Chapitre 2 : Parcours de graphes

1 Graphes et accessibilité

1.1 Définitions de base et vocabulaire

Un graphe est un objet mathématique permettant de modéliser une relation binaire sur un ensemble fini. Selon que cette relation est asymétrique ou symétrique, le graphe sera orienté ou non.

Définition 1.1

*Un graphe orienté $G = (S, A)$ est composé d'un ensemble fini S d'éléments appelés **sommets** et d'une partie A de $S \times S$ dont les éléments sont appelés **arcs**.*

Définition 1.2

Si (x, y) est un arc d'un graphe orienté, on dit alors :
- *que x et y sont les **extrémités** de cet arc (x l'origine et y la destination) ;*
- *que x est un **prédécesseur** de y et que x est un **successeur** de y.*

*De plus, si $x = y$, l'arc (x, x) est appelé une **boucle**.*

Exercice de cours 1.3

Représenter graphiquement le graphe $G = (S, A)$ où $S = \{1, 2, 3, 4, 5, 6\}$, $A = \{(1, 2), (2, 4), (2, 5), (4, 1), (4, 4), (4, 5), (5, 4), (6, 3)\}$. Quels sont les successeurs de 2 ? Les prédécesseurs de 5 ?

Définition 1.4

*Un graphe non orienté $G = (S, A)$ est composé d'un ensemble fini S d'éléments appelés **sommets** et d'un ensemble de paires de S dont les éléments sont appelés **arêtes**.*

Exercice de cours 1.5

Représenter graphiquement le graphe $G = (S, A)$ où $S = \{1, 2, 3, 4, 5, 6\}$ et $A = \{\{1, 2\}, \{1, 5\}, \{5, 2\}, \{3, 6\}\}$.

Exercice de cours 1.6

Rappeler quelles sont les deux implémentations concrètes de la structure de graphe.

Remarque 1.7

Étant donné un graphe orienté, sa version non orientée est obtenue en supprimant les boucles et en substituant à chaque arc restant (x, y) la paire $\{x, y\}$[♣]. Réciproquement la version orientée d'un graphe non orienté est obtenue en substituant à chaque arête $\{x, y\}$ les deux arcs (x, y) et (y, x).

♣. Comme il n'y a pas de doublons dans un ensemble, l'arête $\{x, y\}$ est présente au plus une fois, même si les arcs (x, y) et (y, x) étaient tous deux présents dans le graphe initial.

Exercice de cours 1.8

Combien d'arcs a la version orientée du graphe G de l'exercice de cours 1.5 ? Combien d'arêtes a la version non orientée du graphe G de l'exercice de cours 1.3 ?

Définition 1.9

*Soit $G = (S, A)$ un graphe non orienté (resp. orienté). Soit $G' = (S', A')$ un graphe non orienté (resp. orienté). On dit que G' est un **sous-graphe** de G si et seulement si $S' \subseteq S$, et $A' \subseteq A$.*

Exercice de cours 1.10

Donner deux sous-graphes distincts à trois sommets du graphe G de l'exercice de cours 1.5.

Définition 1.11

*Soit $G = (S, A)$ un graphe non orienté (resp. orienté). Soit $W \subseteq S$ un sous-ensemble de sommets. Le **sous-graphe de** G **induit par** W est le graphe $G' = (W, A')$ où $A' = \{\{x, y\} \in A \mid (x, y) \in W^2\})$ (resp. $A' = \{(x, y) \in A \mid (x, y) \in W^2\})$).*

Exercice de cours 1.12

Donner le sous-graphe induit par le graphe G de l'exercice de cours 1.5 sur l'ensemble de sommets $\{2, 4, 6\}$.

Exercice de cours 1.13

Donner un sous-graphe de G de l'exercice de cours 1.5 qui ne soit pas un sous-graphe induit.

Définition 1.14

*Deux sommets distincts d'un graphe orienté $G = (S, A)$ (resp. non orienté) G sont **adjacents** ou **voisins** s'ils sont les extrémités d'un même arc (respectivement d'une même arête), i.e. si $(x, y) \in A$ (resp. $\{x, y\} \in A$).*

*On appelle **voisinage** d'un sommet x l'ensemble des sommets qui lui sont adjacents, et **degré** de x (noté $\deg(x)$) le nombre de tels sommets.*

*Si x est un sommet d'un graphe orienté, on appelle **degré sortant** de x (noté $d_+(x)$) le nombre de successeurs de x, et **degré entrant** de x (noté $d_-(x)$) son nombre de prédecesseurs.*

Exercice de cours 1.15

Donner les sommets adjacents du sommet 2 dans le graphe G de l'exercice de cours 1.5. En déduire le degré du sommet 2. Donner le degré sortant du sommet 4 dans le graphe G de l'exercice de cours 1.3

Exercice de cours 1.16

Que vaut la somme des degrés des sommets d'un graphe non orienté ? Que vaut la somme des degrés entrants (resp. sortants) des sommets d'un graphe orienté ?

Définition 1.17

*Soit $G = (S, A)$ un graphe orienté (resp. non orienté). Un **chemin** (resp. une **chaîne**) est une suite finie non vide de sommets $c = (s_0, \ldots, s_p)$ telle que : $s_0 = x$, $s_p = y$ et $\forall k \in [\![0, p-1]\!], (s_k, s_{k+1}) \in A$ (resp. $\{s_k, s_{k+1}\} \in A$).*

*La **longueur** c est p, c'est le nombre d'arcs (resp. d'arêtes) empruntées par ce chemin (resp. cette chaîne) comptés avec multiplicité.*

*On dit qu'un chemin (resp. une chaîne) $c = (s_0, \ldots, s_p)$ est **élémentaire** si et seulement si ses sommets sont deux à deux distincts, i.e. $\forall (i, j) \in [\![0, p]\!]2, s_i = s_j \Rightarrow i = j$.*

Définition 1.18

*Soit $G = (S, A)$ un graphe non orienté(resp. non orienté). Un chemin (resp. une chaîne) $s_0 = x$, $s_p = y$ est un **circuit** (resp. un **cycle**) $s_0 = s_p$ et $p \geqslant 1$ (resp.$p \geqslant 3$).*
*On dit d'un tel circuit (resp. d'un tel cycle) qu'il est **élémentaire** si et seulement si le seul sommet répété est son extrémité, i.e. $\forall (i, j) \in [\![0, p-1]\!]^2, s_i = s_j \Rightarrow i = j$.*

Remarque 1.19

La définition ci-dessus de cycle fixe une origine : s_0. Néanmoins on assimilera souvent comme étant un même cycle les suites $s_0, s_1 \dots s_p$ et $s'_0, s'_1 \dots s'_p$ s'il existe $k \in [\![1, p-1]\!]$ tel que $\forall i \in [\![0, p]\!], s'_i = s_{i+k \mod p}$.

Exercice de cours 1.20

Dans le graphe G de l'exercice de cours 1.3, donner un chemin de longueur 5, un circuit de longueur 6 et un circuit élémentaire de longueur 3.

Exercice de cours 1.21

Soit un graphe non orienté $G = (S, A)$ à n sommets. Montrer qu'il n'admet pas de chaîne élémentaire de longueur supérieure à n.

Définition 1.22

Soit $G = (S, A)$ un graphe non orienté (resp. orienté). On dit que :
- *G est **acyclique** dès lors qu'il n'existe aucun cycle élémentaire ;*
- *G est **complet** dès lors que $A = \{\{x, y\} \mid (x, y) \in A^2\}$ (resp. $A = S \times S$)*

Soit $G = (S, A)$ un graphe non orienté, on dit que :
- *G est **biparti** dès lors qu'il existe une partition $\{W_1, W_2\}$ de ses sommets tel que $\forall \{u, v\} \in A, (u, v) \in W_1 \times W_2 \cup W_2 \times W_1$;*
- *G est un **arbre** si et seulement si G est connexe♣ et acyclique.*

Exercice de cours 1.23

Soit un graphe orienté $G = (S, A)$. Montrer que si G est acyclique alors il admet un sommet sans prédécesseur (resp. sans successeur).

Exercice de cours 1.24

Dessiner le graphe complet à 5 sommets.

1.2 Accessibilité dans le cas non orienté et connexité

Dans toute cette section, on travaille sur un graphe non orienté $G = (S, A)$. On notera $n = \text{card}(S)$ et $m = \text{card}(A)$. On supposera que $n > 0$.
On définit sur S la relation binaire $\sim$ par $\forall (u, v) \in S^2, u \sim v$ si et seulement s'il existe une chaîne reliant u à v.

Remarque 1.25

Dès que plusieurs graphes seront en jeu, on indicera les relations par le graphe concerné afin de lever toute ambiguïté, par exemple on notera $u \sim_G v$ ou $u \sim_{G'} v$.

Proposition 1.26

La relation $\sim$ est une relation d'équivalence.

Exercice de cours 1.27

Démontrer que $\sim$ est bien une relation d'équivalence.

Remarque 1.28

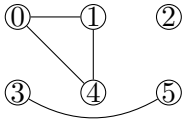
Pour les graphes non orientés, la relation binaire $\sim$ peut être définie comme la clôture reflexive transitive de la relation $\sim$ sur S définie par A, *i.e.* telle que $\forall (u,v) \in S^2,\ u \sim v \leftrightarrow u,v \in A$.

Définition 1.29

*Soit $\sim$ la relation définie ci-avant. On appelle **composantes connexes** de G les classes d'équivalences de $\sim$. Elles forment une partition de S* ♣.

Exercice de cours 1.30

Donner les composantes connexes du graphe ci-dessous.



Définition 1.31

Soit $W \subseteq S$,

- *le graphe G est dit **connexe** si et seulement s'il admet une seule composante connexe ;*
- *l'ensemble W est dit **connexe** si et seulement le sous-graphe induit par G sur W est connexe.*

Exercice de cours 1.32

Donner l'ensemble des ensembles connexes du graphe de l'exercice de cours 1.30.

Remarque 1.33

De manière équivalente on peut dire qu'un ensemble de sommets $W \subseteq S$ est connexe si et seulement si les sommets de W sont deux à deux reliés par une chaîne de G n'empruntant que des sommets de W.

Exercice de cours 1.34

Donner un exemple d'ensemble dont les sommets sont 2 à 2 reliés par une chaîne du graphe mais qui ne forme pas un ensemble connexe.

Proposition 1.35

Soit $W \subseteq S$, W est une composante connexe de G si et seulement si W est connexe maximale (au sens de $\subseteq$).

Exercice de cours 1.36

Démontrer le résultat précédent. On pourra s'inspirer de la démonstration du résultat similaire, dans le cas orienté, se trouvant dans la section suivante (*Cf.* 1.48).

1.3 Accessibilité dans le cas orienté et forte connexité

Dans toute cette section, on travaille sur un graphe orienté sans boucle $G = (S, A)$. On notera $n = \mathrm{card}(S)$ et $m = \mathrm{card}(A)$. On supposera que $n > 0$.

Définition 1.37

*Pour $(u, v) \in S^2$, on dit que v est **accessible** depuis u si et seulement s'il existe un chemin de u à v dans G.*

Remarque 1.38

On note parfois $u \xrightarrow{*} v$ pour signifier "v est accessible depuis u". Cette notation est justifiée par le fait que $\xrightarrow{*}$ est la clôture réflexive transitive de la relation $\rightarrow$ définie par A.
On parlera parfois des **descendants** de u pour parler des sommets v tels que $u \xrightarrow{*} v$ et des **descendants propres** pour parler des sommets v tels que $u \xrightarrow{*} v$ et $v \not\xrightarrow{*} u$.

Dans la suite, on note $\sim$ la relation binaire sur S définie par $\forall (u, v) \in S^2$, $u \sim v$ si et seulement si u est accessible depuis v **et** v est accessible depuis u.

Proposition 1.39

La relation $\sim$ est une relation d'équivalence.

Exercice de cours 1.40

Démontrer le résultat précédent.

Remarque 1.41

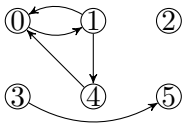
Contrairement à ce qu'on avait pu faire pour les graphes non orientés, la relation binaire $\sim$ n'est pas la clôture transitive de la relation $\rightleftarrows$ définie sur S par $\forall (u, v) \in S^2$, $u \rightleftarrows v$ dès lors $(u, v) \in A$ et $(v, u) \in A$.
En effet considérons le contre exemple ci-contre. Il y a une unique CFC contenant les 3 sommets mais il n'existe aucun couple de sommets x et y tels que $x \rightleftarrows y$.

Définition 1.42

*On appelle **composantes fortement connexes** (CFC) de G les classes d'équivalences de $\sim$.*

Exercice de cours 1.43

Donner les composantes fortement connexes du graphe ci-dessous.



Définition 1.44

Soit $W \subseteq S$,

- *le graphe G est dit **fortement connexe** si et seulement s'il admet une seule composante fortement connexe ;*
- *l'ensemble W est dit **fortement connexe** si et seulement le sous-graphe induit par G sur W est fortement connexe.*

Remarque 1.45

Un ensemble de sommets $W \subseteq S$ est connexe si et seulement si les sommets de W sont deux à deux reliés par une chaîne de G.

Exercice de cours 1.46

Donner l'ensemble des ensembles fortement connexes du graphe de l'exercice de cours 1.30.

Proposition 1.47

Si W est une composante fortement connexe d'un graphe G alors W est fortement connexe.

Démonstration : Soit W une composante fortement connexe de G. Soient u et v deux sommets de W.

Par définition de CFC, W est une classe d'équivalence de $\sim_G$, donc $u \sim v$. Ainsi il existe dans G :

- un chemin γ_1 de u à v qu'on note $u \to_G u_1 \to_G \ldots \to_G u_{p-1} \to_G v$;
- un chemin γ_2 de v à u qu'on note $v \to_G v_1 \to_G \ldots \to v_{q-1} \to_G u$.

Pour tout $i \in [\![1, n[\![$ on a alors $u_i \sim_G v$, car il existe dans G

- un chemin de u_i à v, à savoir $u_i \to_G u_{i+1} \to_G \ldots \to_G v$;
- un chemin de v à u_i à savoir $v \to_G v_1 \to_G \ldots \to_G v_{q-1} \to_G \boldsymbol{u} \to u_1 \to_G \ldots \to_G u_{i-1} \to u_i$

Ainsi pour tout $i \in [\![1, p[\![$, $u_i \in W$. De même pour tout $i \in [\![1, q[\![$ on a $v_i \sim_G v$ donc $v_i \in W$. Ainsi les chemins γ_1 et γ_2 sont aussi des chemins du graphe induit par W, $G_W = (W, A \cap W^2)$, on a donc $u \xrightarrow{*}_{G_W} v$ et $v \xrightarrow{*}_{G_W} w$, et donc $u \sim_{G_W} v$.

Ceci étant vrai pour tout couple de sommets $(u, v) \in W^2$, on en déduit que G_W admet une unique composante fortement connexe, et donc que W est fortement connexe par définition. □

Proposition 1.48

Les composantes fortement connexes sont les ensembles de sommets fortement connexes maximaux (au sens de $\subseteq$).

Démonstration :

- Soit un ensemble $V \subseteq S$ fortement connexe et maximal pour l'inclusion. On a $V \neq \emptyset$, en effet n'importe quel singleton $\{s\}$ avec $s \in S$ est fortement connexe donc un sur-ensemble fortement connexe de $\emptyset$[♣]. Puisque V est fortement connexe, le graphe qu'il induit, $G_V = (V, A \cap V^2)$, est fortement connexe. Autrement dit V admet une unique classe pour la relation $\sim_{G_V}$. Ainsi pour tout $(u, v) \in V^2$ on a $u \sim_{G_V} v$, et comme les arcs de G_V sont des arcs de G on a aussi $u \sim_G v$.
  Ainsi V est non vide et ne contient que des éléments deux à deux équivalents pour $\sim_G$, on peut donc considérer W la classe d'équivalence des éléments de V pour $\sim_G$. Par construction, W est une CFC de G, et donc en particulier W est fortement connexe d'après la propriété précédente. Or on a $V \subseteq W$, donc par maximalité de V on en déduit $V = W$, ainsi V est bien une CFC de G.
- Réciproquement considérons V une CFC de G. D'après la propriété précédente, V est alors un ensemble fortement connexe. Par l'absurde supposons qu'il existe W un sur-ensemble strict de V, *i.e.* $V \subsetneq W$, fortement connexe. On peut alors considérer un sommet $w \in W \setminus V$. Considérons aussi u un sommet de V (une CFC est nécessairement non vide) et donc de W. W étant fortement connexe, le graphe $G_W = (W, A \cap W^2)$ est fortement connexe, ainsi il existe dans G_W un chemin de u à w et un chemin de w à u. Puisque les arcs de G_W sont des arcs de G, ces chemins existent aussi dans G, ainsi $u \sim_G w$. Or $u \in V$ et V est une classe d'équivalence de $\sim_G$ en tant que CFC de G, donc $w \in V$ par transitivité. Absurde.
  Ainsi V n'admet est bien fortement connexe maximal.

□

♣. Cet argument est concluant parce que $S \neq \emptyset$

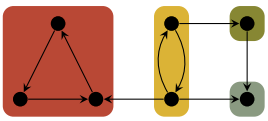
Définition 1.49

On appelle ***graphe réduit*** *de G le graphe orienté $\widehat{G} = (\widehat{S}, \widehat{A})$ où $\widehat{S}$ est l'ensemble des CFC de G et*

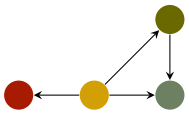
$$\widehat{A} = \left\{ (C_1, C_2) \in \widehat{S}^2 \,\middle|\, C_1 \neq C_2,\ \exists (s_1, s_2) \in C_1 \times C_2, (s_1, s_2) \in A \right\}.$$

Autrement dit le graphe réduit s'obtient en fusionnant les sommets équivalents pour $\sim$, et en supprimant les éventuelles boucles apparues lors de ces fusions.

Exemple 1.50



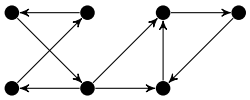
Avant réduction



Après réduction

Exercice de cours 1.51

Donner le graphe réduit du graphe ci-dessous.



Lemme 1.52

S'il existe un chemin de C_i à C_j dans $\widehat{G}$, alors il existe un chemin dans G de n'importe quel sommet de C_i à n'importe quel sommet de C_j.

Exercice de cours 1.53

Démontrer le lemme précédent.

Proposition 1.54

Un graphe réduit est acyclique.

Démonstration : Corollaire direct du lemme : s'il y avait un circuit dans $\widehat{G}$, il existerait deux classes distinctes C_i et C_j telles que dans $\widehat{G}$ il existe un chemin de C_i à C_j et de C_j à C_i. D'après le lemme, pour tout $s \in C_i$ et $t \in C_j$ il existe alors dans G un chemin de s à t et un chemin de t à s. Ainsi $t \sim s$, donc s et t sont dans la même CFC, soit $C_i = C_j$. Absurde. □

Proposition 1.55

Si G est sans circuit alors :

- *chacun de ses sommets est seul dans sa CFC ;*
- *le graphe réduit de G est G lui-même.*

Exercice de cours 1.56

Démontrer la propriété précédente 1.55.

2 Parcours de graphes

2.1 Bordure et parcours : définitions

Dans cette section on travaille sur un graphe orienté $G = (S, A)$ à n sommets, mais les définitions et résultats se transposent facilement au cas non orienté. On prendra cependant garde à la différence entre les propositions 2.13 et 2.14.

On appelle **permutation des sommets** une famille finie de sommets de S dans laquelle chaque sommet apparaît exactement une fois. Mathématiquement un tel objet est une bijection de $[\![1, n]\!]$ dans S, autrement dit c'est juste une numérotation des sommets.

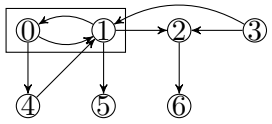
Définition 2.1

*La **bordure** d'un ensemble de sommets $V \subseteq S$, notée $\mathcal{B}(V)$, est l'ensemble des successeurs des sommets de V n'étant pas eux-mêmes dans V.*

$$\mathcal{B}(V) = \{u \in S \setminus V \mid \exists v \in V, (v, u) \in A\}$$

Exercice de cours 2.2

Donner la bordure de l'ensemble encadré dans le graphe ci-dessous.



Définition 2.3

*Un **parcours** $(L_i)_{i \in [\![1,n]\!]}$ de G est une permutation des sommets telle que :*

$$\forall i \in [\![1, n]\!],\ L_i \in \mathcal{B}(\{L_j \mid j \in [\![1, i[\![\}) \quad \textit{ou} \quad \mathcal{B}(\{L_j \mid j \in [\![1, i[\![\}) = \emptyset$$

Autrement dit, un parcours est une permutation dans laquelle un sommet n'apparaît que s'il est dans la bordure des sommets précédents, à moins que celle-ci ne soit vide.

Définition 2.4

Soit $L = (L_i)_{i \in [\![1,n]\!]}$ un parcours de G et $i \in [\![1, n]\!]$.
*On dit que L_i est un **point de régénération** de L dès lors que $\mathcal{B}(\{L_j \mid j \in [\![1, i[\![\}) = \emptyset$.*

Remarque 2.5

Le premier sommet d'un parcours est toujours un point de régénération car $\mathcal{B}(\emptyset) = \emptyset$.

Exercice de cours 2.6

Donner deux parcours distincts du graphe de l'exercice de cours 2.2. Souligner leurs points de régénération.

Lemme 2.7

Si $V \subseteq S$ est tel que $\mathcal{B}(V) = \emptyset$, alors il n'existe aucun chemin allant d'un sommet de V vers un sommet de $S \setminus V$.

Démonstration : Si un tel chemin existait, il passerait nécessairement par un arc (a,b) allant de $a \in V$ vers $b \in S \setminus V$, ainsi b serait un successeur de $a \in V$ tel que $b \notin V$, donc b serait dans $\mathcal{B}(V)$ ce qui nierait sa vacuité. ABSURDE. □

Remarque 2.8

Attention il peut exister des chemins allant de $S \setminus V$ vers S car la bordure ne regarde que les successeurs, pas les prédécesseurs.

2.2 Partition associée à un parcours

Les points de régénération "découpent" le parcours. On associe donc à un parcours le partitionnement de l'ensemble des sommets induit par ce découpage : deux sommets sont dans la même partie dès lors qu'ils se trouvent entre les deux mêmes points de régénération. C'est le sens de la définition suivante.

Définition 2.9

Soit $L = (L_i)_{i \in [\![1,n]\!]}$ un parcours de G. On note K son nombre de points de régénération, et $(r_k)_{k \in [\![1,K]\!]} \in [\![1,n]\!]^K$ la suite strictement croissante des indices des points de régénération de L ♣. En notant de plus $r_{K+1} = n+1$, on peut définir ***la partition associée au parcours L*** *comme étant $\{C_k \mid k \in [\![1,K]\!]\}$ où pour tout $k \in [\![1,K]\!]$, $C_k = \{L_j \mid j \in [\![r_k, r_{k+1}[\![\}$.*

Exercice de cours 2.10

Donner le partitionnement associé au parcours $(2,6,5,1,0,4,3)$ du graphe de l'exercice de cours 2.2.

Lemme 2.11

Soit $L = (L_i)_{i \in [\![1,n]\!]}$ un parcours de G. Soit $(p,q) \in [\![1,n]\!]^2$ tel que $p < q$.
S'il existe un chemin de L_p à L_q, alors aucun des L_k pour $k \in]\!]p,q]\!]$ n'est point de régénération.

Démonstration : Soit $(p,q) \in [\![1,n]\!]^2$ tels que $p < q$. Supposons qu'il existe un chemin de L_p à L_q. Par l'absurde supposons qu'il existe aussi $k \in]\!]p,q]\!]$ tel que L_k est un point de régénération. Par définition $\mathcal{B}(\{L_j \mid j \in [\![1,k[\![\}) = \emptyset$. On déduit donc du lemme 2.7 qu'il n'existe aucun chemin allant d'un sommet de $\{L_j \mid j \in [\![1,k[\![\}$ vers un sommet de $\{L_j \mid j \in [\![k,n]\!]\}$. Le chemin de L_p à L_q contredit ce fait. ABSURDE. □

Lemme 2.12

Soit $L = (L_i)_{i \in [\![1,n]\!]}$ un parcours de G. Soit L_r pour $r \in [\![1,n]\!]$ un point de régénération de L.
Notons C_r la partie associée à ce point de régénération dans le partition associée à L.
Tout sommet de C_r est accessible depuis L_r par un chemin dont tous les sommets sont dans C_r.

Démonstration : Par définition de la partition associée à un parcours, il existe $q \in [\![1,n]\!]$ tel que $C_r = \{L_r, L_{r+1}, \ldots, L_q\}$. On peut alors montrer par récurrence forte et finie sur $k \in [\![r,q]\!]$ la propriété suivante.

$$H_k : \text{il existe un chemin de } L_r \text{ vers } L_k \text{ dont tous les sommets sont dans } C_r$$

- Si $k = r$ alors le chemin réduit au sommet L_r convient, ainsi H_r est vraie.
- Soit $k \in [\![r,q[\![$. Supposons que $\forall i \in [\![r,k]\!], H_i$ est vraie. Montrons H_{k+1}. Puisque $L_{k+1} \in C_r$ et $k+1 \neq r$, L_{k+1} n'est pas un point de régénération (par définition de la décomposition associée à un parcours). Ainsi $\mathcal{B}(\{L_j \mid j \in [\![1,k+1[\![\}) \neq \emptyset$, donc par définition d'un parcours $L_{k+1} \in \mathcal{B}(\{L_j \mid j \in [\![1,k+1[\![\})$ nécessairement. Comme $\mathcal{B}(\{L_j \mid j \in [\![1,r[\![\} = \emptyset$ (car L_r est point de régénération), on en déduit que L_{k+1} est successeur d'un sommets de $L_r, L_{r+1} \ldots L_k$. Il existe donc $i_0 \in [\![r,k]\!]$ tel que $(L_{i_0}, L_{k+1}) \in A$.

♣. Ainsi les points de régénération sont les $(L_{r_k})_{k \in [\![1,K]\!]}$.

Or d'après H_{i_0}, il existe un chemin de L_r à L_{i_0} dont tous les sommets sont dans C_r. En composant ce chemin avec l'arc (L_{i_0}, L_{k+1}), on obtient bien un chemin de L_r à L_{k+1} dont tous les sommets sont dans C_r. Ainsi H_{k+1} est vraie.

□

Proposition 2.13

Dans un graphe non orienté, la partition associée à un parcours quelconque est la décomposition en composantes connexes.

Démonstration : C'est un corollaire des deux lemmes précédents.

En effet si deux sommets sont dans une même classe $\mathcal{C}$ du parcours, d'après le lemme 2.12 (dans sa version non orientée) ils sont reliés par une chaîne qui ne passe que par des sommets de cette classe. Ainsi une classe du parcours est un ensemble de sommets connexe. De plus il est connexe maximal, car les sommets des autres classes ne peuvent être reliés à ces sommets par contraposée du lemme 2.11 (en s'appuyant sur un point de régénération qui sépare $\mathcal{C}$ de l'autre sommet dans le parcours). □

Proposition 2.14

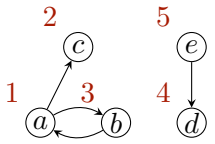
Dans un graphe orienté, un parcours offre une partition dont la décomposition en CFC est un raffinement. Autrement dit les parties associées à un parcours sont toujours des unions de CFC.

Démonstration : C'est un corollaire du lemme 2.11.

Il suffit de montrer qu'une même CFC ne s'étale pas sur plusieurs parties du parcours. Soient L_i et L_j deux sommets deux sommets d'une même CFC. Quitte à les échanger on peut supposer que $i < j$. Par définition de CFC, il existe en particulier un chemin de L_i à L_j. Donc d'après le lemme 2.11, aucun des L_k pour $k \in [\![i, j]\!]$ n'est point de régénération. Par définition de la partition associée à un parcours, cela signifie que L_i et L_j sont nécessairement dans la même partie du parcours. □

Exemple 2.15

Le parcours $(\underline{a}, c, b, \underline{d}, \underline{e})$ (dont on a souligné les points de régénération) du graphe ci-dessous conduit à la partition $\{\{a, c, b\}, \{d\}, \{e\}\}$ de S. On remarque que les composantes fortement connexes sont $\{a, b\}$, $\{c\}$, $\{d\}$ et finalement $\{e\}$.



2.3 Parcours particuliers

Définition 2.16

Soit $L = (L_i)_{i \in [\![1,n]\!]}$ un parcours de G.

Pour $k \in [\![1, n]\!]$ et $i \in [\![1, k[\![$, on dit que L_i est ***ouvert à l'étape*** *k dès lors que $\mathrm{Succ}(L_i) \not\subseteq \{L_j \mid j \in [\![1, k[\![\}$ ♣, autrement dit si L_i "contribue" à la bordure de $\{L_j \mid j \in [\![1, k[\![\}$.*

Exercice de cours 2.17

Donner, **à chaque étape** k **du parcours** l'ensemble des sommets ouverts à l'étape k lors du parcours $(2, 6, 5, 1, 0, 4, 3)$ du graphe de l'exercice de cours 2.2.

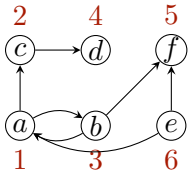
♣. $\mathrm{Succ}(L_i)$ est l'ensemble des successeurs du sommet L_i

Définition 2.18

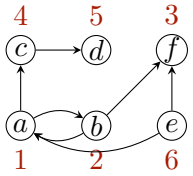
Soit $L = (L_i)_{i\in[\![1,n]\!]}$ un parcours de G.

- *On dit que L est un **parcours en largeur** de G dès lors que chaque sommet du parcours qui n'est pas point de régénération est successeur du **premier** sommet ouvert à cette étape, i.e. $\forall k \in [\![1,n]\!], \mathcal{B}(\{L_j \mid j \in [\![1,k[\![\}) = \emptyset$ ou $L_k \in Succ(L_{i_0})$ avec $i_0 = \min\{i \in [\![1,k[\![\mid L_i$ ouvert à l'étape $k\}$.*
- *On dit que L est un **parcours en profondeur** de G dès lors chaque sommet du parcours qui n'est pas point de régénération est successeur du **dernier** sommet ouvert à cette étape, i.e. $\forall k \in [\![1,n]\!], \mathcal{B}(\{L_j \mid j \in [\![1,k[\![\}) = \emptyset$ ou $L_k \in Succ(L_{i_0})$ avec $i_0 = \max\{i \in [\![1,k[\![\mid L_i$ ouvert à l'étape $k\}$.*

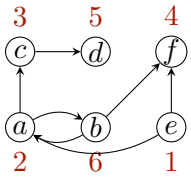
Exemple 2.19



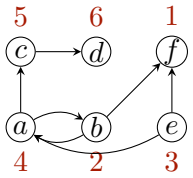
Parcours largeur



Parcours profondeur



Parcours ni largeur ni profondeur



Parcours largeur et profondeur

Exercice de cours 2.20

Justifier les affirmations de l'exemple 2.3.

Exercice de cours 2.21

Donner une forme de graphe orienté à n sommets pour lequel tous les parcours sont à la fois des parcours en largeur et en profondeur.

2.4 Forêt associée à un parcours

Dans un parcours L, les sommets qui ne sont pas point de régénération apparaissent dans le parcours après un de leur prédécesseur. Une forêt associée à un parcours représente le choix, pour chaque sommet non point de régénération, d'un père, à savoir un prédécesseur qui le précède dans le parcours. L'ensemble de tels arcs père-sommet forme un graphe acyclique ♣, c'est pourquoi on parle de forêt.

Exercice de cours 2.22

Combien d'arcs a la forêt associée à un parcours d'un graphe à n sommets qui a p points de régénération ?

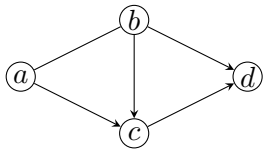
Dans le cas d'un parcours en largeur (resp. en profondeur), le père d'un sommet est fixé pour chaque sommet L_k (non point de régénération) comme étant le premier (resp. dernier) sommet ouvert à l'étape k. Aussi la forêt associé à un tel parcours est unique.

♣. les arcs sont de la forme (L_i, L_j) avec $i < j$, assurant la stricte croissance des indices le long d'un chemin

Exercice de cours 2.23

On considère le parcours $[a, b, c, d]$ du graphe ci-dessous.

- Donner la forêt associée à ce parcours vu comme un parcours en largeur.
- Donner la forêt associée à ce parcours vu comme un parcours en profondeur.
- Donner une autre forêt associée à ce parcours vu comme un parcours quelconque.



Remarque 2.24

La forêt associée à un parcours en largeur (resp. profondeur) est aussi une forêt associée à ce parcours vu comme un parcours quelconque, mais la réciproque n'est pas toujours vraie.

Remarque 2.25

On peut définir formellement la notion de **forêt associée** à un parcours L de $G = (S, A)$ comme un sous-graphe (S, A') où l'ensemble d'arcs A' associe un père à chaque sommet non point de régénération dans L.

- Si L est un parcours en largeur (resp. en profondeur), pour tout $k \in [\![1, n]\!]$ tel que L_k n'est pas point de régénération, on peut associer à k l'indice $\pi(k)$ du premier (resp. dernier) sommet ouvert à l'étape k de L, ainsi $\pi(k) < k$ et $(L_{\pi(k)}, L_k)$ est un arc de G. Dans ce cas la forêt associée au parcours en largeur (resp. en profondeur) L est définie par l'ensemble A' ci-dessous.
- Si L est un parcours quelconque, pour tout $k \in [\![1, n]\!]$ tel que L_k n'est pas point de régénération, on peut associer à k un indice $\pi(k) < k$ tel que $L_k \in \mathrm{Succ}(L_{\pi(k)})$, *i.e.* tel que $(L_{\pi(k)}, L_k)$ est un arc de G. Dans ce cas une forêt associée au parcours L est définie par l'ensemble A' ci-dessous.

$$A' = \{(L_{\pi(k)}, L_k) \mid L_k \text{ n'est pas point de régénération }\}$$

2.5 Implémentation de parcours

Attention : L'algorithme 1 donné ci-dessous a un intérêt purement pédagogique, il ne doit **en aucun cas** être implémenté directement pour calculer un parcours.

Algorithme 1 : Parcours naïf

Entrée : Un graphe $G = (S, A)$
Sortie : Un parcours de G

```
1  n ← card(S) ;
2  L ← liste vide ;
3  Visités ← ∅ ;
4  pour k = 1 à n faire
5      si B(L) ≠ ∅ alors
6          Choisir un sommet u dans B(L) ;
7      sinon
8          Choisir un sommet u quelconque de S \ Visités ;
9      Ajouter u à Visités ;
10     L ← L · u ;
11 retourner L
```

Exercice de cours 2.26

Donner un invariant liant les variables L et Visités dans l'algorithme 1.

Proposition 2.27

Les parcours de $G = (S, A)$ sont exactement les listes pouvant être fabriquées par l'algorithme 1.

On remarque que $\mathcal{B}(L)$ évolue peu entre chaque tour, puisqu'un seul élément est ajouté à L. Tirant profit de cette observation, on raffine l'algorithme précédent en l'algorithme 2.

Algorithme 2 : Parcours avec ensemble todo

```
   Entrée : Un graphe G = (S, A)
   Sortie : Un parcours de G
 1 L ← liste vide ;
 2 Visités ← ∅ ;
 3 Procedure ExploreDescendants :
       Hypothèse : B(Visités) = ∅
       Entrée : s ∈ S \ Visités
       Effet : Ajoute à L et à Visités s et tous ses descendants, de manière à ce que L reste un
               parcours partiel ♡.
 4     todo ← {s} ;
 5     tant que todo ≠ ∅ faire
 6         Choisir un sommet u ∈ todo ;
 7         Ôter u de todo ;
 8         si u ∉ Visités alors
 9             Ajouter u à Visités ;
10             L ← L · u ;
11             pour tout v ∈ succ(u) faire
12                 Ajouter v à todo ;
13 tant que S\Visités ≠ ∅ faire
14     s ← un sommet de S\Visités ;
15     ExploreDescendants(s) ;
16 retourner L ;
```

Exercice de cours 2.28

Donner un invariant liant $\mathcal{B}(L)$, todo et Visités dans l'algorithme 2.

Exercice de cours 2.29

Comment modifier l'algorithme 2, afin de réaliser efficacement le test ligne 13 ?

Proposition 2.30

Si le graphe G est représenté par table de listes d'adjacence♣, l'algorithme 2 effectue $\mathcal{O}(n + m)$ itérations de la boucle ligne 5 et $\mathcal{O}(m)$ itérations de la boucle ligne 11.

Démonstration : Ces complexités découlent du fait que la boucle tant que de la ligne 5 est exécutée une fois pour chaque ajout d'un élément dans todo, en effet chaque tour de cette boucle extrait un élément de todo. Estimons donc le nombre de fois qu'un sommet est ajouté à todo.

Remarquons au préalable que le corps de la conditionnelle **si** ligne 8 est exécuté une seule fois par sommets du graphe. En effet, pour un sommet $u \in S$ donné, u n'est initialement pas dans Visités, y est ajouté si le corps du **si** ligne 8 est exécuté pour u, et ne peut par la suite plus en être retiré, ce qui exclut d'entrer à nouveau dans le **si** ligne 8.

♡. Pour les besoins des énoncés, on dira ici qu'une séquence de sommets L est un parcours partiel de G si L est un parcours du sous-graphe de G induit par l'ensemble des sommets figurant dans L.

- Chaque point de régénération est ajouté dans todo une fois ligne 4 ;
- Un sommet v est ajouté dans todo ligne 12 à chaque exécution de la boucle ligne 8 telle que u est un prédécesseur de v, ainsi on en déduit que chaque sommet v est ajouté $d_-(v)$ fois à la ligne 12.

Finalement par sommation $\sum_{v\in S}(1+d_-(v)) = \mathcal{O}(n+m)$.

Il est important de remarquer ici que l'algorithme de parcours ne se contente pas de visiter une fois chaque sommet, mais visite en fait aussi une fois chaque arc du graphe. □

Remarque 2.31

Intéressons nous aux structures de données concrètes usuellement utilisées pour implémenter les objets todo et Visités.

- Les opérations requises pour l'objet todo sont l'ajout et l'extraction, aussi peut-on se contenter d'une structure de pile ou de file pour l'implémenter. Les lignes 4, 5, 6, 7 et 12 se font alors en $\mathcal{O}(1)$.
- Les opérations requises pour l'objet Visités sont le test d'appartenance et l'ajout. On reconnaît le type de données abstrait Ensemble. Dans le cas, usuel, où l'ensemble S des sommets peut être assimilé à $[\![0, n-1]\!]$ on utilise un tableau de booléens indicé par $S = [\![0, n-1]\!]$ pour implémenter Visités. Les lignes 8 et 9 se font alors en $\mathcal{O}(1)$.

Dans le cas où $S = [\![0, n-1]\!]$, le cumul des recherches des points de régénération (ligne 14) peut se faire en $\mathcal{O}(n)$♣. Ainsi **on retiendra** que, sous ces hypothèses, la complexité d'un parcours de graphe est en $\mathcal{O}(n+m)$.

Remarque 2.32

Le choix d'une implémentation de todo au moyen d'une pile (resp. d'une file) conduit à un parcours en profondeur (resp. largeur).

2.6 Implémentations en OCaml

Dans toute cette section, on suppose que l'ensemble des sommets du graphe est $[\![0, n-1]\!]$ et qu'il est représenté par une table de listes de successeurs. Ainsi le type utilisé (en OCaml) est le suivant.

```
1 type graphe = int list array
```

Par programmation impérative. On fournit ci-dessous une implémentation en OCaml de l'algorithme de parcours présenté dans la section précédente. L'utilisation du module `Stack` pour la gestion de l'ensemble todo assure un parcours en profondeur du graphe. Si on remplace l'utilisation du module `Stack` par un module `Queue`, on obtient un parcours en largeur.

```
1  let parcours_avec_todo (g: graphe): int list =
2    (* nombre de sommets dans le graphe *)
3    let n       = Array.length g in
4    (* parcours que nous sommes en train de fabriquer *)
5    let l       = ref [] in
6    (* un sommet i est déjà traité et apparaît ds l ssi visité.(i) *)
7    let visites = Array.make n false in
8    let explore_descendants (s: int) : unit =
9      (* todo list *)
10     let todo = Stack.create () in
11     (* on ajoute s dans la pile des sommets à traiter *)
```

♣. on le verra dans la section suivante

```
    Stack.push s todo;
    while (not (Stack.is_empty todo)) do
      (* on extrait un élément dans la pile des sommets à traiter *)
      let u = Stack.pop todo in
      (* on teste si u a déjà été visité *)
      if not (visites.(u)) then
        (* si ce n'est pas le cas on le visite *)
        begin
          (* on l'ajoute aux visités pour ne pas le revisiter plus tard *)
          visites.(u) <- true;
          (* on ajoute u au parcours *)
          l := u :: !l;
          (* on ajoute les successeurs de u dans todo *)
          List.iter (fun v -> Stack.push v todo) g.(u)
        end
    done
  in

  (* fonction auxiliaire retournant l'indice du premier sommet non visités
     d'indice supérieur à j < n, n s'il n'en existe pas *)
  let trouve_non_visites (j: int) : int =
    let i = ref j in
    while (!i < n && visites.(!i)) do
      incr i
    done;
    !i
  in

  let s = ref 0 in
  (* tant qu'il existe un sommet s non encore visité *)
  while (!s < n) do
    (* on visite s et ses descendants *)
    explore_descendants !s;
    (* on calcule un nouveau sommet s non encore visité *)
    s := trouve_non_visites (!s + 1)
  done;
  List.rev !l

```

En utilisant la récursivité. Dans le cas d'un parcours en profondeur, on peut utiliser la pile des appels récursifs pour gérer les sommets à visiter plutôt qu'une pile explicite. En remplaçant la fonction `explore_descendants` de l'implémentation précédente par la fonction récursive ci-dessous, on obtient une nouvelle implémentation de parcours **en profondeur**.

```
  let rec explore_descendants (s: int) : unit =
    (* on teste si u a déjà été visité *)
    if not (visites.(s)) then
      (* si ce n'est pas le cas on le visite *)
      begin
        (* on l'ajoute aux visités pour ne pas le revisiter plus tard *)
        visites.(s) <- true;
```

```
        (* on ajoute s au parcours *)
        l := s :: !l;
        (* on visite les successeurs de s au moyen d'appels récursifs *)
        List.iter (fun v -> explore_descendants v) g.(s)
      end
```

Exercice de cours 2.33

> Comme annoncé précédemment (Remarque 2.31), nous avons ici précisé comment effectuer la recherche des points de régénération. Donner le coût cumulé de ces recherches dans les implémentations ci-dessus.

Remarque 2.34

> Dans cette section, nous avons fourni une implémentation "générique" de parcours : on se contente de calculer une liste de sommets qui forme un parcours. Il faut être capable d'adapter cet algorithme pour répondre à divers problèmes, par exemple : le langage d'un automate est-il vide ? Tous les sommets sont-ils accessibles depuis un certain sommet ? Combien de composantes connexes a ce graphe non orienté ? Donner un circuit de ce graphe orienté ….

3 Interlude : graphes implicites

Les algorithmes et structures de données présentées dans ce chapitre supposent que les graphes manipulés sont des graphes pour lesquels nous avons accès aux ensembles de sommets et d'arcs♣. Il est toutefois fréquent de devoir manipuler des graphes pour lesquels cette hypothèse n'est pas vérifiée. Considérons par exemple le graphe du Web, dont les sommets sont les pages Web (identifiées par leur URL) et dont les arcs correspondent à des liens hypertextes. Les données qui définissent ce graphe ne sont pas accessibles sur une seule machine, mais l'information est au contraire distribuée sur plusieurs machines : les sites, et donc les pages sont hébergées sur différents serveurs, et chaque page contient en elle-même ses liens sortants. La taille du graphe, et son caractère dynamique rendent inenvisageable un algorithme contenant un "Pour chaque arc u du graphe du Web …".

Mentionnons aussi les graphes qui sont mathématiquement bien définis mais dont la taille rend impossible la description exhaustive en machine. C'est le cas par exemple du graphe des échecs : les sommets sont les différents états possibles de la partie (il y en a un nombre fini), ces états sont reliés par des arcs représentant les coups pouvant être joués.

On dit de tels graphes que nous en avons une définition **implicite** : étant donné un sommet il est possible de calculer la liste de ses successeurs, mais il est impossible d'obtenir une liste explicite des sommets ou des arcs du graphe. De tels graphes sont donc représentés en machine par une fonction qui calcule les successeurs d'un sommet. En OCaml et pour le graphe des échecs par exemple, on aurait des définitions de types de la forme suivante.

```
type etat = ... (* Une définition de type pour un plateau d'échec *)

let init : etat =
  ...
  (* L'état initial de la partie d'échec *)

let successeurs (e: etat): etat list =
  ...
  (* La liste des états pouvant être obtenus après un coup depuis e *)
```

♣. on rédige cette partie pour le cas des graphes orientés, mais cela est vrai aussi pour des graphes non orientés

Dans le cas du graphe du Web on pourrait avoir les définitions de types suivantes.

```
type page = string (* Une page Web est identifiée par son URL *)

let successeurs (p: page): page list =
  ...
  (* La liste des pages accessibles en suivant un lien depuis p *)
```

Remarquons qu'avec de telles définitions il est possible de faire des parcours partiels (en profondeur, en largeur, ou autres) des graphes implicites ainsi représentés. De plus, afin de réaliser de tels parcours, on se dotera d'une structure de données pour représenter l'ensemble des sommets visités (un ensemble d'`etat` ou de `page` ci-dessus), ainsi que des structures de mise en attente (pile, file, ou autres, d'`etat` ou de `page`). Dans le cas du graphe du Web par exemple, on pourrait opter en OCaml pour un ensemble des visités représenté au moyen d'une `(string, bool) Hashtbl.t` et une structure de mise en attente de type `String Queue.t`.

4 Rangements particuliers des sommets

Dans cette section on étudie d'autres permutations remarquables ♣ pour les graphes orientés. Ainsi dans toute cette section on considère $G = (S, A)$ un graphe orienté. On notera $n = \text{card}(S)$ et $m = \text{card}(A)$.

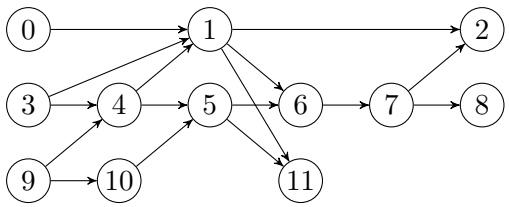
4.1 Tri topologique

Définition 4.1

Soit $T = (T_i)_{i \in [\![1,n]\!]}$ une permutation des sommets de G.
*On dit que T est un **tri topologique** de G dès lors que $\forall (i,j) \in [\![1,n]\!]^2$, $(T_i, T_j) \in A \Rightarrow i \leqslant j$, autrement dit tous les arcs de G ont leur origine placée avant leur destination dans T.*

Exercice de cours 4.2

Donner un tri topologique du graphe ci-dessous.



Exercice de cours 4.3

Que peut-on dire du premier sommet d'un tri topologique ?

Exercice de cours 4.4

Y a-t-il unicité du tri topologique ?

♣. autres que les parcours

Proposition 4.5

Un graphe admet un tri topologique si et seulement s'il est sans circuit.

Démonstration : Il s'agit de démontrer les deux implications.

- Si G admet un tri topologique des sommets $(T_1, T_2, \ldots, T_n)$ et un circuit $(T_{i_1}, T_{i_2}, \ldots T_{i_p} = T_{i_1})$, on a $i_1 \leqslant i_2 \ldots \leqslant i_p = i_1$ donc $T_{i_1} = T_{i_2} = \ldots = T_{i_p}$ donc G est sans circuit.
- On montre la réciproque par induction sur n. La propriété est bien sûr vraie si $n = 1$. Soit $n \geqslant 2$. Supposons qu'elle le soit pour tout graphe à $n-1$ sommets. Soit G un graphe à n sommets sans circuit. Il existe au moins un sommet s_0 sans successeurs car dans le cas contraire, on pourrait construire un chemin infini. Or comme il y a un nombre fini de sommets, l'un d'eux serait nécessairement répété dans ce chemin infini qui présenterait donc un circuit, ce qui est absurde.
  Le sous-graphe G_0 de G induit par $S \setminus \{s_0\}$ a $n-1$ sommets et ne possède pas de circuits. Par hypothèse d'induction il existe donc un tri topologique L_0 des sommets de G_0. En ajoutant le sommet s_0 à la fin du tri L_0 on obtient un tri topologique des sommets de G, puisque les seuls arcs de G qui ne sont pas dans G_0 ont s_0 comme extrémité, et même comme destination puisque s_0 est sans successeur dans G. Autrement dit ce sont bien des arcs qui vont d'un sommet de L_0, le début de L, vers s_0, la fin de L.

□

4.2 Tri préfixe

La notion de tri topologique introduite dans la section précédente n'a de sens que dans les graphes sans circuit. On "relâche" la définition de tri topologique pour obtenir celle de tri préfixe : c'est une permutation des sommets du graphe qui donne un tri topologique quand on passe au graphe réduit.

Définition 4.6

Soit $T = (T_i)_{i \in [\![1,n]\!]}$ une permutation des sommets de G.
*On définit le **rang** d'un sommet u dans la permutation T, noté $\mathrm{rg}_T(u)$, comme étant le plus petit indice d'un élément dans la même CFC que u.*

$$\mathrm{rg}_T(u) \overset{\text{déf}}{=} \min\{i \in [\![1,n]\!] \mid T_i \sim_G u\}$$

*On dit que T est un **tri préfixe** de G dès lors que $\forall (u,v) \in S^2, (u,v) \in A \Rightarrow \mathrm{rg}_T(u) \leqslant \mathrm{rg}_T(v)$.*

Remarque 4.7

Dans un tri topologique, on demande que si (u,v) est une arête du graphe alors u est placé avant v dans le tri topologique ; dans le cas du tri préfixe on demande seulement que la composante connexe de u apparaisse avant la composante connexe de v au sens où un des éléments de cette composante apparaît avant tous les éléments de l'autre composante.

Lemme 4.8

Pour tout $(u,v) \in S^2$ et toute permutation des sommets T, $u \sim v \Leftrightarrow \mathrm{rg}_T(u) = \mathrm{rg}_T(v)$.

Démonstration :

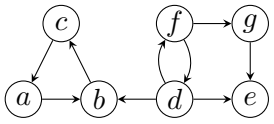
- Si $u \sim_G v$ alors $\{i \in [\![1,n]\!] \mid T_i \sim_G u\} = \{i \in [\![1,n]\!] \mid T_i \sim_G v\}$ et donc $\mathrm{rg}_T(u) = \mathrm{rg}_T(v)$.
- Si $\mathrm{rg}_T(u) = \mathrm{rg}_T(v)$ alors soit $i = \mathrm{rg}_T(v)$, on a donc $u \sim T_i \sim v$.

□

Exercice de cours 4.9

Dans les permutations suivantes des sommets du graphe ci-contre, donner le rang de chaque sommet et dire si cette permutation est un tri préfixe.

- $[g, d, e, f, a, c, b]$
- $[d, f, a, e, g, c, b]$
- $[d, a, g, e, b, c, f]$
- $[f, d, b, g, a, c, e]$



Proposition 4.10

Un tri topologique est un tri préfixe.
Un tri préfixe d'un graphe sans circuit est un tri topologique.

Démonstration : S'il existe un tri topologique, le graphe est nécessairement sans circuit, et donc sa décomposition en CFC est la partition en singletons. Ainsi le rang de n'importe quel sommet T_i est simplement i, son propre indice puisqu'il est le seul, et donc le premier, représentant de sa classe. La condition sur T pour qu'il soit tri préfixe se réécrit alors exactement comme la condition pour qu'il soit un tri topologique :

$$\forall (T_i, T_j) \in S^2,\ (T_i, T_j) \in A \Rightarrow i = \mathsf{rg}_T(T_i) \leqslant \mathsf{rg}_T(T_j) = j$$

□

Exercice de cours 4.11

Supposons connu la décomposition en CFC de G.
1. Connaissant un tri préfixe de G, comment obtenir un tri topologique de son graphe réduit $\widehat{G}$?
2. Connaissant un tri topologique du graphe réduit $\widehat{G}$, comment obtenir un tri préfixe de G ?

Calcul d'un tri préfixe. On s'intéresse à la mise en place d'un algorithme permettant le calcul d'un tri préfixe. On remarque le fait suivant : si tous les descendants stricts d'un sommet u (les descendants de u qui ne sont pas dans la même composante fortement connexe que u) sont déjà placés dans une liste, alors on peut placer u en tête de cette liste.

Exercice de cours 4.12

Justifier cette affirmation.

Ainsi on construit le tri préfixe de la fin vers le début en adoptant la politique suivante : on ajoute x en tête après que tous ses descendants stricts ont été ajoutés. On remarque que cet ajout correspond alors au moment où on ferme l'appel récursif à `explore_descendants` lors d'un parcours en profondeur. On implémente donc cette politique avec un algorithme récursif. La fonction `Insérer` prend en argument un sommet x et insére le sommet x et ses descendants non encore ajoutés au tri préfixe en cours de calcul. Si x n'est pas encore visité, on appelle récursivement `Insérer` sur les descendants de x afin de s'assurer que tous les descendants de x sont déjà rangés dans le tri préfixe,

puis on ajoute x en tête du tri préfixe.

Algorithme 3 : Calcul d'un tri préfixe

Entrée : Un graphe orienté $G = (S, A)$ représenté par une table de listes de successeurs
Sortie : Un tri préfixe des sommets de G

```
1  Res ← liste vide ;
2  Ouverts ← ∅ ;
3  Fermés ← ∅ ;
4  Procedure Insérer(s) :
       Entrée : s ∈ S
       Effet : Modifie Fermés et Res de sorte que Fermés contienne s et tous ses descendants
               (sauf ceux qui sont dans Ouverts) et que Res reste un tri préfixe de Fermés
5      si s ∉ Ouverts et s ∉ Fermés alors
6          Ajouter s à Ouverts ;
7          pour tout x ∈ Succ(s) faire
8              Insérer(x) ;
9          Supprimer s de Ouverts ;
10         Ajouter s à Fermés ;
11         Res ← s · Res ;
12 tant que S\Fermés ≠ ∅ faire
13     s ← un sommet de S\Fermés ;
14     Insérer(s) ;
15 retourner Res ;
```

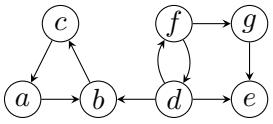
Exercice de cours 4.13

Donner un invariant liant les ensembles Ouverts et Fermés.
Que peut-on dire de l'ensemble Ouverts à la ligne 13 ?

Exercice de cours 4.14

Exécuter l'algorithme précédent sur le graphe ci-contre, en choisissant les points de régénération selon les ordres suivants.

- $[a, b, c, d, e, f, g]$
- $[g, f, e, d, c, b, a]$



5 Calcul de plus courts chemins

L'algorithme A*, de la section suivante, sera présenté de manière incrémentale par rapport à l'algorithme de Dijkstra. Pour cette raison on rappelle ici l'algorithme de Dijkstra, on ne redonne pas les résultats de correction et de complexité de l'algorithme, on renvoit pour cela aux cours de première année.
Ainsi dans toute cette section on considère $G = (S, A, c)$ un graphe orienté pondéré où $c : A \to \mathbb{R}^+$. On notera $n = \text{card}(S)$ et $m = \text{card}(A)$. Étant donnés deux sommets u et v de S, on définit la distance de u à v dans G, ce que l'on note $d(u, v)$, comme étant la plus petite longueur d'un chemin de u à v dans G. Dans le cas où un tel chemin n'existe pas, on convient que $d(u, v) = +\infty$.

Exercice de cours 5.1

Donner la définition formelle de d, comme un $\min$ sur un ensemble qui admet un minimum ou qui est vide.

Exercice de cours 5.2

Supposons que $\gamma = \gamma_0\gamma_1\dots\gamma_q$ est un chemin de longueur minimale, *i.e.* de longueur $d(\gamma_0, \gamma_q)$. Soit $(i, j) \in [\![0]\!], q]\!]^2$ tel que $i \leqslant j$. Montrer que le sous-chemin $\gamma_i\gamma_{i+1}\dots\gamma_j$ est aussi un chemin minimal (de γ_i à γ_j).

5.1 L'algorithme de Dijkstra

Algorithme 4 : Dijkstra

Entrée : Un graphe pondéré $G = (S, A, c)$ où $c : A \to \mathbb{R}^+$, une source $s \in S$
Sortie : Une table indexée par S donnant la distance depuis s de chaque sommet de G

```
1  Initialiser μ[u] à +∞ pour chaque u ∈ S ;
2  Initialiser π[u] à ?? pour chaque u ∈ S ;
3  μ[s] ← 0 ;                          (* La source est à distance 0 d'elle-même *)
4  π[s] ← s ;                          (* La source est la racine de l'arborescence *)
5  todo ← {s} ;                        (* La source est le premier sommet à traiter*)
6  Procedure Relâcher((u, v)) :
       Entrée : u et v deux sommets voisins dans G ;
7      si μ[u] + c((u, v)) < μ[v] alors
8          μ[v] ← μ[u] + c((u, v)) ;
9          π[v] ← u ;
10         Ajouter v à todo ;
11 tant que todo ≠ ∅ faire
12     Extraire de todo un sommet u minimisant μ[u] ;
13     pour tout v ∈ succ(u) faire
14         Relâcher(u, v)
15 retourner μ
```

Exercice de cours 5.3

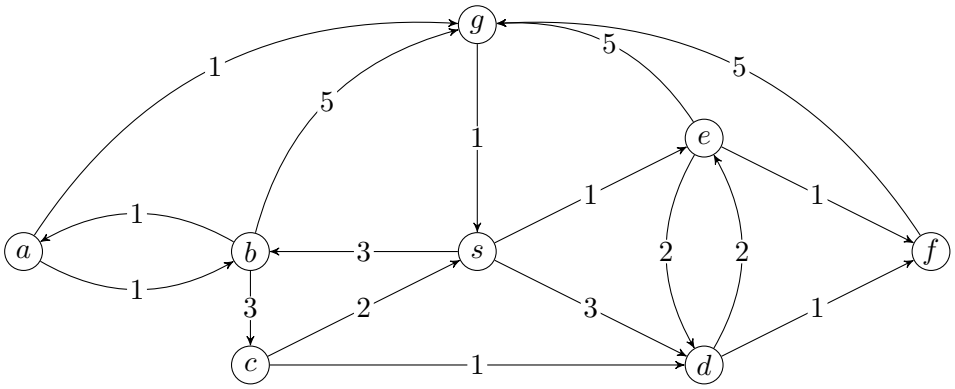
Quelle structure de donnée concrète est adaptée pour implémenter l'ensemble todo ?
Redonner alors la complexité de l'algorithme de Dijkstra, et justifier.

Exercice de cours 5.4

En quoi l'algorithme de Dijkstra est-il un algorithme de parcours ?
Donner un cas où l'algorithme de Dijkstra coïncide avec un algorithme de parcours vu précédemment.

Exercice de cours 5.5

Exécuter l'algorithme de Dijkstra sur le graphe ci-dessous, depuis la source s.



5.2 L'algorithme A*

L'algorithme de Dijkstra est un algorithme de calcul des plus courts chemins depuis une source du graphe, vers chacun des sommets du graphe. Il n'est pas rare que l'on ait seulement besoin d'un plus court chemin entre deux sommets bien identifiés d'un graphe : non seulement nous disposons d'un sommet **source** mais aussi d'un sommet **objectif**. On présente donc dans cette section une variante de l'algorithme de Dijkstra : l'algorithme A*, qui met à profit cette information supplémentaire d'objectif pour essayer d'améliorer le temps de calcul de l'algorithme de Dijkstra.

Exercice de cours 5.6

Si l'on connaît le sommet objectif o, et que l'on cherche à calculer seulement la distance $d(s, o)$, peut-on arrêter l'algorithme de Dijkstra dès que $\mu[o]$ prend une valeur finie ? Dès que o est sorti de todo ? Dans chaque cas, si la réponse est oui justifier, et sinon donner un contre exemple.

L'algorithme de Dijkstra visite les sommets du graphe à la manière d'un parcours en largeur depuis la source : les sommets sont parcourus par distances croissantes depuis la source. Toutefois dans le cas où l'on connaît l'objectif, on aimerait aussi que l'algorithme cherche "dans la direction" de l'objectif.

On fournit ci-dessous des exemples sur un graphe non orienté représentant les déplacements sur les cases d'une grille : d'une case (un sommet du graphe) on peut se déplacer (les arcs du graphe) dans une des 8 cases adjacentes ♣, à moins qu'elles ne soient des murs. Les arcs sont pondérés de la manière suivante : on se déplace horizontalement et verticalement pour un coût de 1, on se déplace en diagonale pour un coût de $\sqrt{2}$.

On présente ci-dessous (Figure 1) les sommets parcourus par l'algorithme de Dijkstra lors de la recherche d'un chemin depuis la source (en bas à gauche) ● et vers l'objectif ●. Les cases ■ représentent des murs et ne sont donc pas accessibles. Les cases ■ sont les sommets qui ont été visités tandis que les cases ■ sont les sommets se trouvant dans la file de priorité. Finalement les traits représentent les plus courts chemins, soit l'arborescence encodée par le tableau de prédécesseurs π. Cet exemple met en avant le fait que l'algorithme de Dijkstra parcourt les sommets "en largeur" depuis la source.

♣. Les huit cases sont : haut, bas, gauche, droite, coin haut gauche, coin haut droite, coin bas gauche, coin bas droite.

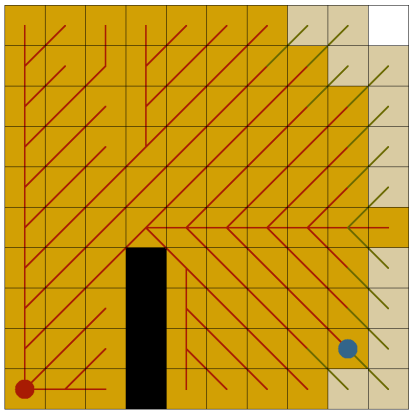


FIGURE 1 – Exécution de l'algorithme de Dijkstra

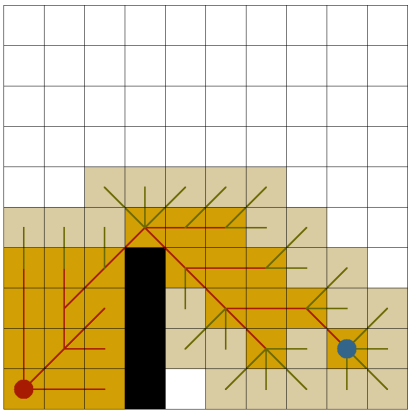


FIGURE 2 – Exécution de l'algorithme A*

L'algorithme A* évite l'écueil de l'algorithme de Dijkstra mis en avant dans l'exemple précédent grâce à une fonction heuristique qui donne, pour chaque sommet du graphe, une estimation de la distance restant à parcourir pour atteindre l'objectif. Dans ce cadre, on appelle **heuristique** une fonction associant un score positif à chaque sommet du graphe, $h : S \to \mathbb{R}^+$.

Définition 5.7

*On dit d'une heuristique qu'elle est **admissible** dès lors qu'elle donne une minoration de la distance à l'objectif dans le graphe. Autrement dit, h est une heuristique admissible dès lors que pour tout sommet $u \in S$ tel qu'il existe un chemin de u au sommet objectif o de longueur l, on a $h(u) \leqslant l$.*

Une telle heuristique est alors utilisée pour choisir le sommet à traiter dans l'ensemble todo comme étant, non pas un de ceux qui minimisent $y \mapsto \mu[y]$, mais un de ceux qui minimisent $y \mapsto \mu[y] + h(y)$. Autrement dit, au lieu de choisir d'explorer un sommet dont la distance connue à la source est minimale♣, on choisit d'explorer un sommet par lequel pourrait passer un chemin de longueur minimale reliant s et o si on en croit l'heuristique. On présente ci-dessus (Figure 2) les sommets parcourus par l'algorithme de A* lors de la recherche d'un chemin dans les mêmes conditions que précédemment, en utilisant la distance à vol d'oiseau comme heuristique.

Exercice de cours 5.8

Expliquer pourquoi la distance à vol d'oiseau donne une heuristique admissible.

On présente ci-dessous l'algorithme A* en mettant en jaune les lignes qui changent par rapport à

♣. ce qui assure en fait, on le rappelle, que la distance exacte à la source est minimale,

l'algorithme de Dijkstra.

Algorithme 5 : A*

Entrée : Un graphe pondéré $G = (S, A, c)$ où $c : A \to \mathbb{R}^+$, une source $s \in S$, un objectif $o \in S$, une heuristique $h : S \to \mathbb{R}^+$

Sortie : La distance de s à o

1 Initialiser $\mu[u]$ à $+\infty$ pour $u \in S$;
2 Initialiser $\eta[u]$ à $+\infty$ pour $u \in S$;
3 Initialiser $\pi[u]$ à ?? pour $u \in S$;
4 $\mu[s] \leftarrow 0$; (* La source est à distance 0 d'elle-même *)
5 $\eta[s] \leftarrow \mu[s] + h(s)$; (* Estimation de la distance source objectif *)
6 $\pi[s] \leftarrow s$; (* La source est la racine de l'arborescence *)
7 todo $\leftarrow \{s\}$; (* La racine est le premier sommet à traiter *)
8 **Procedure** Relâcher$((u, v))$:
    Entrée : u et v deux sommets voisins dans G ;
9 **si** $\mu[u] + c((u, v)) < \mu[v]$ **alors**
10 $\mu[v] \leftarrow \mu[u] + c((u, v))$;
11 $\eta[v] \leftarrow \mu[v] + h(v)$;
12 $\pi[v] \leftarrow u$;
13 Ajouter v à todo ;
14 **tant que** todo $\neq \emptyset$ **faire**
15 Extraire de todo un sommet u minimisant $\eta[u]$;
16 **si** $u = o$ **alors**
17 **retourner** $\mu[o]$; (* Objectif atteint*)
18 **pour tout** $v \in$ voisins(u) **faire**
19 Relâcher(u, v)
20 **retourner** ∞ ; (* Objectif inaccessible*)

Exercice de cours 5.9

Que dire de l'exécution de l'algorithme $A^\star$ avec l'heuristique nulle (*i.e.* $h : s \in S \mapsto 0$) ?

Théorème 5.10

Si l'heuristique utilisée est admissible, l'algorithme A* est correct.

Démonstration :

1) Remarquons tout d'abord que pour tout sommet $x \in S$, la valeur de $\mu[x]$ ne fait que décroître à mesure que l'algorithme A* s'exécute. En effet les valeurs de μ ne sont modifiées qu'à la ligne 10 de la fonction Relâcher, et d'après le test qui précède cette instruction, la valeur est modifiée pour une valeur strictement plus petite.

2) La boucle **tant que** de l'algorithme A* vérifie les invariants suivants[♣].

$$\mathcal{I}_1 : \forall x \in S,\ \mu[x] < +\infty \Rightarrow \exists \gamma \text{ chemin de } G \text{ de longueur } \mu[x] \text{ reliant } s \text{ et } x$$
$$\mathcal{I}_2 : \forall x \in \text{todo},\ \mu[x] < +\infty$$

- Initialement (*i.e.* l.6), le seul sommet dans todo est s, et on a alors $\mu[s] = 0 < +\infty$ d'où l'invariant $\mathcal{I}_2$. De plus le chemin $(\gamma_0 = s)$ étant un chemin de longueur nulle reliant s et s, l'invariant $\mathcal{I}_1$ est bien vérifié.

♣. les mêmes que l'algorithme de Dijkstra

- Supposons que les deux invariants sont vérifiés au début d'un tour de la boucle **tant que**. Soit $u \in S$ le sommet traité lors de ce tour de boucle. D'après l'invariant $\mathcal{I}_2$, on a donc $\mu[u] < +\infty$. Les seuls sommets ajoutés à todo lors de ce tour de boucle sont les voisins v de u, dont la valeur de μ est d'abord ajustée à $\mu[u] + c((u,v))$, et donc en particulier finie. Ainsi l'invariant $\mathcal{I}_2$ reste vrai.

  De plus, si l'on considère $x \in S$ un sommet dont la valeur de μ passe de $+\infty$ à une valeur finie au cours de ce tour de boucle, on sait que x est nécessairement voisin de u. Or l'invariant $\mathcal{I}_1$ assure (puisque $\mu[u] < +\infty$) qu'il existe un chemin de s à u. On en déduit l'existence d'un chemin, de longueur convenable, de s à x. Ainsi l'invariant $\mathcal{I}_1$ se propage.

À ce point de la preuve, on peut déjà conclure quant à la correction de l'algorithme dans le cas où $d(s,o) = \infty$. Dans ce cas, il n'existe aucun chemin reliant s et o donc $\mu[o]$ ne peut jamais prendre une valeur finie, en particulier o ne sera jamais ajouté à todo et on ne sortira pas de la fonction à la ligne 17. Ainsi, pourvu que l'algorithme termine, la valeur renvoyée dans ce cas là est $+\infty$ (*Cf.* l. 20).
Dans les autres cas, ces invariants assurent que la valeur $\mu[o]$ renvoyée par la ligne 17 est un majorant de la distance $d(s,o)$. On s'attache dans la suite à montrer qu'au moment où o est traité cette valeur est exacte.
Dans la suite de la preuve, on suppose donc que $d(s,o) < +\infty$, ainsi il existe un chemin de G reliant s et o de longueur $d(s,o)$.

3) Soit γ un chemin reliant $s = \gamma_0$ à $o = \gamma_q$ de longueur minimale. On montre ici qu'à n'importe quelle étape de l'algorithme après la ligne 6, et pour tout $k \in [\![0, q[\![$, si $\mu[\gamma_k] = d(s,\gamma_k)$ et $\gamma_k \notin$ todo, alors $\mu[\gamma_{k+1}] = d(s,\gamma_{k+1})$.

Considérons un instant t de l'exécution de l'algorithme et $k \in [\![0,q[\![$ tel que $\mu[\gamma_k] = d(s,\gamma_k)$ et $\gamma_k \notin$ todo . Puisque $\mu[\gamma_k] < +\infty$, le sommet a été ajouté à todo au moins au moment où il a reçu pour la première fois une valeur de μ finie. Du fait que $\gamma_k \notin$ todo à l'instant t, on sait que γ_k a été extrait de todo.

Considérons l'instant $t' < t$ où γ_k a été extrait de todo pour la dernière fois. La valeur de $\mu[\gamma_k]$ n'a pas changé depuis (sans quoi γ_k aurait été remis dans todo). Lors du traitement de γ_k qui a précédé son extraction à l'instant t', on a mis à jour la valeur de μ pour γ_{k+1} voisin de γ_k. Ainsi à l'instant t' :

$$\mu[\gamma_{k+1}] \leqslant \mu[\gamma_k] + c((\gamma_k,\gamma_{k+1})).$$

Entre t' et t la valeur de $\mu[\gamma_{k+1}]$ n'a pu que décroître d'après le point **1)** et celle de $\mu[\gamma_k]$ n'a pas changé, ainsi à l'instant t on a :

$$\mu[\gamma_{k+1}] \leqslant \mu[\gamma_k] + c((\gamma_k,\gamma_{k+1})).$$

Or par hypothèse on a aussi $\mu[\gamma_k] = d(s,\gamma_k)$, donc :

$$\mu[\gamma_{k+1}] \leqslant d(s,\gamma_k) + c((\gamma_k,\gamma_{k+1})).$$

De plus par minimalité de γ on a $d(s,\gamma_k) + c((\gamma_k,\gamma_{k+1})) = d(s,\gamma_{k+1})$, d'où :

$$\mu[\gamma_{k+1}] \leqslant d(s,\gamma_{k+1}).$$

Enfin par l'invariant vu au point **2)** on a $\mu[\gamma_{k+1}] \geqslant d(s,\gamma_{k+1})$, d'où l'égalité souhaitée :

$$\mu[\gamma_{k+1}] = d(s,\gamma_{k+1}).$$

4) On montre ici que dès la ligne 6 et tant que o n'est pas extrait de todo, il existe un sommet $x \in S$ vérifiant les trois conditions suivantes :

$a)$ x est dans todo ;
$b)$ $\mu[x] = d(s,x)$;
$c)$ il existe un chemin minimal de s à o passant par x.

- Initialement todo contient s qui est bien sur un chemin minimal reliant s et o (puisqu'un tel chemin existe), et qui vérifie $\mu[s] = 0$ d'après la ligne 4, soit $\mu[s] = d(s, s)$.
- Supposons qu'au début d'un tour de boucle il existe un tel sommet x dans todo et que l'on traite un sommet $u \neq o$.
  - Si $u \neq x$, alors x est encore dans todo à la fin du tour. L'existence d'un chemin minimal passant par x n'est pas remise en cause. Enfin la valeur de $\mu[x]$ a pu être modifiée, mais seulement à la baisse d'après le point **1)** or on a aussi $\mu[x] \geqslant d(s, x)$ d'après le point **2)**. Ainsi x justifie encore que la propriété est vérifiée à la fin du tour de boucle.
  - Si $u = x$, alors $x \neq o$. De plus x est par hypothèse sur un chemin élémentaire γ reliant $s = \gamma_0$ à $o = \gamma_q$ de longueur minimale. Notons $i \in [\![0, q[\![$ l'indice tel que $\gamma_i = x$. À la fin du tour de boucle on a $\gamma_i \notin$ todo et $\mu[\gamma_i] = d(s, \gamma_i)$, donc le point **3)** assure que $\mu[\gamma_{i+1}] = d(s, \gamma_{i+1})$. Ou bien $\gamma_{i+1} \in$ todo, et alors ce sommet vérifie $a)$ et $b)$ et $c)$, ou bien $\gamma_{i+1} \notin$ todo et on peut à nouveau utiliser le point **3)**. En itérant ainsi, on aboutit nécessairement sur un indice $k \in [\![i+1, q]\!]$ tel que $\gamma_k \in$ todo et $\mu[\gamma_k] = d(s, \gamma_k)$, sans quoi on aurait $o = \gamma_q \notin$ todo et aussi $\mu[\gamma_q] = d(s, \gamma_q)$, donc $\mu[\gamma_q] < \infty$ ce qui contredit le fait qu'on n'ait pas encore traité o. Ce sommet γ_k vérifie donc $a)$ et $b)$ et $c)$, ainsi la propriété est vérifiée en fin de tour de boucle.

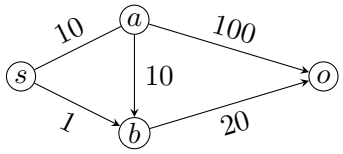
On a traité plus haut le cas où $d(s, o) = \infty$, on traite donc ici le cas où o est accessible depuis s. Ainsi l'invariant du point **4)** assure que la condition de sortie de la boucle while ne peut être satisfaite tant que o n'a pas été traité, autrement dit on sort nécessairement de l'algorithme par la ligne 17.
Notons x un sommet vérifiant $a)$, $b)$, $c)$ au moment où o est sélectionné à la ligne 15 précédant cette sortie. Puisque $x \in$ todo (propriété a), le choix de o comme sommet à traiter assure $\eta[o] \leqslant \eta[x]$. Or :

$$\begin{aligned}
\eta[x] &= \mu[x] + h(x) && \textit{par invariant immédiat sur } \eta \\
&= d(s, x) + h(x) && \textit{par } b \\
&\leqslant d(s, x) + d(x, o) && \textit{car } h \textit{ est minorante} \\
&= d(s, o) && \textit{par } c
\end{aligned}$$

et $h(o) = 0$ puisque h est minorante. On en déduit que $\mu[o] + 0 \leqslant d(s, o)$, or $\mu[o] \geqslant d(s, o)$ par invariant du point **2)** donc $\mu[o] = d(s, o)$ et la valeur renvoyée est bien celle attendue. □

Exercice de cours 5.11

Exécuter l'algorithme A⋆ sur le graphe ci-contre avec l'heuristique h définie comme suit.
- $h = s \mapsto 0, a \mapsto 0, b \mapsto 0, o \mapsto 0$ (heuristique nulle)
- $h = s \mapsto 10, a \mapsto 1, b \mapsto 20, o \mapsto 0$ (heuristique admissible non monotone)
- $h = s \mapsto 10, a \mapsto 10, b \mapsto 9, o \mapsto 0$ (heuristique admissible et monotone)



Remarque 5.12

On remarque que dans l'algorithme A⋆, contrairement à ce qui se passe dans l'algorithme de Dijkstra, un sommet x peut être extrait de todo avec une valeur $\mu[x] > d(s, x)$ (et risque dans ce cas d'y être à nouveau ajouté avec une valeur de $\mu[x]$ réduite), et qu'un arc peut être relâché plusieurs fois ♣.

Remarque 5.13

Pour démontrer la correction de l'algorithme de Dijkstra, on peut utiliser de même la remarque du point **1)** et les invariants $\mathcal{I}_1$ et $\mathcal{I}_2$, mais les points suivants se simplifient, car on peut directement montrer que lorsqu'un sommet x est extrait de todo la valeurs $\mu[x]$ vaut $d(s, x)$.

♣. *Cf.* le sommet a et l'arc (a, o) dans l'exercice de cours 5.11 pour l'heuristique non monotone.

Remarque 5.14

La terminaison de l'algorithme A* découle du fait qu'un sommet u ne peut être extrait de todo qu'un nombre fini de fois, assurant ainsi un nombre de tours de boucle ligne 14 fini. En effet l'invariant $\mathcal{I}_1$ démontré ci-avant peut être renforcé comme suit.

$$\mathcal{I}_1' : \forall x \in S,\ \mu[x] < +\infty \Rightarrow \exists \gamma \text{ chemin } \textbf{élémentaire} \text{ de } G \text{ de longueur } \mu[x] \text{ reliant } s \text{ et } x$$

Ainsi, pour un sommet x fixé, la suite des valeurs prises par $\mu[x]$ à chaque fois que x est traité est une suite strictement décroissante à valeurs dans un ensemble fini — à savoir l'ensemble des longueurs des chemins élémentaires dans G — et donc finie.

Définition 5.15

*On dit qu'une heuristique h est **monotone** dès lors que* $\forall (u,v) \in A,\ h(u) \leqslant c(u,v) + h(v)$

Exercice de cours 5.16

La distance à vol d'oisau dans le graphe sur une grille donné en exemple plus haut est-elle une heuristique monotone ?

Proposition 5.17

Si l'heuristique utilisée est admissible et monotone, chaque sommet u n'est traité qu'une seule fois par l'algorithme A*, et dès qu'il est traité, $\mu[u]$ vaut $d(s,u)$.

Démonstration : On reprend les invariants démontrés dans la preuve de la proposition 5.10. Ils assurent en particulier qu'un sommet x hors de todo tel que $\mu[x] = d(s,x)$ ne peut pas être y ajouté à nouveau, car on ajoute un sommet x dans todo que lorsqu'on donne à $\mu[x]$ une valeur strictement plus petite, or $\mathcal{I}_1$ assure que $\mu[x]$ est déjà au minimum. Ainsi il suffit de montrer que lorsqu'un sommet u est traité la valeur $\mu[u]$ donne $d(s,u)$. On montre ce résultat par récurrence finie sur les étapes de traitement de sommet.

Lors de la première étape c'est le sommet s qui est traité, il a bien la valeur $\mu[s] = 0 = d(s,s)$.

Plaçons nous à une étape suivante de l'algorithme, et supposons que tous les sommets traités auparavant vérifient la propriété. Notons v ($v \neq s$) le sommet traité à l'étape courante, et supposons par l'absurde que $\mu[v] \neq d(s,v)$. D'après l'invariant $\mathcal{I}_2$, $\mu[v] < \infty$ puisque $v \in$ todo, et donc $\mu[v] \geqslant d(s,v)$ par l'invariant $\mathcal{I}_1$. Ainsi on suppose en fait que $\mu[v] > d(s,v)$. Il existe donc un chemin γ de $\gamma_0 = s$ à $\gamma_q = v$ de longueur $d(s,v) < \mu[v]$. On sait que $\gamma_0 = s$ a déjà été traité, et $\gamma_q = v$ pas encore, on peut donc considérer $k \in [\![0, q[\![$ l'indice du dernier sommet traité le long de γ. Ainsi par hypothèse de récurrence, $\mu[\gamma_k] = d(s,k)$. Lors du traitement de γ_k, son successeur γ_{k+1} a été ajouté à todo et a reçu la valeur $\mu[\gamma_{k+1}] = \mu[\gamma_k] + c(\gamma_k, \gamma_{k+1}) = d(s,k) + c(\gamma_k, \gamma_{k+1}) =$ ♣$d(s,\gamma_{k+1})$. Ainsi $d(s,v)$ la longueur du chemin γ se réécrit comme $\mu[\gamma_{k+1}] + l$ où $l = c(\gamma_{k+1}, \gamma_{k+2}) + c(\gamma_k, \gamma_{k+1}) \cdots + c(\gamma_{q-1}, \gamma_q)$ désigne la longueur du chemin $\gamma_{k+1} \dots \gamma_q$.

- Si $k = q-1$, lors du traitement du sommet γ_k, le sommet v a reçu la valeur $\mu[v] = \mu[\gamma_k] + c(\gamma_k, \gamma_{k+1})$ soit $d(s,v)$. Absurde.
- Si $k < q-1$, on utilise la monotonie de h le long du chemin $\gamma_{k+1} \dots \gamma_q$.

$$\begin{aligned} h(v) = h(\gamma_q) &\geqslant h(\gamma_{q-1}) - c(\gamma_{q-1}, \gamma_q) \\ &\geqslant h(\gamma_{q-2}) - \big(c(\gamma_{q-2}, \gamma_{q-1}) + c(\gamma_{q-1}, \gamma_q)\big) \\ &\dots \\ &\geqslant h(\gamma_{k+1}) - \big(\underbrace{c(\gamma_{k+1}, \gamma_{k+2}) + \cdots + c(\gamma_{q-1}, \gamma_q)}_{=l}\big) \end{aligned}$$

♣. On utilise implicitement le fait que le chemin $\gamma_0 \dots \gamma_{k+1}$ est minimal, en tant que sous-chemin d'un chemin minimal, et donc de longueur est $d(s, \gamma_{k+1})$.

Ainsi :

$$\begin{aligned}
\mu[v] + h(v) &\geqslant \mu[v] + h(\gamma_{k+1}) - l && \textit{résultat de la monotonie ci-dessus} \\
&> d(s,v) - l + h(\gamma_{k+1}) && \textit{par notre hypothèse absurde} \\
&> (\mu[\gamma_{k+1}] + l) - l + h(\gamma_{k+1}) && \textit{en décomposant la longueur de } \gamma \\
&> \mu[\gamma_{k+1}] + h(\gamma_{k+1})
\end{aligned}$$

or il est absurde que $\mu[v] + h(v) > \mu[\gamma_{k+1}] + h(\gamma_{k+1})$ sans quoi on aurait traité γ_{k+1} (présent dans todo et de meilleur score) et pas v à cette étape.

Dans les deux cas on arrive à une contradiction, ainsi on a bien $\mu[v] = d(s,v)$. □