IFT436 – Algorithmes et structures de données

Retour sur la force brute

Problèmes étudiés :

— Chemins hamiltoniens dans un graphe

— Ensembles dominants minimaux

Séance du 11 novembre 2025

Université de Sherbrooke

Lors de la séance du 11 novembre 2025, les sujets de la force brute et des graphes ont été discutés. Durant cette séance, deux problèmes ont été couverts :

(1) Un *chemin hamiltonien* est un chemin qui passe par chaque sommet d'un graphe exactement une fois. Donnez un algorithme qui détermine s'il existe un chemin hamiltonien dans un graphe $\mathcal{G}$.

(2) Un *ensemble dominant* d'un graphe $\mathcal{G} = (V, E)$ est un sous-ensemble de sommets D tel que chaque sommet V du graphe appartient à D ou est adjacent à au moins un sommet de D. Donnez un algorithme qui détermine le plus petit ensemble dominant d'un graphe $\mathcal{G} = (V, E)$.

Je vous invite à essayer ces deux problèmes par vous-mêmes avant de poursuivre. Vous trouverez dans la suite du document une démarche de résolution détaillée qui, je l'espère, vous permettra de mieux comprendre les notions abordées.

Si jamais vous avez des questions ou des commentaires, n'hésitez pas à me contacter à l'adresse suivante :
benjamin.courchesne@usherbrooke.ca.
Dernière compilation : 17 novembre 2025.

(1) Un *chemin hamiltonien* est un chemin qui passe par chaque sommet d'un graphe exactement une fois. Donnez un algorithme qui détermine s'il existe un chemin hamiltonien dans un graphe $\mathcal{G}$.

Solution.

Rappel force brute

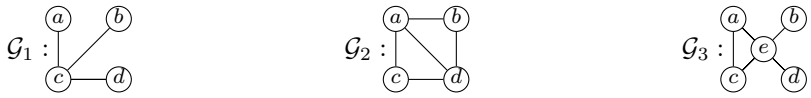
La méthode force brute consiste à explorer l'ensemble des solutions possibles pour un problème donné. Pour cela il faut donc *deux ingrédients* :
1. Une méthode pour générer toutes les solutions possibles
2. Une méthode pour vérifier si une solution est valide

Ensuite, lors de l'implémentation, on parcourt l'ensemble des solutions possibles **(étape 1)** et on utilise **(étape 2)** pour vérifier si la solution courante est valide.

Avant d'écrire un algorithme, assurons-nous de bien comprendre ce qu'est un chemin hamiltonien.

Rappel : Un chemin hamiltonien visite chaque sommet *exactement une fois*, sans jamais revenir sur ses pas.

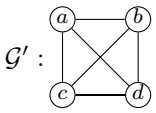
Voici trois exemples de graphes pour illustrer ce concept :



- $\mathcal{G}_1$: Essayons différents sommets de départ. Si on commence par a, b ou d (qui ont chacun un seul voisin), on doit obligatoirement passer par c en deuxième. Mais une fois à c, il reste deux sommets à visiter, et on ne peut en atteindre qu'un seul ! C'est une impasse. **Conclusion :** il n'existe pas de chemin hamiltonien dans $\mathcal{G}_1$.
- $\mathcal{G}_2$: Il existe un chemin hamiltonien : $a \to c \to d \to b$.
- $\mathcal{G}_3$: Ce graphe contient un triangle (a–c–e). Si on commence par parcourir ce triangle, on rencontre le même problème que dans $\mathcal{G}_1$ pour visiter b et d. Si on commence par e (le centre), on doit choisir entre aller vers $\{a, c\}$ ou vers $\{b, d\}$, et on ne pourra jamais revenir visiter l'autre côté. **Conclusion :** il n'existe pas de chemin hamiltonien dans $\mathcal{G}_3$.

Maintenant que nous avons compris intuitivement le problème, appliquons la méthode force brute. Rappelons qu'il nous faut **deux ingrédients** :

1. Dans le contexte des chemins Hamiltoniens, un ensemble de toutes les solutions possibles est l'ensemble de tous les chemins Hamiltoniens dans le graphe complet $\mathcal{G}'$ ayant les V sommets. **Pourquoi utiliser le graphe complet ?** L'idée la suivante : si un chemin hamiltonien existe dans $\mathcal{G}$, il existera forcément aussi dans $\mathcal{G}'$ (puisqu'on a *ajouté* des arêtes, pas enlevé). En générant tous les chemins possibles dans $\mathcal{G}'$, on est sûr de ne manquer aucune solution potentielle de $\mathcal{G}$.



Dans $\mathcal{G}'$, tous les sommets sont reliés entre eux. Un chemin hamiltonien dans ce graphe est simplement un ordre de visite des sommets.

Cela revient à générer toutes les permutations des V sommets (que nous noterons $perm(V)$).

Pourquoi des permutations ? Une permutation est un ordre différent des sommets. Par exemple, $[a, b, c, d]$ signifie « visiter a, puis b, puis c, puis d ».

exemple avec $V = \{a, b, c, d\}$:

$$\begin{aligned} perm(\{a,b,c,d\}) = \{&[a,b,c,d],[a,b,d,c],[a,c,b,d],[a,c,d,b],[a,d,b,c],[a,d,c,b],\\ &[b,a,c,d],[b,a,d,c],[b,c,a,d],[b,c,d,a],[b,d,a,c],[b,d,c,a],\\ &[c,a,b,d],[c,a,d,b],[c,b,a,d],[c,b,d,a],[c,d,a,b],[c,d,b,a],\\ &[d,a,b,c],[d,a,c,b],[d,b,a,c],[d,b,c,a],[d,c,a,b],[d,c,b,a]\} \end{aligned}$$

(24 permutations au total puisque $|perm(V)| = |V|!$)

($\Rightarrow$) **(Ingrédient 1)** : Nous savons maintenant générer toutes les solutions possibles, soit $perm(V)$.

2. Maintenant il nous faut une méthode pour vérifier si une solution (un chemin hamiltonien candidat) est valide. Pour cela, il suffit de vérifier que chaque arête entre deux sommets consécutifs dans le chemin existe dans le graphe $\mathcal{G}$.

   ($\Rightarrow$) **(Ingrédient 2)** : Nous savons maintenant vérifier si une solution est valide dans $\mathcal{G}$.

Algorithme 1 - Validation de chemin

1. **Entrée :** Les arêtes E d'un graphe $\mathcal{G} = (V, E)$ et un chemin candidat $C = [c_1, c_2, \ldots, c_{|V|}]$
2. **Sortie :** Vrai si C est un chemin *réalisable* dans $\mathcal{G}$, Faux sinon
3.
4. **fonction** EST_CHEMIN_VALIDE(E, C) :
5. **pour** $i \in [1..(|C| - 1)]$, **faire** :
6. **si** $(c_i, c_{i+1}) \notin E$ **alors**
7. **renvoyer** Faux
8. **fin si**
9. **fin pour**
10. **renvoyer** Vrai
11. **fin fonction**

Analyse de la complexité : à la ligne 5, l'algorithme fait n itérations où n est le nombre de sommets dans le graphe. À chaque itération, on regarde si un couple (c_i, c_{i+1}) est dans l'ensemble des arêtes E (cela se fait en temps constant si E est représenté comme une matrice d'adjacence). La complexité de l'algorithme 1 est donc $\mathcal{O}(n)$.

En d'autres mots : vérifier un chemin demande de parcourir tous ses sommets une fois, donc un temps proportionnel à n.

Pour compléter notre algorithme force brute, combinons maintenant les deux ingrédients : parcourons toutes les permutations des sommets (Algorithme 2) et vérifions chacune avec notre fonction de validation (Algorithme 1).

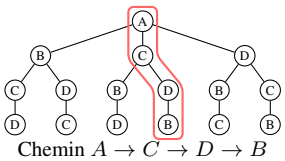
Algorithme 2 - Chemin Hamiltonien par force brute

1. **Entrée :** Un graphe $\mathcal{G} = (V, E)$
2. **Sortie :** Vrai s'il existe un chemin hamiltonien dans $\mathcal{G}$, Faux sinon
3.
4. **fonction** EXISTE_CHEMIN_HAMILTONIEN($\mathcal{G} = (V, E)$) :
5. **pour tout** $C \in perm(V)$ **faire**
6. **si** EST_CHEMIN_VALIDE(E, C) **alors**
7. **renvoyer** Vrai
8. **fin si**
9. **fin pour**
10. **renvoyer** Faux
11. **fin fonction**

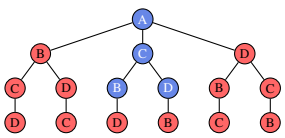
Analyse de la complexité : Pour un ensemble de n éléments, il existe $n!$ permutations possibles (voir l'exemple avec la construction de $\mathcal{G}'$ à la page précédente). Ainsi, à la ligne 5, l'algorithme fait $n!$ itérations. Pour chaque itération, il appelle l'algorithme 1 qui a une complexité de $\mathcal{O}(n)$. La complexité totale de l'algorithme 2 est donc $\mathcal{O}(n \cdot n!)$.

Intuition d'amélioration. Lors de la première solution, nous avons généré l'ensemble des permutations des sommets du graphe complet $\mathcal{G}'$ pour ensuite vérifier chacune d'entre elles. Dans le pire cas, il n'y a aucune permutation valide, et nous devons toutes les vérifier. De manière visuelle, on pourrait représenter l'ensemble des permutations sous la forme d'arbres.

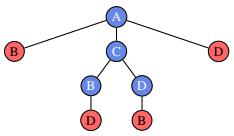
Dans le graphe $\mathcal{G}'_1$ il est possible de commencer avec le sommet A et de continuer vers B ou C ou D. Les chemins sont représentés par des branches dans l'arbre des permutations.



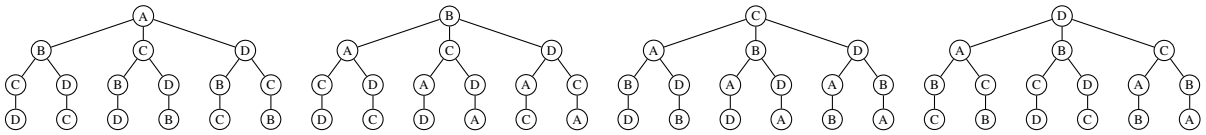
On peut de manière *itérative* se promener dans l'arbre en partant de la racine et en explorant chaque branche. À chaque sommet visité, on se demande si le sous-chemin menant à ce sommet est valide dans le graphe $\mathcal{G}_1$. Si ce n'est pas le cas, on marque le sommet en rouge (indiquant qu'il n'y a pas de chemin hamiltonien passant par ce sous-chemin) sinon on le marque en bleu.



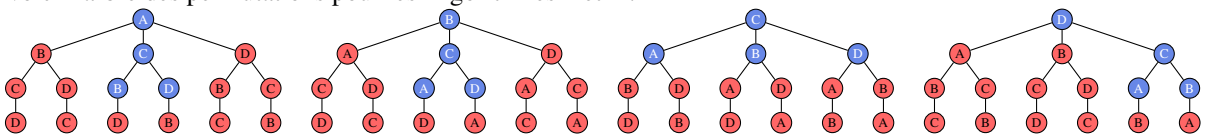
Si le sous-chemin $A \to B$ est impossible, est-ce possible qu'un chemin hamiltonien passe par $A \to B$? Non, donc on peut se dire qu'il n'est pas nécessaire de continuer à explorer cette branche. Cela nous permet d'éliminer toutes les permutations commençant par $A \to B$.



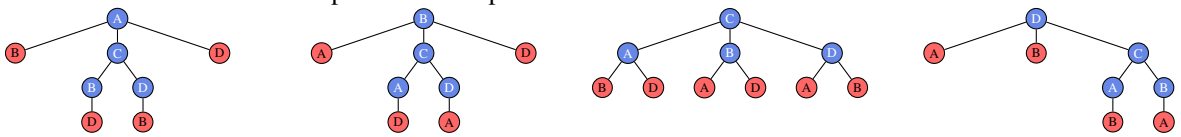
Soit pour le graphe $\mathcal{G}_1$ l'arbre des permutations suivant :



Voici l'arbre des permutations pour les Algorithmes 1 et 2 :



Voici maintenant les arbres de permutations après avoir éliminé les sous-chemins invalides :



Dans la première version (Algorithme 1 et 2) nous devions tester exactement $n!$ permutations pour trouver un chemin hamiltonien valide (ou conclure qu'il n'en existe pas). En retirant préemptivement toutes les permutations ayant un sous-chemin invalide, nous pouvons réduire le nombre de permutations à tester.

Pour le graphe $\mathcal{G}_1$, nous avions initialement $n \cdot n!$ comparaisons à faire (*96* puisque $n = |V| = |\{A, B, C, D\}| = 4$). Suite aux éliminations nous avons 12 + 12 + 18 + 12 = *54 comparaisons* à faire.

(2) Un *ensemble dominant* d'un graphe $\mathcal{G} = (V, E)$ est un sous-ensemble de sommets D tel que chaque sommet V du graphe appartient à D ou est adjacent à au moins un sommet de D. Donnez un algorithme qui détermine le plus petit ensemble dominant d'un graphe $\mathcal{G} = (V, E)$.

Solution.

Interlude formel

Lors de la présentation en classe de ce problème, j'ai utilisé la notation mathématique suivante pour définir un ensemble dominant.

$$D \subseteq V \mid \forall s \in V, (s \in D) \vee (\exists u \in D, (u, s) \in E)$$

Je me suis rendu compte que les notations par compréhension d'ensemble n'étaient pas toujours claires pour tout le monde. Voici donc une explication détaillée de cette notation.

$\boldsymbol{D \subseteq V} \mid \forall s \in V, (s \in D) \vee (\exists u \in D, (u, s) \in E)$

D est un sous-ensemble de V tel que ...

On pourrait faire une analogie avec un langage de programmation typé. À gauche du symbole $\mid$ on trouve le type de l'objet que l'on décrit (ici D qui est un sous-ensemble de V) et à droite du symbole $\mid$ on trouve la condition que doit respecter cet objet (comme une sorte de `assert` en programmation).

Par exemple, en C++ on pourrait écrire :

```cpp
// La partie à gauche du '|' : type de D
std::set<int> D = ...;

// La partie à droite du '|' : condition que doit respecter D
for (auto s : V) {
  bool ok = D.contains(s) ||
            std::any_of(D.begin(), D.end(),
                        [&](int u) { return E.contains({u, s}); });
  assert(ok); // on vérifie que la condition est respectée
}
```

Pour l'exemple, voici une seconde traduction d'une notation par compréhension :

$$x \in \mathbb{N} \mid x > 42$$

se traduirait en C++ par :

```cpp
int x = ...; // x est un entier naturel
assert(x > 42); // on vérifie que la condition est respectée
```

$D \subseteq V \mid \boldsymbol{\forall s \in V, (s \in D) \vee (\exists u \in D, (u, s) \in E)}$

... pour tout sommet s appartenant à V, ... (on doit vérifier la suite (en orange) pour chaque sommet de V, s'il y en a un seul pour lequel la partie en orange est fausse, alors D ne respecte pas la condition pour être un ensemble dominant)

$D \subseteq V \mid \forall s \in V, \boldsymbol{(s \in D)} \vee (\exists u \in D, (u, s) \in E)$

... soit le sommet s appartient à D, ...

$D \subseteq V \mid \forall s \in V, (s \in D) \vee \boldsymbol{(\exists u \in D, (u, s) \in E)}$

... soit il existe un sommet u appartenant à D tel que $(u, s) \in E$ (c'est-à-dire que u est adjacent à s).

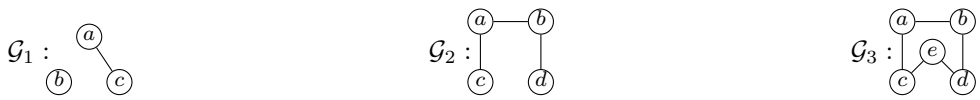
La traduction est donc :

D est un sous-ensemble de V tel que pour tout sommet s appartenant à V, soit le sommet s appartient à D, soit il existe un sommet u appartenant à D tel que $(u, s) \in E$ (c'est-à-dire que u est adjacent à s).

Avant d'écrire un algorithme, assurons-nous de bien comprendre ce qu'est un ensemble dominant.

Rappel : Un ensemble dominant D est un sous-ensemble de sommets tel que *tous* les sommets du graphe sont soit dans D, soit voisins d'au moins un sommet de D.

Voici trois graphes pour lesquels nous allons identifier tous leurs ensembles dominants :



$\mathcal{G}_1$:

$D = \{\}$		non-dominant car b n'est pas dans D et D est vide.
$D = \{a\}$		non-dominant car b n'est pas dans D et n'est pas adjacent à a
$D = \{b\}$		non-dominant car c n'est pas dans D et n'est pas adjacent à b
$D = \{c\}$		non-dominant car b n'est pas dans D et n'est pas adjacent à c
$D = \{a, b\}$		dominant car a et b sont dans D et c est adjacent à a
$D = \{a, c\}$		non-dominant car b n'est pas dans D et n'est pas adjacent à a ou c
$D = \{b, c\}$		dominant car b et c sont dans D et a est adjacent à c
$D = \{a, b, c\}$		dominant car tous les sommets sont dans D

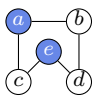
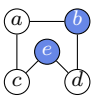
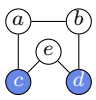
L'ensemble dominant le plus petit est donc $\{a, b\}$ ou $\{b, c\}$.

$\mathcal{G}_2$: Par économie d'espace, en rouge un sommet donnant un contre-exemple pour l'ensemble dominant, et en bleu les sommets de D.

$D = \{\}$		$D = \{b, c\}$	
$D = \{a\}$		$D = \{b, d\}$	
$D = \{b\}$		$D = \{c, d\}$	
$D = \{c\}$		$D = \{a, b, c\}$	
$D = \{d\}$		$D = \{a, b, d\}$	
$D = \{a, b\}$		$D = \{a, c, d\}$	
$D = \{a, c\}$		$D = \{b, c, d\}$	
$D = \{a, d\}$		$D = \{a, b, c, d\}$	

Les ensembles dominants sont : $\{a, b\}, \{a, d\}, \{b, c\}, \{c, d\}, \{a, b, c\}, \{a, b, d\}, \{a, c, d\}, \{b, c, d\}$ ou $\{a, b, c, d\}$

$\mathcal{G}_3$: Voici les trois ensembles dominants minimaux :

Maintenant que nous avons compris intuitivement le problème, appliquons la méthode force brute. Rappelons qu'il nous faut **deux ingrédients** :

1. Dans le contexte des ensembles dominants, un ensemble de toutes les solutions possibles est l'ensemble de toutes les combinaisons possibles de sommets (comme listé à la page précédente pour le calcul de l'ensemble dominant minimal de $\mathcal{G}_1$). Cela peut être représenté par $P(V)$ [1] pour V l'ensemble des sommets ou par 2^V [2].

   Pourquoi générer toutes les combinaisons ? Chaque sous-ensemble de sommets est un ensemble dominant potentiel. En les testant tous, on garantit de trouver le plus petit.

   Par exemple, pour $V = \{a, b\}$ nous aurons $P(V) = 2^V = \{\{\}, \{a\}, \{b\}, \{a, b\}\}$.

   ($\Rightarrow$) **Ingrédient 1** : Nous savons maintenant générer toutes les solutions possibles, soit $P(V)$.
2. Maintenant il nous faut une méthode pour vérifier si une solution (un sous-ensemble de V) est dominant. Pour cela, chaque sommet de V doit respecter au moins une des contraintes suivantes :
   (a) Le sommet est dans l'ensemble dominant
   (b) Le sommet a au moins un voisin dans l'ensemble dominant

   ($\Rightarrow$) **Ingrédient 2** : Nous savons maintenant vérifier si une solution est valide dans $\mathcal{G}$.

Algorithme 3 - Validation ensemble dominant

```
1.  Entrée : Un graphe G = (V, E) et un ensemble dominant candidat D
2.  Sortie : Vrai si D est un ensemble dominant, Faux sinon
3.
4.  fonction EST_DOMINANT(G = (V, E), D) :
5.      pour s ∈ V, faire :
6.          si s ∉ D alors
7.              voisins ← {u ∈ V | (u, s) ∈ E}
8.              si voisins ∩ D = ∅ alors
9.                  renvoyer Faux
10.             fin si
11.         fin si
12.     fin pour
13.     renvoyer Vrai
14. fin fonction
```

Analysons le fonctionnement de l'algorithme :

Étape 1. L'algorithme 3 est une écriture sous format de pseudo-code de la contrainte $\forall s \in V, (s \in D) \vee (\exists u \in D, (u, s) \in E)$. Puisque nous sommes face à un pour tout $(\forall s \in V)$, on doit *tester l'ensemble des valeurs dans V* et *si une seule valeur évalue à faux alors le $\forall \ldots$ évalue à faux*.

```
1.  fonction EST_DOMINANT(G = (V, E), D) :
2.      pour s ∈ V, faire :
3.          si s ∉ D alors
4.              voisins ← {u ∈ V | (u, s) ∈ E}
5.              si voisins ∩ D = ∅ alors
6.                  renvoyer Faux
7.              fin si
8.          fin si
9.      fin pour
10.     renvoyer Vrai
11. fin fonction
```

Étape 2. Si le *sommet est dans D*, il satisfait la contrainte (a). Nous n'avons donc pas besoin de faire d'autre vérification

$$vrai \vee ? = vrai$$

```
1.  fonction EST_DOMINANT(G = (V, E), D) :
2.      pour s ∈ V, faire :
3.          si s ∉ D alors
4.              voisins ← {u ∈ V | (u, s) ∈ E}
5.              si voisins ∩ D = ∅ alors
6.                  renvoyer Faux
7.              fin si
8.          fin si
9.      fin pour
10.     renvoyer Vrai
11. fin fonction
```

1. Cet opérateur se nomme le *powerset* en anglais
2. La notation 2^V est utilisée pour donner un clin d'œil au nombre d'éléments qui constitue toutes les combinaisons possibles, soit $2^{|V|}$ éléments

Étape 3. Les voisins d'un sommet s sont *l'ensemble de tous les sommets du graphe ayant une arête vers s*. Soit pour un sommet u, ce sommet fait partie des voisins s'il y a l'arête (u, s) dans l'ensemble des arêtes noté E.

```
1.  fonction EST_DOMINANT(G = (V, E), D) :
2.      pour s ∈ V, faire :
3.          si s ∉ D alors
4.              voisins ← {u ∈ V | (u, s) ∈ E}
5.              si voisins ∩ D = ∅ alors
6.                  renvoyer Faux
7.              fin si
8.          fin si
9.      fin pour
10.     renvoyer Vrai
11. fin fonction
```

Étape 4. *S'il n'y a aucun voisin qui est dans D* (que ce sont deux ensembles disjoints), alors la contrainte (b) est fausse. Puisque nous regardons seulement si la contrainte (b) est fausse, on retourne faux (revoir l'explication à la première étape de l'algorithme).

```
1.  fonction EST_DOMINANT(G = (V, E), D) :
2.      pour s ∈ V, faire :
3.          si s ∉ D alors
4.              voisins ← {u ∈ V | (u, s) ∈ E}
5.              si voisins ∩ D = ∅ alors
6.                  renvoyer Faux
7.              fin si
8.          fin si
9.      fin pour
10.     renvoyer Vrai
11. fin fonction
```

Analyse de la complexité : L'analyse peut sembler contre-intuitive au premier abord. La boucle externe (`pour` $s \in V$`, faire`) s'exécute $|V|$ fois. À chaque itération, on construit l'ensemble $voisins$. C'est souvent ici que le calcul de complexité devient difficile.

Si le graphe utilise une liste d'adjacence, accéder aux voisins de s prend un temps $O(deg(s))$ où $deg(s)$ est le degré du sommet s. En faisant cela pour *chaque* sommet s dans V, on obtient une complexité totale de $O\left(\sum_{s \in V} deg(s)\right) = O(2|E|) = O(|E|)$ où $|E|$ est le nombre d'arêtes dans le graphe.

Pourquoi $\sum deg(s) = 2|E|$ *?* Chaque arête contribue au degré de ses deux extrémités. En sommant tous les degrés, on compte donc chaque arête exactement deux fois.

Ainsi, on fait $|V|$ tours de boucle et à chaque tour, on fait des opérations en $O(deg(s))$. La complexité totale de l'algorithme est donc $O(|V| + |E|)$.

En d'autres mots : vérifier un ensemble dominant demande de parcourir tous les sommets et potentiellement toutes les arêtes, donc un temps proportionnel à la taille du graphe.

Pour compléter notre algorithme force brute, combinons maintenant les deux ingrédients : générons toutes les combinaisons possibles de sommets (Algorithme 4) et vérifions chacune avec notre fonction de validation (Algorithme 3).

Algorithme 4 - Ensemble dominant minimal par force brute

```
1.  Entrée : Un graphe G = (V, E)
2.  Sortie : Un ensemble dominant minimal D
3.
4.  fonction ENSEMBLE_DOMINANT_MINIMAL(G = (V, E)) :
5.      meilleur_D ← V
6.      pour D ∈ P(V), faire :
7.          si EST_DOMINANT(G, D) et |D| < |meilleur_D| alors
8.              meilleur_D ← D
9.          fin si
10.     fin pour
11.     renvoyer meilleur_D
12. fin fonction
```

Analyse de la complexité : Pour un ensemble de n sommets, il existe 2^n combinaisons possibles (l'ensemble des parties $P(V)$). Ainsi, la boucle principale fait 2^n itérations. Pour chaque itération, on appelle l'algorithme 3 qui a une complexité de $O(|V| + |E|)$. La complexité totale de l'algorithme 4 est donc $O(2^n \cdot (|V| + |E|))$.