GRAPHES AVANCÉS

À la fin de ce chapitre, je sais :

- ☞ expliquer le concept d'arbre recouvrant
- ☞ expliquer l'intérêt d'un tri topologique
- ☞ schématiser le concept de forte connexité
- ☞ expliquer ce qu'est un arbre recouvrant
- ☞ expliquer l'intérêt d'un graphe biparti

A Rappels sur les graphes

■ **Définition 1 — Graphe.** Un graphe G est un couple $G = (S, A)$ où S est un ensemble fini et non vide d'éléments appelés **sommets** et A un ensemble de paires d'éléments de S appelées **arêtes**.

■ **Définition 2 — Sous-graphe.** Soit $G = (S, A)$ un graphe, alors $G' = \{S', A'\}$ est un sous-graphe de G si et seulement si $S' \subseteq S$ et $A' \subseteq A$.

■ **Définition 3 — Sous-graphe couvrant.** G' est un sous-graphe couvrant de G si et seulement si G' est un sous-graphe de G et $S' = S$.

■ **Définition 4 — Graphe connexe.** Un graphe $G = (S, A)$ est connexe si et seulement si pour tout couple de sommets (a, b) de G, il existe une chaîne d'extrémités a et b.

ă

Théorème 1 — Condition nécessaire d'acyclicité d'un graphe. Soit un graphe $G = (S, A)$ possédant au moins une arête et acyclique alors G possède au moins deux sommets de degré un et on a :

$$|A| \leqslant |S| - 1 \tag{1}$$

Théorème 2 — Condition nécessaire de connexité d'un graphe. Si un graphe $G = (S, A)$ est connexe alors on a :

$$|A| \geqslant |S| - 1 \tag{2}$$

■ **Exemple 1 — Exercices types.** Soit $G = (S, A)$ un graphe non orienté dont l'ordre est n.

1. Montrer que si le G est connexe alors il comporte au moins $n-1$ arêtes. (Procéder par récurrence)
2. Montrer que si G a tous les sommets de degré supérieur ou égal à 2 alors il possède un cycle. En déduire, qu'un graphe acyclique admet un sommet de degré 0 ou 1.
3. Montrer que que si G est acyclique alors il possède au plus $n-1$ arêtes. (Procéder par récurrence)

B Arbres recouvrants

■ **Définition 5 — Arbre.** Un arbre est un graphe non orienté connexe et acyclique.

R On déduit des deux théorèmes précédents qu'un arbre possède **exactement** $|S| - 1$ arêtes.

Théorème 3 — Caractérisation des arbres. Soit G un graphe non orienté à n sommets. Les propositions suivantes sont équivalentes :

- G est un arbre
- G est connexe et acyclique
- G est connexe et possède $n-1$ arêtes
- G est connexe et la suppression de n'importe quelle arête le rend non connexe
- G est acyclique et possède $n-1$ arêtes
- G est acyclique et l'ajout de n'importe quelle arête crée un cycle
- entre toute paire de sommets de G il existe une unique chaîne dont les extrémités sont ces sommets

■ **Définition 6 — Forêt.** Une forêt est un graphe non orienté sans cycle.

■ **Définition 7 — Arbre recouvrant un graphe.** Dans un graphe non orienté et connexe, un arbre est dit recouvrant s'il est inclus dans ce graphe et s'il connecte tous les sommets de ce graphe.

R On peut dire de manière équivalente qu'un arbre recouvrant est :

- un sous-graphe acyclique maximal,
- un sous-graphe recouvrant connexe minimal.

Les arbres recouvrants sont des éléments essentiels de l'industrie, notamment du point de vue de l'efficacité, de la robustesse et de l'optimisation. En ce sens, ils sont également indispensables à toute vision durable de notre développement. L'idée d'un arbre recouvrant est de ne sélectionner que certaines arêtes dans un graphe pondéré afin de garantir un service optimal (en fonction des pondérations). Les arbres recouvrants sont donc utilisés dans les domaines des réseaux d'énergie, des télécommunications, des réseaux de fluides mais également en intelligence artificielle et en électronique. Les deux algorithmes phares pour construire des arbres recouvrants sont l'algorithme de Kruskal [**kruskal_shortest_1956**] et l'algorithme de Prim [**prim_shortest_1957**].

R Si les poids du graphe sont tous différents, alors il existe un unique arbre recouvrant de poids minimal.

a Algorithme de Prim

L'algorithme de Prim est un algorithme **glouton optimal** qui s'applique aux **graphes pondérés connexes**. Pour construire l'arbre, l'algorithme part d'un sommet et fait croitre l'arbre en choisissant un sommet dont la distance est la plus faible et n'appartenant pas à l'arbre, garantissant ainsi l'absence de cycle.

Algorithme 1 Algorithme de Prim, arbre recouvrant

```
1 : Fonction PRIM(G = (V, E, w))
2 :     visited ← s un sommet quelconque de V
3 :     F ← une file de priorités (sommet, cout)          ▷ cout nul 0 si s sommet, ∞ sinon
4 :     couts ← le tableau des coût pour atteindre chaque sommet depuis s
5 :     T ← ∅                          ▷ la sortie : l'ensemble des arêtes de l'arbre recouvrant
6 :     tant que F n'est pas vide répéter
7 :         (u,priority) ← min F                        ▷ Choix glouton ! Le coût le moindre
8 :         visited ← visited ∪{u}
9 :         pour chaque voisin v non visité de u répéter
10 :            w ← poids de l'arc (u, v)
11 :            si couts[v] > couts[u] + w alors
12 :                T ← T ∪ {(u, v, w)}
13 :                couts[v] ←couts[u] + w
14 :                INSERTION(F,(v,cout[v]))
15 :    renvoyer T
```

La complexité de cet algorithme, si l'on utilise un tas min pour le file et une liste d'adjacence pour représenter G, est en $O((n+m)\log n)$ si $m = |E|$ est le nombre d'arêtes du graphe et n

l'ordre du graphe.

R Doit-on utiliser Dijkstra ou Prim? En pratique, l'algorithme de Dijkstra est utilisé lorsque l'on souhaite économiser du temps et du carburant pour se déplacer d'un point à un autre. L'algorithme de Prim, quant à lui, est utilisé lorsque l'on souhaite minimiser les coûts de matériaux lors de la construction de routes reliant plusieurs points entre eux.

Les algorithmes de Prim et de Dijkstra présentent trois différences principalesă:

- Dijkstra trouve le chemin le plus court, mais l'algorithme de Prim trouve le l'arbre recouvrant minimal.
- Dijkstra s'applique sur les graphes orientés et non orientés mais Prim ne s'applique qu'à des graphes non orientés.
- Prim peut gérer des pondérations négatives alors que Dijkstra ne l'admet pas.

b Algorithme de Kruskal

L'algorithme de Kruskal (cf. algorithme 2) est un algorithme **glouton optimal** qui s'applique aux **graphes pondérés**. Le graphe peut ne pas être connexe et dans ce cas on obtient un **forêt** d'arbres recouvrants. Pour construire la forêt et effectuer un choix d'arête, il ordonne les arêtes du graphe par ordre de pondération croissante.

Algorithme 2 Algorithme de Kruskal, arbre recouvrant

```
1 : Fonction KRUSKAL(G = (V, E, w))
2 :     T ← ∅                          ▷ la sortie : l'ensemble des arêtes de l'arbre recouvrant
3 :     pour k de 1 à |E| répéter
4 :         e ← l'arête de pondération la plus faible de E          ▷ Choix glouton !
5 :         si (S, T ∪ {e}, w) est un graphe acyclique alors
6 :             T ← T ∪ {e}
7 :     renvoyer T
```

La complexité de cet algorithme, si l'on utilise un tas binaire, est en $O(n \log m)$ si $m = |E|$ est le nombre d'arêtes du graphe et n l'ordre du graphe.

C Tri topologique d'un graphe orienté

a Ordre dans un graphe orienté acyclique

Dans un graphe orienté acyclique, les arcs définissent un **ordre partiel**, le sommet à l'origine de l'arc pouvant être considéré comme le prédécesseur du sommet à l'extrémité de l'arc. Par exemple, sur la figure 1, a et b sont des prédécesseurs de d et e est un prédécesseur de g. Mais ces arcs ne disent rien de l'ordre entre e et h, l'ordre n'est pas total.

L'algorithme de tri topologique permet de créer un ordre total $\preceq$ sur un graphe orienté

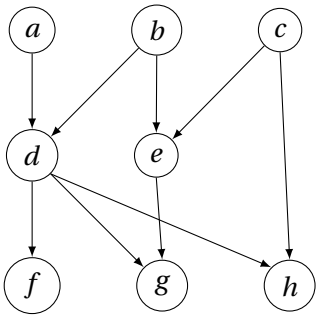


FIGURE 1 – Exemple de graphe orienté acyclique

acyclique. Formulé mathématiquement :

$$\forall (v,u) \in V^2, (v,u) \in E \Longrightarrow v \preceq u \tag{3}$$

Sur l'exemple de la figure 1, plusieurs ordre topologiques sont possibles. Par exemple :

- a,b,c,d,e,f,g,h
- a,b,d,f,c,h,e,g

b Tri topologique et détection de cycles dans un graphe orienté

L'algorithme de tri topologique (cf. algorithme 3) permet de construire un ordre dans un graphe orienté acyclique. C'est en fait un parcours en profondeur du graphe qui construit une pile en ajoutant le concept de date à chaque sommet : un date de début qui correspond au début du traitement du sommets et une date de fin qui correspond à la fin du traitement du sommet par l'algorithme. La pile contient à la fin les sommets dans un ordre topologique, les sommets par ordre de date de fin de traitement.

Au cours du parcours en profondeur, un sommet passe tout d'abord de l'ensemble des sommets non traités à l'ensemble des sommets en cours de traitement (date de début). Puis, lorsque la descente est finie (plus aucun arc ne sort du sommet courant), le sommet passe de l'ensemble en cours de traitement à l'ensemble des sommets traités (date de fin).

D Composantes fortement connexes d'un graphe orienté et 2-SAT

■ **Définition 8 — Composante fortement connexe d'un graphe orienté** $G = (V,E)$**.** Une composante fortement connexe d'un graphe orienté G est un sous-ensemble S de ses sommets, maximal au sens de l'inclusion, tel que pour tout couple de sommets $(s,t) \in S$ il existe un chemin de s à t dans G.

Le calcul des composantes connexes d'un graphe est par exemple utilisé pour résoudre le problème 2-SAT. Dans le cadre de ce problème, on dispose d'une formule logique sous la forme conjonctive normale et chaque clause comporte deux variables. L'idée est de construire

Algorithme 3 Algorithme de tri topologique

```
1 : Fonction TOPO_SORT(G = (V, E))
2 :     pile ← une pile vide                          ▷ Contiendra les sommets dans l'ordre topologique
3 :     états ← un tableau des états des sommets  ▷ pas traité, en cours de traitement ou traité
4 :     dates ← un tableau des dates associées aux sommets
5 :     Les cases du tableau états sont initialisées à «pas traité»
6 :     Les cases du tableau dates sont initialisées à max_int  ▷ Date inconnue représentée par
    max_int
7 :     pour chaque sommet v de G répéter
8 :         si v n'est pas traité alors
9 :             TOPO_DFS(G = (V, E), pile, états, dates, v, 0)
10 :    renvoyer (pile, dates)
11 :
12 : Fonction TOPO_DFS(G = (V, E), pile, états, dates, v, date)
13 :    états[v] ← «en cours de traitement»
14 :    dates[v] ← date                          ▷ Au début de l'exploration la date de v vaut date
15 :    pour chaque voisin u de v répéter
16 :        si u n'est pas traité alors TOPO_DFS(G = (V, E), pile, états, dates, u, (date + 1))
17 :    états[v] ← «traité»
18 :    dates[v]+ = 1                          ▷ Pour distinguer le début de la fin de l'exploration
19 :    EMPILER(v, pile)
```

un graphe à partir de la formule de cette formule F. Supposons qu'elle soit constituée de m clauses et n variables $(v_1, v_2, \ldots, v_n)$. On élabore alors un graphe $G = (V, E)$ à $2n$ sommets et $2m$ arêtes. Les sommets représentent les n variables v_i ainsi que leur négation $\neg v_i$. Les arêtes sont construites de la manière suivante : on transforme chaque clause de F de la forme $v_i \vee v_j$ en deux implications $\neg v_1 \Longrightarrow v_2$ ou $\neg v_2 \Longrightarrow v_1$. Cette transformation utilise le fait que la formule $a \Longrightarrow b$ est équivalent à $\neg a \vee b$.

Théorème 4 F n'est pas satisfaisable si et seulement s'il existe une composante fortement connexe contenant une variable v_i et sa négation $\neg v_i$.

Démonstration. ($\Longleftarrow$) S'il existe une composante fortement connexe contenant a et $\neg a$, alors cela signifie $F : (a \Longrightarrow \neg a) \wedge (\neg a \Longrightarrow a)$. Or cette formule n'est pas satisfaisable. En effet, si a est vrai alors $(a \Longrightarrow \neg a)$ est faux, car du vrai on ne peut pas conclure le faux d'après la définition sémantique de l'implication. De même, si a est faux alors $(\neg a \Longrightarrow a)$ est faux, pour la même raison. Dans tous les cas, la formule est fausse. F n'est pas satisfaisable.

($\Longrightarrow$), par contraposée. Supposons qu'il n'existe pas de composante fortement connexe contenant a et $\neg a$. Cela peut se traduire en la formule suivante :

$$F : (\neg(a \Longrightarrow \neg a) \wedge \neg(\neg a \Longrightarrow a)) \vee ((a \Longrightarrow \neg a) \wedge \neg(\neg a \Longrightarrow a)) \vee ((\neg a \Longrightarrow a) \wedge \neg(a \Longrightarrow \neg a)) \quad (4)$$

Ce qui signifie :

- soit il n'existe aucun chemin de a à $\neg a$,
- soit il existe un chemin dans un seul sens, mais pas dans les deux.

Cette formule F est toujours satisfaisable. En effet, si a est vrai, alors $\neg a \Longrightarrow a$ est vraie, car *ex falso quodlibet*, et donc F est vraie. Si a est faux, alors $a \Longrightarrow \neg a$ est vraie pour la même raison. Par ailleurs, si a est vrai, $a \Longrightarrow \neg a$ est fausse, car on ne peut pas déduire le faux du vrai. Dans tous les cas, F est vraie. On peut également le montrer en simplifiant la formule. F est donc satisfaisable. ■

On peut montrer que SAT-2 est un problème de décision polynomial. C'est une restriction à des clauses de deux variables du problème général SAT qui lui est NP-complet.

E Graphes bipartis et couplage maximum

Théorème 5 — Caractérisation des graphes bipartis. Un graphe est biparti si et seulement s'il ne possède aucun cycle de longueur impair.

Pour savoir si un graphe est biparti on pourrait donc rechercher les cycles et évaluer leurs longueurs. Il existe cependant une autre approche. Un graphe biparti est un graphe bicolorable comme le montre la figure 2.

a Couplage dans un graphe biparti

■ **Définition 9 — Couplage.** Un couplage Γ dans un graphe non orienté $G = (V, E)$ est un ensemble d'arêtes deux à deux non adjacentes. Formellement :

$$\forall (e_1, e_2) \in E^2, e_1 \neq e_2 \Longrightarrow e_1 \cap e_2 = \emptyset \tag{5}$$

c'est à dire que les sommets de e_1 et e_2 ne sont pas les mêmes.

■ **Définition 10 — Sommets couplés, sommets exposés.** Un sommet est couplé s'il fait parti d'une arête de Γ. Un sommet est exposé s'il ne fait pas parti des arêtes de Γ, c'est-à-dire il n'est pas couplé.

■ **Définition 11 — Couplage maximal.** Un couplage maximal est tel que si on lui ajoute une arête, ce n'est plus un couplage. Il est donc maximal au sens de l'inclusion.

■ **Définition 12 — Couplage de cardinal maximum.** Un couplage de cardinal maximum est un couplage contenant le plus grand nombre d'arêtes possible.

■ **Exemple 2 — Affectation des cadeaux sous le sapin .** Au pied du sapin de Noël, un papa a disposé six cadeaux dont les paquets sont tous différents et numérotés de 0 à 5 [a]. Il a décidé que les cadeaux seraient répartis en fonction des paquets que les enfants préfèrent.

Ses cinq enfants expriment donc leurs préférences. Le papa pourra-t-il affecter un cadeau à chaque enfant et faire en sorte que ce cadeau soit un de leurs préférés ?

Évidemment la réponse à cette question dépend des préférences émises par les enfants. Supposons qu'ils se soient exprimés ainsi :

Alix 0,2

Brieuc 1,3,4,5

Céline 1,2

Dimitri 0,1,2

Enora 2

On peut représenter par un graphe biparti cette situation comme sur la figure 2. Dans ce cas précis, comme il y a quatre enfants qui ne veulent que trois les trois premiers cadeaux, il n'y a pas de solution. Mais si Enora avait choisit 4 et 5 ???

a. Ce papa est informaticien !

R Si les arêtes du graphe sont valuées (graphe pondéré), alors la recherche d'un couplage de cardinal maximum et de poids minimum dans un graphe biparti est en fait le problème de l'**affectation des ressources**. Ceci n'est pas au programme.

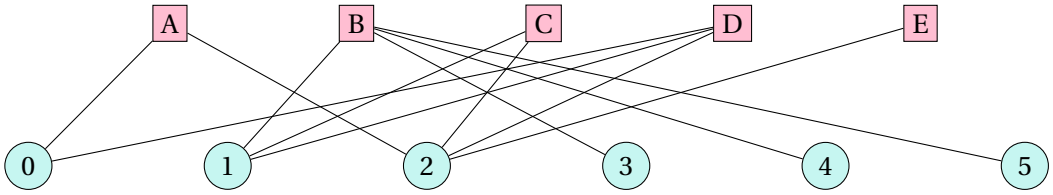


FIGURE 2 – Exemple de graphe biparti pour un problème d'affectation sans solution.

b Chemin augmentant

Pour résoudre le problème de trouver un couplage de cardinal maximum, on peut procéder en suivant l'algorithme 4. Il s'agit de construire un chemin augmentant pour atteindre un couplage de cardinal maximum.

■ **Définition 13 — Chemin alternant.** Une chemin alternant dans un graphe non orienté G et pour un couplage Γ est tel que les arêtes appartiennent successivement à Γ et $E \setminus \Gamma$.

■ **Définition 14 — Chemin augmentant.** Un chemin augmentant est un chemin alternant dont les extrémités sont des sommets exposés, c'est-à-dire qui n'appartiennent pas au couplage Γ.

La stratégie de l'algorithme de recherche d'un couplage de cardinal maximum est la suivante : à partir d'un couplage Γ, on construit un nouveau couplage de cardinal supérieur à l'aide d'un chemin augmentant comme le montre la figure 3.

Dans un graphe **biparti**, il est facile d'augmenter la taille d'un couplage jusqu'au cardinal maximum :

1. s'il existe deux sommets exposés reliés par une arête, il suffit d'ajouter cette arête au couplage. Puis, on appelle récursivement l'algorithme sur ce nouveau couplage.
2. sinon il faut trouver un chemin augmentant π dans le graphe.
   (a) s'il n'y en a pas, l'algorithme est terminé.
   (b) sinon on effectue la différence symétrique entre le couplage Γ et l'ensemble des arêtes du chemin augmentant π pour obtenir le nouveau couplage : $\Gamma \oplus \pi = \{e \in E, e \in \Gamma \setminus \pi \cup e \in \pi \setminus \Gamma\}$. Puis, on appelle récursivement l'algorithme avec ce nouveau couplage.

Il faut noter que le cardinal du couplage n'augmente pas nécessairement lorsqu'on effectue la différence symétrique mais il ne diminue pas.

Pour trouver un chemin augmentant dans un graphe biparti $G = ((U, D), E)$, on cherche le plus court chemin entre deux sommets exposés dans un graphe orienté auxiliaire G_o construit de la manière suivante :

1. toutes les arêtes de E qui n'appartiennent pas au couplage Γ sont orientés de U vers D.
2. toutes les arêtes de Γ sont orientées de D vers U.

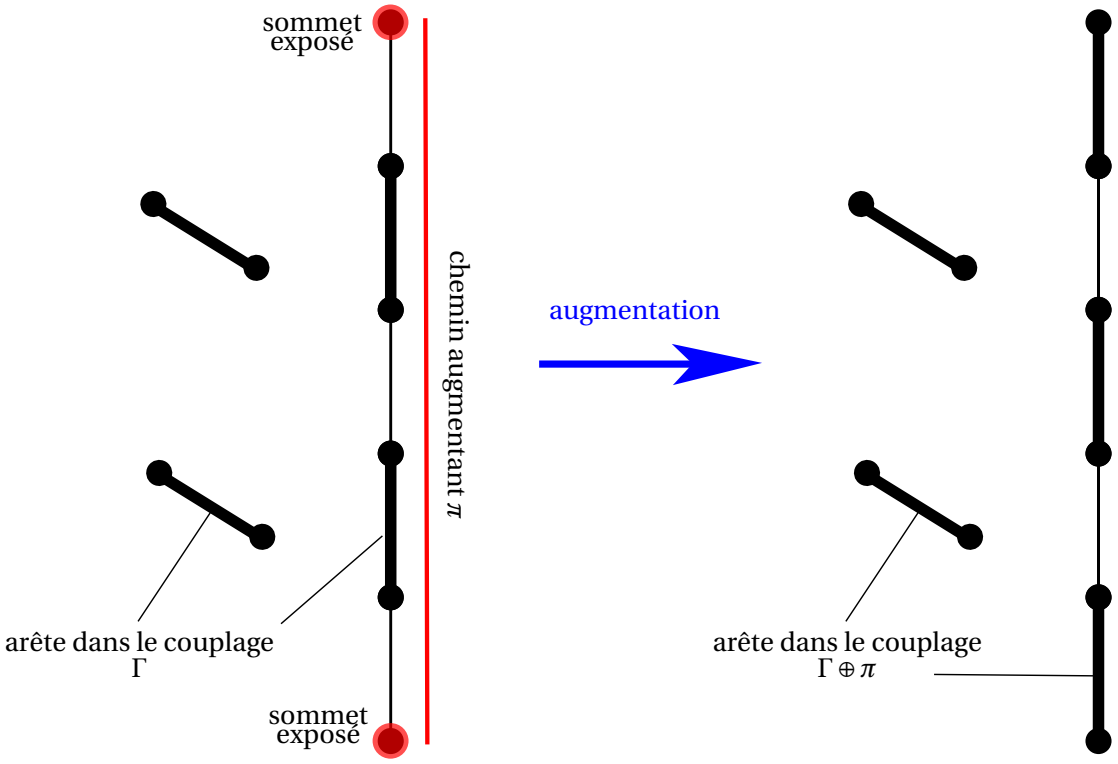
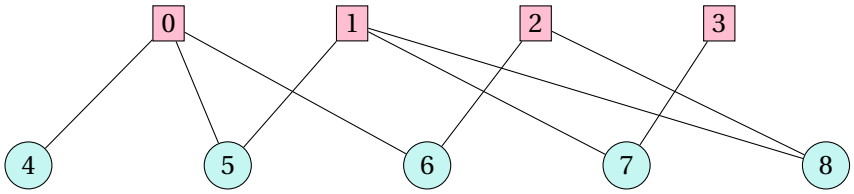


FIGURE 3 – Illustration de la construction d'un couplage de cardinal supérieur grâce à un chemin augmentant. (Source originale : Wikimedia Commons)

Le plus court chemin entre deux sommets exposés de G_o est nécessairement un chemin augmentant, son caractère alternant vient du fait que le graphe est biparti.

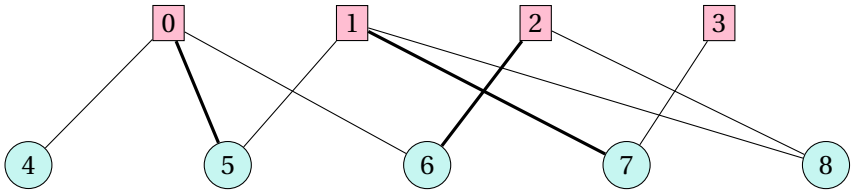
La figure 4 illustre les différentes étapes de l'algorithme. On vérifie bien que le résultat est correct : chaque enfant aura bien un cadeau qu'il avait mis sur sa liste.

Le graphe de départ de l'algorithme est le suivant :



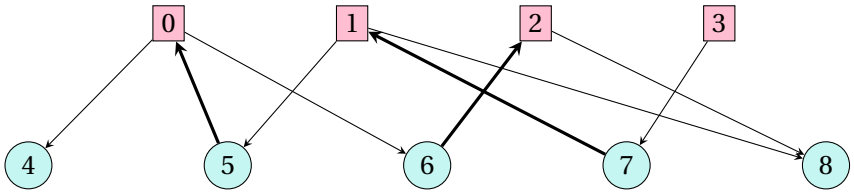
$\Gamma = \emptyset$

On effectue **trois appels récursifs** et, à chaque fois, on a trouvé une arête dont les sommets sont tous les deux exposés.



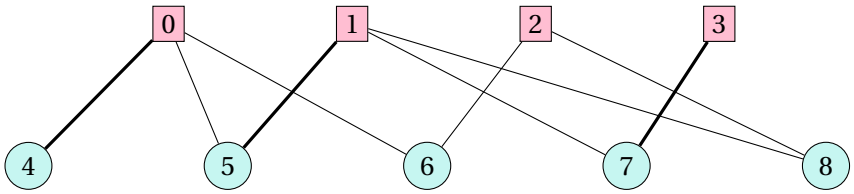
$\Gamma = \{(0,5),(1,7),(2,6)\}$

À ce stade de l'algorithme, aucun sommet exposé n'est relié par une arête à un autre sommet exposé. Donc, on construit le graphe G_o d'après le couplage Γ. On trouve le chemin le plus court entre les deux premiers sommets exposés 3 et 4 : π.



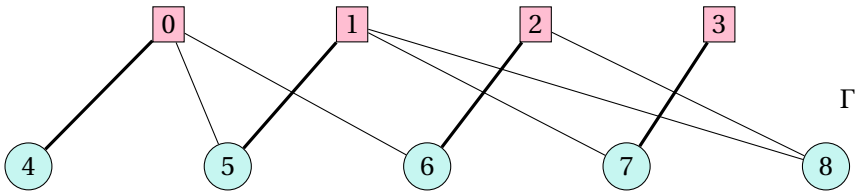
$\Gamma = \{(0,5),(1,7),(2,6)\}$
$\pi = \{3,7,1,5,0,4\}$

On en déduit un nouveau couplage $\Gamma = \Gamma \oplus \pi$:



$\Gamma = \{(0,4),(1,5),(3,7)\}$

On effectue **un appel récursif** et on trouve une arête dont les sommets sont tous les deux exposés : $(2,6)$. On effectue un dernier appel récursif et l'algorithme se termine car un seul sommet est non couplé.



$\Gamma = \{(0,4),(1,5),(3,7),(2,6)\}$

FIGURE 4 – Étapes de l'algorithme de recherche d'un couplage de cardinal maximum

Algorithme 4 Recherche d'un couplage de cardinal maximum

Entrée : un graphe biparti $G = ((U, D), E)$
Entrée : un couplage Γ initialement vide
Entrée : F_U, l'ensemble de sommets exposés de U initialement U
Entrée : F_D, l'ensemble de sommets exposés de D initialement D

```
1 : Fonction CHEMIN_AUGMENTANT(G = (V = (U, D), E), Γ, F_U, F_D))  ▷ M est le couplage, vide initialement
2 :     si une arête e = (u, v) entre un sommet de F_U et un sommet de F_D existe alors
3 :         CHEMIN_AUGMENTANT(G, Γ ∪ {e}, F_U \ {u}, F_D \ {v})
4 :     sinon
5 :         Créer le graphe orienté G_o                ▷ ∀e ∈ E, e de U vers D si e ∉ Γ, l'inverse sinon
6 :         Calculer le plus court chemin π entre un sommet de F_U et un de F_D dans G_o
7 :         si un tel chemin π n'existe pas alors
8 :             renvoyer M
9 :         sinon
10 :            CHEMIN_AUGMENTANT(G, Γ ⊕ π, F_U \ {π_start}, F_D \ {π_end})
11 :                                ▷ π_start début du chemin π, π_end fin du chemin π et Γ ⊕ π = {e ∈ E, e ∈ Γ \ π ∪ e ∈ π \ Γ}
```