Chapitre deux

Explorations de graphes

MPI/MPI*, lycée Faidherbe

Résumé

Dans ce chapitre on revient sur le parcours en profondeur ; on introduit les ordre préfixe et postfixe associés et on étudie l'algorithme de Kosaraju-Sharir de calcul des composantes fortement connexes d'un graphe orienté.

I Rappels

I.1 Exploration en profondeur récursive (dfs)

Le code du parcours en profondeur peut s'écrire

Algorithme 1 : Exploration depuis un sommet

```
fonction explorer(s : sommet) =
    if s n'est pas marqué vu then
        marquer s comme vu
        /* pré-traitement */
        pour tous les t voisins de s dans G faire explorer(t)
        /* post-traitement */
```

- La fonction est récursive, on peut en donner une forme itérative en employant une pile.
- On a besoin d'une fonction qui renvoie les voisins d'un sommet dans un graphe. La complexité est meilleure lorsqu'on emploie un tableau (ou un dictionnaire) de listes d'adjacence pour représenter le graphe.
- Si les sommets sont des entiers de 0 à $n-1$, le marquage des éléments peut être fait par un tableau de booléens. Dans le cas général on peut utiliser une structure d'ensemble ou un dictionnaire liant sommets et booléens (table de hachage, arbre binaire de recherche, ...)
- Cet algorithme est un squelette ; dans la pratique ce sont les pré-traitement et post-traitement qui donneront les résultats voulus. Cela nécessitera peut-être d'ajouter des variables à la fonction.
- Si les fonctions de marquage et le calcul de l'ensemble des voisins sont en temps constant, la complexité est en $\mathcal{O}(|V|)$ où V est l'ensemble des arêtes.

Théorème 1 du chemin blanc

Les sommets marqués comme vus lors du traitement de `explorer(`s`)` sont ceux pour lesquels il existe un chemin à partir de s composé de sommets non encore marqués comme vus avant l'appel de `explorer` (s).

Démonstration *Sens direct.* Soit $s = s_0 \to s_1 \to \cdots \to s_{p-1} \to s_p = t$ un chemin de s à t tel que s_i est non vu lors de l'appel de de `explorer` (s).

- $s_0 = s$ est marqué comme vu lors de l'appel.

- Si s_i est marqué comme vu, cela a été fait lors d'un appel récursif de `explorer` (s_i). Le traitement de s_i va appeler `explorer` (s_{i+1}) car s_{i+1} est un voisin de s_i. À ce moment, soit s_{i+1} est déjà marqué comme vu, soit il le sera lors du traitement de s_{i+1}. Dans tous les cas il est marqué comme vu durant le traitement de `explorer(`s`)`.
- Ainsi, par récurrence, tous les s_i, en particulier $t = s_p$, sont marqués comme vu lors du traitement de `explorer` (s).

Sens réciproque. On remarque que si un sommet u différent de s est marqué comme vu lors de l'exploration de s alors `explorer(u)` a été appelé ; cela n'est possible que lors du traitement d'un sommet non vu v dont u est voisin. À ce moment v doit être marqué comme vu.
Ainsi, depuis un sommet t marqué comme vu lors de l'exploration de s, on peut remonter $t_1 \to t$ puis $t_2 \to t_1$ et ainsi de suite tant que t_i est distinct de s.
Comme les sommets ne peuvent marqués qu'une seule fois, les t_i sont distincts. De plus il y a un nombre fini de sommets donc la suite des t_i est finie : on finit par arriver à s. On construit ainsi un chemin de s à t dont les sommets sont non vus initialement. . . . . ■

On utilise le plus souvent l'exploration plus globalement :

Algorithme 2 : Exploration de tous les sommet

```
fonction explorer-tout(G : graphe) =
    pour tous les sommets s de G faire marquer s comme non vu
    pour tous les sommets s de G faire
        si s n'est pas marqué vu alors
            /* pré-traitement */
            explorer(s)
            /* post-traitement */
```

On note que, lors de l'exploration d'un graphe par la fonction `explorer-tout`, tous les sommets sont marqués vus donc la fonction `explorer` est appelée pour chaque sommet.

I.2 Applications

On peut déduire deux résultats de première année.

Exercice 1 **Solution page 9**

Dans la cas d'un graphe non orienté, prouver qu'à chaque appel de `explorer` (s) dans `explorer-tout`, les sommets explorés forment la composante connexe de s.

On peut donc calculer les composantes connexes d'un graphe ; pour cela on peut donner un numéro de composante connexe pour chaque élément ou construire une liste de listes.

Exercice 2 **Solution page 9**

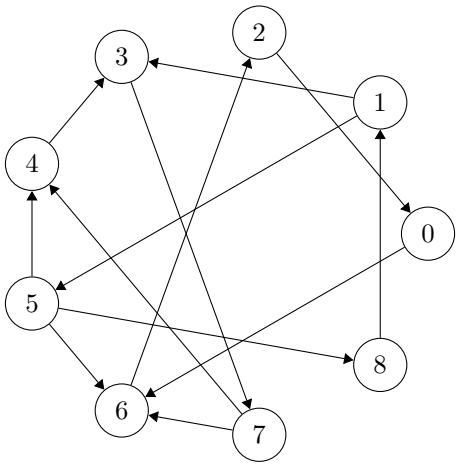
Dans la cas d'un graphe orienté acyclique montrer que, si (s, t) est une arête, alors soit t a été exploré entièrement avant l'appel de `explorer` (s) soit t est exploré pendant l'appel de `explorer` (s).

Ainsi, si on empile les sommets à la fin de leur traitement et si (s, t) est une arête, alors s est situé au-dessus de t dans la pile et l'ordre de de vidage de la pile est un ordre topologique.

II Algorithme de Kosaraju-Sharir

À partir de maintenant, les graphes sont orientés.
Les arêtes sont donc des couples et non des ensembles de 2 sommets.
On utilisera le graphe ci-dessous pour illustrer les notions introduites.

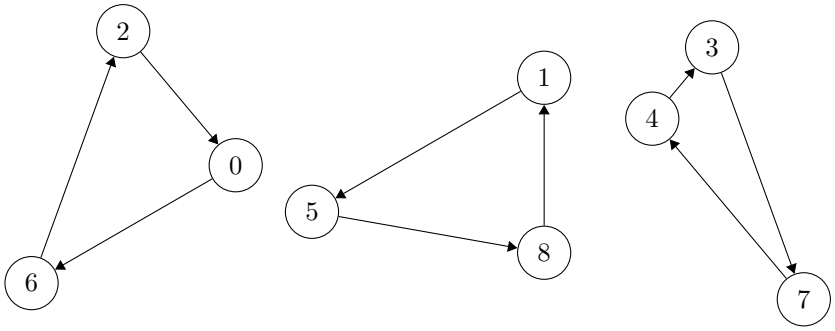
Graphe G_0



II.1 Composantes fortement connexes

On rappelle que deux sommets s et t sont équivalents s'il existe un chemin de s à t et un chemin de t à s. Les composantes fortement connexes (C.F.C.) d'un graphe sont les graphes induits par les classes d'équivalente de cette relation. Par abus de langage on parlera aussi de composantes fortement connexes pour parler des ensembles des sommets d'un graphe induits.

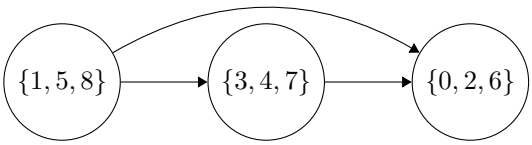
C.F.C. de G_0



Définition 1 : Graphe des composantes

Si G est un graphe orienté, son graphe des composantes est le graphe dont les sommets sont les composantes fortement connexes de G et les arêtes sont les couples de (C, C') tels qu'il existe un sommet s de C et un sommet s' de C' formant une arête (s, s') de G.

Graphe des composantes de G_0



Exercice 3 **Solution page 9**

Montrer que s'il existe un chemin entre deux C.F.C. dans le graphe des composantes alors il existe un chemin entre s et t pour tous $s \in C$ et $t \in C'$.

Exercice 4 Solution page 9

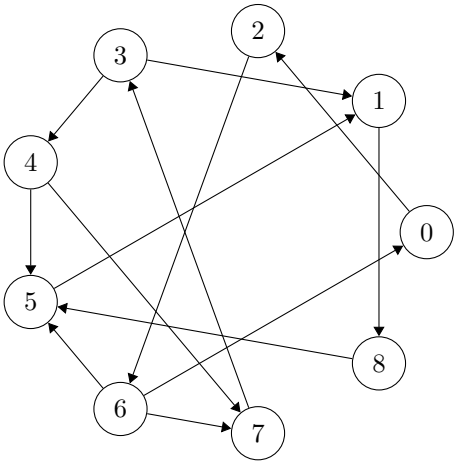
Prouver que le graphe des composantes est acyclique., le premier élément de C dans cet ordre.

Définition 2 : Graphe Transposé

Si $G = (S, A)$ est un graphe orienté, le graphe transposé de G est $G^\mathsf{T} = (S, A^\mathsf{T})$ avec $A^\mathsf{T} = \{(t, s) \mid (s, t) \in A\}$.

On retourne les arêtes.

Graphe G_0^T



Algorithme 3 : Calcul du graphe transposé

```
fonction transpose(G : graphe) =
    n ← nombre de sommets de G
    GT ← graohe vide à n sommets
    pour tous les sommets s de G faire
        pour tous les t voisins de s dans G faire
            ajouter s aux voisins de t dans GT
    retourner GT
```

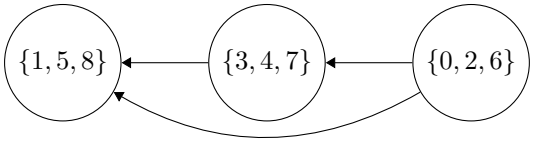
Exercice 5 Solution page 9

Prouver que les composantes fortement connexes de G^T sont celles de G.

Exercice 6

Prouver que le graphe des composantes de G^T est le graphe transposé du graphe des composantes de G.

Graphe des composantes de G_0^T



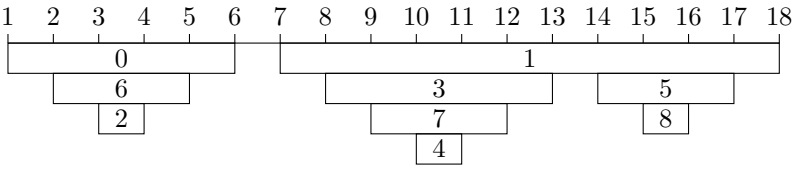
II.2 Ordres du parcours

Lors de l'exploration totale d'un graphe tous les sommets sont parcourus et il est naturel de penser à l'ordre d'appel des sommets. Cependant on a utilisé l'ordre de terminaison des appels pour calculer un ordre topologique.
On introduit ces deux ordres, préfixe et postfixe par les instants d'exécution dans l'algorithme 4.

Algorithme 4 : Exploration chronométrée de tous les sommet

```
fonction explorationTemps(s : sommet) =
    if pre[s] = −1 then
        pre[s] ← i
        incremente(i)
        pour tous les t voisins de s dans G faire explorer(t)
        post[s] ← i
        incremente(i)
fonction explorer-tout(G : graphe) =
    pour tous les sommets s de G faire
        pre[s] ← −1
        post[s] ← −1
    i ← 1 ; pour tous les sommets s de G faire explorationTemps(s)
    retourner pre, post
```

Exploration de G_0



L'ordre des sommets selon les $pre[s]$ croissants est l'odre *préfixe*, c'est l'ordre de découverte des sommets. L'ordre des sommets selon les $post[s]$ croissants est l'ordre *postfixe*, c'est l'ordre de clôture d'exploration des sommets.
Dans G_0, l'ordre préfixe est $(0, 6, 2, 1, 3, 7, 4, 5, 8)$ et l'ordre postfixe est $(2, 6, 0, 4, 7, 3, 8, 5, 1)$.
On peut immédiatement remarquer qu'on a $pre[s] < post[s]$.

Exercice 7 **Solution page 10**

Prouver que si on a $pre[s] < pre[t] < post[s]$ alors $post[t] < post[s]$.

Le théorème du chemin blanc se traduit par le fait que $pre[s] < pre[t] < post[s]$ si et seulement si il existe un chemin $s = s_0 \to s_1 \to \cdots \to s_{p-1} \to s_p = t$ tel que $pre[s] \leqslant pre[s_i]$ pour tout i

Théorème 2 de fin des C.F.C.

Le *point d'entrée* d'une composante fortement connexe est le sommet de cette composante pour lequel $pre[s]$ est minimal
Si s_C est le point d'entrée d'une composante fortement connexe C alors $post[s] \leqslant post[s_C]$ pour tout sommet s accessible depuis s_C dans le graphe.
En particuler s_C est le sommet de C pour lequel $post[s]$ est maximal.

Démonstration Soit $s \neq s_0$ un sommet accessible depuis s_0, on considère un chemin $s_0 \to s_1 \to \cdots \to s_{p-1} \to s_p = s$.

- Si aucun sommet du chemin n'a été visité, le théorème du chemin blanc et l'exercice 7 impliquent que $pre[s_0] < pre[s] < post[s] < post[s_0]$.
- S'il existe i tel que $pre[s_i] < pre[s_0]$ on va montrer qu'on a aussi $pre[s_p] < pre[s_0]$.
  En effet de $pre[s_i] < pre[s_0]$ on déduit, d'après la minimalité de $pre[s_0]$, que s_i n'appartient pas à la même C.F.C. que s_0 ; il n'existe donc pas de chemin de s_i vers s_0. Ainsi s_0 n'est pas exploré lors de l'exploration de s_i d'où $post[s_i] < pre[s_0]$.
  Pour $i \neq p$, s_{i+1} est un voisin de s_i donc il a déjà été exploré donc $pre[s_{i+1}] < pre[s_i] < pre[s_0]$ ou sera exploré entièrement lors de l'exploration de s_i donc
  $pre[s_i] < pre[s_{i+1}] < post[s_{i+1}] < post[s_i] < pre[s_0]$.
  Ainsi, dans les deux cas $pre[s_{i+1}] < pre[s_0]$ puis $pre[s_{i+2}] < pre[s_0]$, ..., jusqu'à $pre[s] = pre[s_p] < pre[s_0]$ d'où, comme ci-dessus, $post[s] < pre[s_0] < post[s]$.
- Il reste le cas évident $s = s_0$ donc $post[s] = post[s_0]$.

Dans tous les cas on a bien $post[s] \leqslant post[s_0]$. . . . . ∎

II.3 Algorithme

On veut rechercher, de manière efficace, les composantes fortement connexes.
L'objectif est de trouver un algorithme semblable à celui de la recherche des composantes connexes.

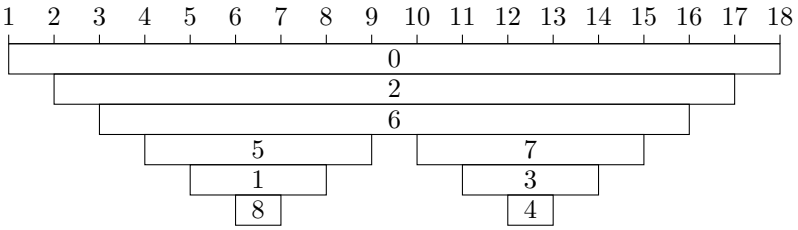
- Dans le cas d'un graphe non orienté, explorer depuis un point d'entrée d'une composante connexe visite tous les sommets de la composante et seulement ceux-ci. Alors que dans le cas d'un graphe orienté, explorer depuis un point d'entrée d'une composante fortement connexe visite tous les sommets de la composante mais peut visiter les sommets de composantes accessibles.
- Si on commence par une C.F.C. terminale dans le graphe des composantes, on ne visite que cette composante.
- De manière générale, si on a déjà visité toutes les composantes accessibles depuis une C.F.C., on ne visitera que cette composante.
- On voit donc qu'il suffit de s'assurer que les points d'entrée des C.F.C. soient dans un ordre topologique inverse c'est-à-dire dans l'ordre topologique du graphe transposé.

On considère alors l'ordre postfixe inversé du graphe transposé, noté σ.

Algorithme 5 : Calcul de σ

```
fonction explorerSigma(s : sommet) =
    if s est non vu then
        marquer s comme vu
        pour tous les t voisin de s dans GT faire explorerSigma(t)
        sig[i] ← s
        decremente(i)
fonction sigma(G : graphe) =
    n ← nombre de sommets de G
    GT ← transpose(G)
    sig ← tableau de taille n
    i ← n − 1
    pour tous les sommets s de G faire marquer s comme non vu
    pour tous les sommets s de G faire explorerSigma(s)
    retourner sig
```

Exploration de G_0^{T}



σ est donc l'ordre $(0, 2, 6, 7, 3, 4, 5, 1, 8)$.
L'algorithme de Kosaraju-Sharir consiste alors à explorer le graphe dans l'ordre σ chaque exploration partielle va donner une composante fortement connexe.

Algorithme 6 : Calcul des composantes fortement connexes

```
fonction explorerKS(s : sommet) =
    if cfc[s] = −1 then
        cfc[s] ← c
        pour tous les t voisin de s dans G faire explorerKS(t)
fonction KosaSha(G : graphe) =
    n ← nombre de sommets de G
    sig ← sigma(G)
    c ← 1
    pour tous les sommets s de G faire cfc[i] ← −1
    pour i de 0 à n − 1 faire
        explorerKS(sig[s])
        incremente(c)
    retourner cfc, c
```

Théorème 3 Preuve de l'algorithme

Toutes les composantes fortement connexes de G sont explorées une par une à chaque appel de `explorerSigma` (s) dans `explorerKS` dans un ordre topologique inverse du graphe des composantes.

DémonstrationOn note $CFC(s)$ la composante fortement connexe contenant s.
On montre que chaque appel de `explorer` (s) dans `explorer-tout` explore entièrement et uniquement $CFC(s)$.
On suppose que, au moment où appelle de `explorer` (s), on a parcouru une union de composantes fortement connexe, éventuellement 0 pour le premier appel.

- s est donc le premier sommet de $CFC(s)$ dans l'odre σ : aucun sommet de $CFC(s)$ n'été exploré. D'après le théorème du chemin blanc, tous les sommets de $CFC(s)$ vont être visités.
- Pour tout sommet d'une autre composantes fortement connexe, s', accessible depuis s, on note t le point d'entrée de $CFC(s')$ dans G^{T}. $CFC(t) = CFC(s')$ est accessible depuis $CFC(s)$ dans G donc $CFC(s)$ est accessible depuis $CFC(t)$ dans G^{T}. D'après théorème 2, t est situé avant s dans l'ordre σ donc $CFC(t)$ a été explorée lors de l'appel de `explorer` (s). Il n'y a donc pas d'autre sommet que ceux de $CFC(s)$ qui sont explorés lors l'appel de `explorer` (s).

Ainsi chaque appel `explorer` (s) dans `explorer-tout` explore entièrement et uniquement $CFC(s)$.
La démonstration ci-dessus montre en particulier que si C' est accessible depuis C dans le graphe des composantes alors C' est explorée avant C. . . . . ∎

On a ainsi un algorithme linéaire pour le calcul des composantes fortement connexes car les parcours

et le calcul du transposé sont linéaire dans le cas d'un graphe défini par tableau ou dictionnaire d'adjacence.

II.4 Résolution de 2-SAT

Définition 3 : problème 2-SAT

Une formule logique est sous forme k-FNC si elle est une conjonction de disjonction de littéraux (FNC) dont les clauses clause contiennent exactement k littéraux.
Le problème `2-SAT` est le problème de décision suivant : une formule booléenne φ sous forme 2-FNC est-elle satisfiable ?

À toute formule φ sous forme `2-SAT` on peut associer un graphe orienté $G_\varphi = (S, A)$:

- Si les variables utilisées dans φ sont $x_1, x_2, \ldots, x_n$, les sommets de G sont les littéraux associées $S = \{x_1, x_2, \ldots, x_n, \overline{x_1}, \overline{x_2}, \ldots, \overline{x_n}\}$.
- Pour chaque clause $\ell_1 \vee \ell_2$, on associe 2 arêtes : $\{\overline{\ell_1}, \ell_2\}$ et $\{\overline{\ell_2}, \ell_1\}$.

Pour éviter les ambiguïtés avec la notation $s \to t$ d'une arête, on note $\Rightarrow$ l'implication.

Exercice 8 **Solution page 10**

Prouver que s'il existe un chemin de ℓ à ℓ' dans G_φ alors $\varphi \vDash \ell \Rightarrow \ell'$.

Exercice 9 **Solution page 10**

Prouver que si les deux littéraux associés à une variable de φ sont dans une même composante fortement connexe de G_φ alors φ n'est pas satisfiable.

La réciproque est vraie aussi, cela donne un critère pour `2-SAT`.

Théorème 4

φ est satisfiable si et seulement il n'existe pas de variable x telle que x et $\overline{x}$ appartiennent à la même composante fortement connexe.

Démonstration Le sens direct est la contraposée de l'exercice 9.

S'il n'y a pas de paire $(x, \overline{x})$ dans les C.F.C on construit une valuation ν.
On considère les C.F.C. dans l'ordre inverse de leur découverte dans l'algorithme de Kosaraju-Sharir, c'est un ordre topologique. L'hypothèse signifie que $CFC(x) \neq CFC(\overline{x})$; on pose

- $\nu(x) =$ `faux` si $CFC(x)$ est avant $CFC(\overline{x})$ dans l'ordre topologique
- $\nu(x) =$ `vrai` sinon

Ainsi $\nu(\ell) =$ `faux` si et seulement si $CFC(\ell)$ précède (strictement) $CFC(\overline{\ell})$ pour tout littéral ℓ.
Si $\ell_1 \vee \ell 2$ est une clause avec $\nu(\ell_1) =$ `faux` alors $CFC(\ell_1)$ précède (strictement) $CFC(\overline{\ell_1})$, $CFC(\overline{\ell_1})$ précède $CFC(\ell_2)$ car il existe une arête entre $\overline{\ell_1}$ et ℓ_2 donc $CFC(\ell_1)$ précède (strictement) $CFC(\ell_2)$.
Si on a de plus $\nu(\ell_1) =$ `faux` alors, de même $CFC(\ell_2)$ précède (strictement) $CFC(\ell_1)$.
On aboutit à une contradiction donc au moins un des deux littéraux ℓ_1 ou ℓ_2 est évalué à `vrai` ; toute clause est donc évaluée à `vrai` d'où ν est un modèle de φ. . . . . ∎

On peut donc tester en temps linéaire si une formule de forme `2-SAT` est satisfiable.

Solutions

Exercice 1
On suppose que les sommets sont appelés dans l'ordre $s_0, s_1, \ldots, s_{n-1}$.
On note C_i la composante connexe contenant s_i.
s_0 n'est pas vu initialement donc l'algorithme appelle `explorer` (s_0). Aucun sommet n'est marqué comme vu donc, d'après le théorème du chemin blanc, les sommets visités sont tous les sommets accessibles depuis s_0, c'est-à-dire les sommets de C_0.
On note $\mathcal{P}(i)$ la propriété : après avoir traité les sommets de s_0 à s_i, les sommets vus sont les sommets de $C_0, \ldots, C_i$.
On vient de prouver $\mathcal{P}(0)$ et, si $\mathcal{P}(i)$ est vérifiée alors

- soit s_{i+1} a déjà été vu, d'après l'hypothèse de récurrence $s_{i+1} \in C_k$ avec $k \leqslant i$ et $\mathcal{P}(i+1)$ est valide,
- soit s_{i+1} n'a pas encore été vu, les sommets de C_{i+1} sont non vus d'après l'hypothèse de récurrence et l'appel de `explorer` (s_{i+1}) marquera comme vus tous les sommets de C_{i+1} : $\mathcal{P}(i+1)$ est valide.

Ainsi, par récurrence, $\mathcal{P}(i)$ est valide pour chaque i et la démonstration montre que chaque appel de `explorer` (s) marque comme vus tous les sommets de la composante connexe contenant s.

Exercice 2
Les deux cas correspondent à la situation de t lors de l'appel de `explorer` (s).

- Si t est marqué comme vu lors de l'appel de `explorer` (s) alors l'appel de `explorer` (t) a eu lieu avant celui de `explorer` (s). Or s n'est pas accessible depuis t car sinon on aurait un cycle en complétant ce chemin de t vers s par l'arête (s,t). Ainsi le théorème du chemin blanc implique que `explorer` (s) n'est pas appelé pendant l'exécution de `explorer` (t) : t a été exploré entièrement avant l'appel de `explorer` (s).
- Si t n'est pas marqué comme vu lors de l'appel de `explorer` (s) alors le théorème du chemin blanc implique que `explorer` (t) est appelé pendant l'exécution de `explorer` (s).

Exercice 3
Par récurrence sur la longueur du chemin.

- Si le chemin est de longueur 0, alors $C = C'$ et, par définition, il existe toujours un chemin entre deux sommets d'une C.F.C..
- On suppose que la propriété est vraie dans le cas de chemins de longueur p. S'il existe un chemin de longueur $p+1$ entre deux C.F.C. C et C', on considère le chemin de longueur p depuis C ; il finit en C''.

  Il existe une arête entre C'' et C' donc il existe une arête de G, (s'', s'), avec $s'' \in C''$ et $s' \in C'$.

  Par hypothèse de récurrence il existe un chemin entre s et s''.

  Dans la C.F.C. C', il existe un chemin entre s'' et t.

  On peut donc construire un chemin de s vers t.

Exercice 4
S'il existait un cycle dans le graphe des composantes $C_1 \to C_2 \to \cdots \to C_p = C_1$ alors, en considérant une arête (s_1, s_2) entre C_1 et C_2, l'exercice ci-dessus montrerait qu'il existe un chemin entre s_2 et s_1. Ainsi s_1 et s_2 serait dans la même C.F.C., ce qui est exclu.

Exercice 5
Si $s = s_0 \to s_1 \to \cdots \to s_{p-1} \to s_p = t$ est un chemin de s vers t dans G alors (s_{i-1}, s_i) est une arête de G donc (s_i, s_{i-1}) est une arête de G^T donc $t = s_p \to s_{p-1} \to \cdots \to s_1 \to s_0 = s$ est un chemin de t vers s dans G^T.
Ainsi, si s et t sont équivalents dans G, comme il existe un chemin de s vers t et un chemin de t vers s dans G, alors il existe un chemin de t vers s et un chemin de s vers t dans G^T donc s et t sont équivalents dans G^T.
Les classes d'équivalences, c'est-à-dire les composantes fortement connexes, sont donc les mêmes.

Exercice 7
Les conditions signifient que t est exploré durant l'exploration de t ; l'appel de `explorationTemps(`t`)` est donc effectué puis fini pendant l'appel de `explorationTemps(`s`)`.

Exercice 8
Si $\ell_1 \vee \ell_2$ est une clause de $\varphi \vDash \ell_1 \vee \ell_2$; de plus $\ell_1 \vee \ell_2 \equiv \overline{\ell_2} \Rightarrow \ell 1 \equiv \overline{\ell_1} \Rightarrow \ell 2$.
S'il existe un chemin $\ell = \ell_0 \to \ell_1 \to \cdots \to \ell_n = \ell'$ alors $\overline{\ell_{i-1}} \vee \ell_i$ est une clause pour tout $i \in \{1, 2, \ldots, n\}$ donc $\varphi \vDash \ell_{i-1} \Rightarrow \ell i$
On a alors $\varphi \vDash \big(\ell_0 \Rightarrow \ell 1\big) \wedge \cdots \wedge \big(\ell_{n-1} \Rightarrow \ell n\big) \vDash \ell_0 \Rightarrow \ell n$.

Exercice 9
Si x et $\overline{x}$ appartiennent à une composante fortement connexe de G_φ alors on a, d'après l'exercice 8, on a $\varphi \vDash x \to \overline{x} \equiv \overline{x}$ et $\varphi \vDash \overline{x} \to x \equiv x$. On en déduit $\varphi \vDash x \wedge \overline{x}$, soit $\varphi \vDash \bot$, donc φ n'est pas satisfiable.