est réalisée dans le cas d'un parcours en largeur, il faudrait ensuite l'adapter au cas du parcours en profondeur. On raisonne en deux temps.

- Supposons d'abord qu'il existe un sommet a qui, au moment où on le défile dans le parcours de graphe, admet parmi ses voisins un sommet s qui est déjà dans la file. Comme c'est un voisin, il existe donc une arête entre a et s . De plus, comme s était déjà dans la file, il existe un autre sommet b , traité auparavant, également voisin de s . a et b sont des sommets traités, ils peuvent donc être connectés à la racine r par des arêtes déjà explorées. Ces chemins remontant à la racine n'utilisent pas les arêtes $\{a, s\}$ et $\{b, s\}$ puisque s n'a pas encore été traité.



Il existe donc un chemin reliant r à a puis s puis b puis a . On peut montrer (ATTENTION : il faudrait le faire pour que ce soit complètement rigoureux mais c'est long, on peut peut-être se contenter de traiter un exemple sur un dessin) que si certaines arêtes sont utilisées plusieurs fois, il est possible de simplifier le chemin en retirant certaines parties, ce qui permet à terme d'obtenir un cycle.

- Supposons maintenant qu'il existe un cycle. Quitte à le réduire, on peut supposer que chaque sommet n'apparaît qu'une seule fois. Dans le parcours de graphe, chacun des sommets de ce cycle apparaît une et une unique fois dans la file.

Considérons le sommet s qui apparaît en dernier. Considérons de plus a et b ses deux voisins (distincts) dans le cycle. Au moment où s est ajouté à la file, au moins l'un des sommets a ou b n'a pas encore été traité (car sinon s serait déjà dans la file). Mettons que a n'a pas été traité. Par définition de s , il est déjà dans la file. Par conséquent, a sera traité avant s , et aura donc, au moment où il est traité, un voisin qui est déjà dans la file (à savoir s).

Donc il existe un sommet qui, quand on le traite, admet parmi ses voisins un sommet qui est déjà dans la file.

□

V Algorithme de Dijkstra

1 Graphes pondérés

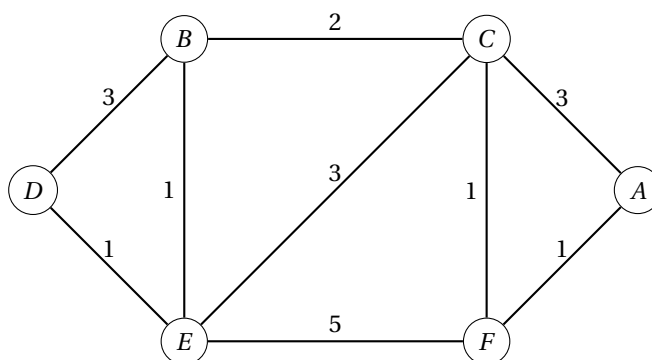
Définition (Graphe pondéré). Un graphe pondéré $G = (S, A)$ est un graphe muni d'une fonction de pondération g définie de A dans $\mathbb{R}$. La valeur $g(s, t)$ s'appelle le **poids** de l'arc (s, t) ou de l'arête $\{s, t\}$.

Le poids d'un chemin est la 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 :



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 $v \in S$ 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 :

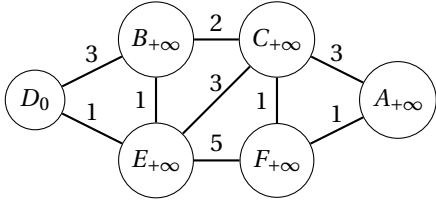
Dijkstra(G, s):

Initialiser les étiquettes (à 0 ou infini suivant les sommets)

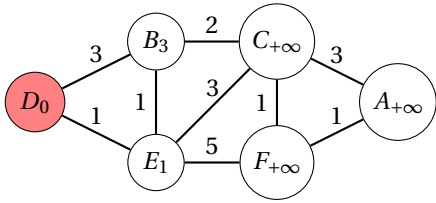
Tant qu'il reste des sommets non visités:

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 $e(u) + g(u,v) < e(v)$:
 $e(v) = e(u) + g(u,v)$

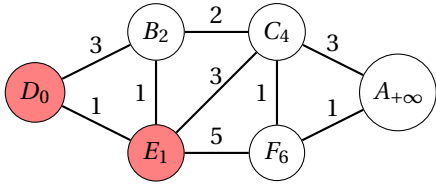
Exemple. Sur l'exemple précédent,



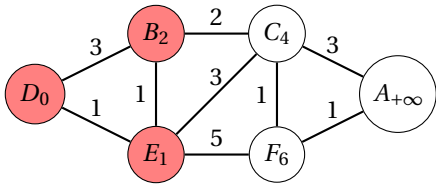
On initialise en $s = D$.



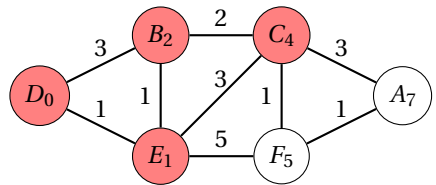
On traite $u = D$ et teste la mise à jour de $e(B)$ et $e(E)$.



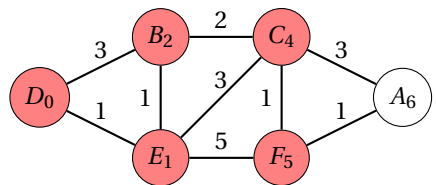
On traite $u = E$ et teste la mise à jour de $e(B)$, $e(C)$ et $e(F)$.



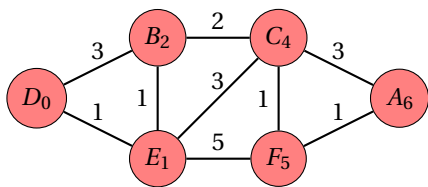
On traite $u = B$ et teste la mise à jour de $e(C)$.



On traite $u = C$ et teste la mise à jour de $e(F)$ et $e(A)$.



On traite $u = F$ et teste la mise à jour de $e(A)$.



On traite $u = A$ et l'algorithme termine.

Les étiquettes contiennent alors les valeurs des plus courtes distances partant de D .

Pour déterminer les plus courts chemins, on adapte l'algorithme précédent : quand on modifie $e(v)$, on garde en mémoire le sommet u à l'origine de cette modification. Le plus court chemin sera alors d'aller en u , puis en v .

Exemple. On reprend l'exemple précédent en synthétisant les étapes de l'algorithme dans un tableau :

A	B	C	D	E	F
$+\infty$	$+\infty$	$+\infty$	0	$+\infty$	$+\infty$
$+\infty$	3^D	$+\infty$	0	1^D	$+\infty$
$+\infty$	2^E	4^E	0	1	6^E
$+\infty$	2	4	0	1	6
7	2	4	0	1	5^C
6^F	2	4	0	1	5
6	2	1	0	4	5

On reconstitue alors facilement les plus courts chemins, par exemple :

- Le plus court chemin de D vers A est réalisé par $D \rightarrow E \rightarrow C \rightarrow F \rightarrow A$.
- Le plus court chemin de D vers E est réalisé par $D \rightarrow E$.
- Le plus court chemin de D vers B est réalisé par $D \rightarrow E \rightarrow B$.

Exemple. (À PASSER EN TD?)

Exécuter de nouveau l'algorithme à la main : soit sur un autre graphe, soit sur le même graphe mais en partant d'un autre sommet.

3 Terminaison, complexité et correction

Propriété (Terminaison). *L'algorithme de Dijkstra termine.*

Démonstration. (À PASSER EN TD?)

On sait que le nombre de sommets est fini. Comme chaque sommet est traité exactement une fois (il est ensuite marqué comme visité), le nombre d'appels de la boucle « Tant que » est fini.

(Autrement dit, le nombre de sommets non encore traités est un variant de boucle.) □

Propriété (Complexité). *La complexité de l'algorithme de Dijkstra est en $O(n^2)$, où n représente le nombre de sommets du graphe.*

Démonstration. (À PASSER EN TD? Si oui, passer le coût du choix du prochain sommet par comparaison des étiquettes comme question intermédiaire)

Notons n et a les nombres de sommets et d'arêtes du graphe.

On commence par étudier le coup de traitement d'un sommet u :

- on considère son étiquette $e(u)$ (à laquelle on ne touche pas).
- on liste ses voisins non encore traités $v_1, \dots, v_k$.
- on teste s'il est nécessaire d'actualiser les étiquettes $e(v_i)$
- on les actualise le cas échéant, en notant dans ce cas que le sommet u est le dernier à avoir entraîné une modification.

Ces opérations sont toutes élémentaires, donc leur coût pour l'ensemble des sommets est un $O(n + a)$.

Cela ne tient cependant pas compte d'un point important de l'algorithme : après avoir traité un sommet, il faut déterminer le sommet suivant, et donc chercher parmi les sommets non traités celui qui a l'étiquette la plus petite. Le programme officiel encourage à utiliser la version naïve, en testant successivement tous les sommets. La complexité du procédé est en $O(n^2)$.

Comme $a = O(n^2)$ (dans le pire des cas, un sommet est relié à tous les autres), on en déduit un coût total en $O(n^2)$. $\square$

Remarque. On pourrait améliorer cette complexité en optimisant la recherche du sommet de plus petite étiquette.

Propriété (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 :

- $v_0, \dots, v_{n-1}$ 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 .
- $a_0, \dots, a_{n-1}$ les sommets du graphe rangés par ordre croissant de distance à s . Notamment, $a_0 = s$ et $\delta(a_0) = 0 \leq \delta(a_1) \leq \delta(a_2) \leq \dots \leq \delta(a_{n-1})$.
- $\tilde{a}_0, \dots, \tilde{a}_{n-1}$ 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 $i \in \llbracket 0, n-1 \rrbracket$ 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 $i \in \llbracket 0, n-1 \rrbracket$, on a aussi $e_i(s) = 0$.

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

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

- $\tilde{a}_0 = s = a_0$, $e_0(s) = 0 = \delta(s)$ et $\forall v \neq s$, $e_0(v) = +\infty \geq \delta(v)$. Donc $H(0)$ est vraie.
- Soit $i \in \llbracket 0, n-2 \rrbracket$, on suppose que $\forall k \in \llbracket 0, i \rrbracket$, $H(k)$ est vraie (c'est-à-dire que $H(0)$, $H(1)$, $\dots$, $H(i)$ 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 :

$$s \rightarrow t_1 \rightarrow \dots \rightarrow t_{j-1} \rightarrow t_j \rightarrow a_{i+1}$$

L'avant dernier sommet visité t_j est par définition plus proche de s que ne l'est a_{i+1} , donc $\exists p \in \llbracket 0, i \rrbracket$ tel que $t_j = a_p$ (c'est-à-dire t_j est l'un des sommets $a_0, \dots, a_i$). 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 $s \rightarrow t_1 \rightarrow \dots \rightarrow t_{j-1} \rightarrow a_p$ 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 $e_i(a_{i+1}) \leq \delta(a_{i+1})$. 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 $k \in \llbracket i+2, n-1 \rrbracket$, l'hypothèse de récurrence $H(i)$, l'inégalité $k \geq i+1$ et la définition des a_j donnent :

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

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} :

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

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 :

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

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 $e_{i+1}(v) \geq \delta(v)$ 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 $s \rightarrow \dots \rightarrow a_{i+1} \rightarrow v$ (où $s \rightarrow \dots \rightarrow a_{i+1}$ 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,

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

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 . $\square$

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 $h : S \rightarrow \mathbb{R}$.

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 chemins 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 !

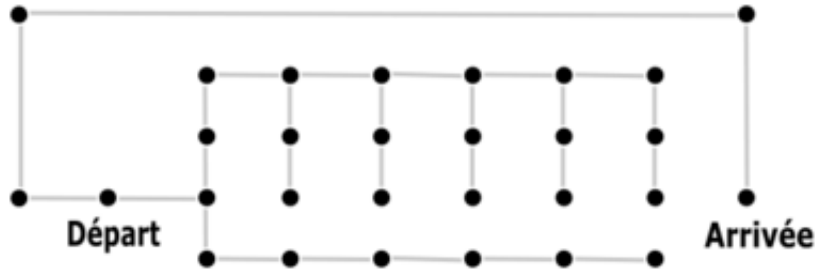


FIGURE 1 – Un exemple où l’heuristique décrite ci-dessus sera inefficace

2 Correction

Définition (Heuristique admissible). Une heuristique est dite **admissible** si elle garantit que l’algorithme A^* renvoie un chemin optimal.

Propriété (Lemme préliminaire). Si un chemin $v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_p$ a une longueur ℓ pour la distance d’origine, alors sa longueur ℓ' vis-à-vis de la pseudo-distance associée à h vaut :

$$\ell' = \ell + h(v_p) - h(v_0).$$

Démonstration. C’est un simple télescopage :

$$\begin{aligned} \ell' &= \sum_{k=0}^{p-1} g' v_k, v_{k+1} \\ &= \sum_{k=0}^{p-1} (g v_k, v_{k+1} + h(v_{k+1}) - h(v_k)) \\ &= \sum_{k=0}^{p-1} g v_k, v_{k+1} + \sum_{k=0}^{p-1} (h(v_{k+1}) - h(v_k)) \\ \ell' &= \ell + h_{v_p} - h_{v_0} \end{aligned}$$

□

Propriété. Si l’heuristique est associée à une fonction h pour laquelle toutes les pseudo-distances sont positives, alors A^* renvoie un chemin optimal (pour la distance d’origine) de s vers a .

De plus, si l’on note $\delta(a)$ la distance optimale renvoyée par Dijkstra et $\delta'(a)$ celle renvoyée par A^* , on a :

$$\delta'(a) = \delta(a) + h(a) - h(s).$$

Démonstration. Si l’on considère l’ensemble des chemins de s à a , le lemme précédent assure que leur longueur vis-à-vis de la distance d’origine est la même que leur longueur vis-à-vis de la pseudo-distance, à la constante additive $h(a) - h(s)$ près. Comme cette constante additive s’applique à tous les chemins, elle ne change rien au problème de minimisation : un chemin qui est de longueur minimale pour la distance d’origine le sera aussi pour la pseudo-distance et réciproquement.

Comme on a supposé les pseudo-distances positives, l’algorithme de Dijkstra fonctionne (la positivité était une hypothèse importante pour sa correction). L’algorithme A^* renvoie donc un chemin optimal, et d’après le lemme, une longueur optimale décalée de la constante $h(a) - h(s)$. □