Jeu d'isolation de propagation dans un graphe de flux

TER M1 Informatique
Isolation de propagation dans les graphes orientés acycliques

Auteur : Ethan MACHE et Théo JOLY
Encadrant : Dominique BARTH

Université Versailles Saint-Quentin-en-Yvelines
Année universitaire 2024–2025

Table des matières

1 Présentation du problème **2**

2 Modélisation du problème **2**
2.1 Données du problème . . . 2
2.2 Dynamique de propagation . . . 2
2.3 Action de suppression . . . 2
2.4 Objectif . . . 2
2.5 Schéma de propagation . . . 2
2.6 Conclusion . . . 3

3 Génération de graphe aléatoire **4**
3.1 Génération d'un arbre connexe aléatoire . . . 4
3.1.1 Exemple de génération d'un arbre aléatoire . . . 4
3.2 Génération d'un graphe orienté acyclique (DAG) connexe . . . 4
3.2.1 Exemple de génération d'un DAG connexe . . . 5

4 Cas particulier : propagation dans un arbre orienté **6**
4.1 Algorithme 1 (glouton) — Suppression du plus grand sous-arbre en priorité . . . 6
4.1.1 Exemple d'exécution de l'algorithme 1 . . . 7
4.1.2 Exemple montrant une limite de l'algorithme 1 . . . 8
4.2 Algorithme 2 (glouton) — Suppression du plus grand nombre de fils en priorité . . . 11
4.2.1 Exemple montrant une limite de l'algorithme 2 . . . 11
4.3 Algorithme 3 (heuristique) — Recuit simulé . . . 13
4.3.1 Fonction auxiliaire : Choisir voisin . . . 13
4.4 Algorithme 4 (heuristique) — Tabu Search . . . 15
4.4.1 Exemple d'exécution de l'Algorithme 4 sur un arbre . . . 17
4.5 Algorithme 5 (heuristique) - Aléatoire pour isolation de propagation . . . 18
4.5.1 Exemple d'exécution . . . 19
4.6 Algorithme 6 (heuristique) — Randomisé ε-greedy . . . 20
4.6.1 Exemple d'exécution de l'Algorithme 6 . . . 21

5 Cas général : propagation dans un graphe orienté acyclique (DAG) **23**
5.1 Algorithme 7 (glouton) - Plus grand sous DAG . . . 23
5.2 Algorithme 8 (heuristique) — Recuit simulé . . . 24
5.2.1 Fonction auxiliaire : Choisir voisin . . . 25
5.3 Algorithme 9 — Tabu Search étendu aux DAGs . . . 26
5.3.1 Exemple d'exécution de l'Algorithme 9 — Tabu Search étendu aux DAGs . . . 27
5.4 Algorithme 10 — Randomisé ε-greedy . . . 29

6 Résultats obtenus **30**
6.1 Résultats moyens sur 100 simulations . . . 30
6.1.1 DAG 1 . . . 31
6.1.2 DAG 2 . . . 34
6.1.3 DAG 3 . . . 37
6.2 Conclusion des expérimentations . . . 39

7 Conclusion générale du TER **40**

1 Présentation du problème

On dispose d'un arbre ou d'un DAG dans lequel il y a un sommet dit infecté au temps 0. À chaque temps, l'infection va se propager aux sommets voisins des sommets infectés, mais on a la possibilité de couper un arc n'importe où à chaque étape. Le but final étant de maximiser le nombre de sommets sains à la fin de la propagation

2 Modélisation du problème

2.1 Données du problème

On considère un graphe connexe $G = (V, E, \tau)$ où :
— V est l'ensemble des sommets,
— E est l'ensemble des arcs,
— $\tau : E \to \mathbb{Z}^{+*}$ est une fonction associant à chaque arc $[x, y] \in E$ un temps de propagation $\tau([x, y])$.
Un sommet $r \in V$ est initialement infecté à l'instant $t = 0$.

2.2 Dynamique de propagation

— À l'étape 0, seul le sommet r est infecté : $I_0 = \{r\}$.
— Si un sommet x est infecté à l'instant t, alors pour tout arc $[x, y] \in E$, le sommet y devient infecté à l'instant $t + \tau([x, y])$ si cet arc n'est pas supprimé.

2.3 Action de suppression

— À chaque étape, il est possible de supprimer un arc du graphe pour bloquer la propagation par ce lien.
— un arc supprimé à l'instant t' empêche toute propagation ultérieure via cet arc.

2.4 Objectif

Déterminer une séquence d'arcs à supprimer $e_1, e_2, \ldots, e_k$ (un par étape), de manière à maximiser le nombre de sommets non infectés après stabilisation.

Fonction objectif

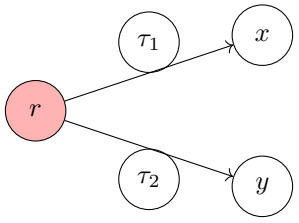
$$\max_{e(1),e(2),\ldots,e(k)} \quad |V| - |I_\infty|$$

où I_∞ est l'ensemble final des sommets infectés.

Contraintes

$$\forall [x, y] \in E, \quad \text{si } x \text{ est infecté à } t \text{ et } e(t') = [x, y] \text{ avec } t' \leq t + \tau([x, y]) \Rightarrow y \text{ n'est pas infecté}$$

2.5 Schéma de propagation



Sans suppression, les sommets x et y sont infectés à $t = \tau_1$ et $t = \tau_2$, respectivement. En supprimant l'arc $[r, y]$ avant $t = \tau_2$, on peut sauver y.

2.6 Conclusion

On cherche à construire une stratégie de suppression d'arcs pour ralentir et stopper la propagation, en maximisant le nombre de sommets sauvés dans le graphe.

3 Génération de graphe aléatoire

3.1 Génération d'un arbre connexe aléatoire

L'objectif est de générer aléatoirement un arbre connexe, de façon uniforme, c'est-à-dire que chaque arbre possible ait la même probabilité d'être produit.

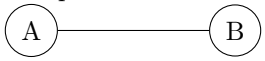
Pour cela, on suit le principe suivant :
- On commence par choisir un sommet initial qui jouera le rôle de racine de l'arbre.
- Ensuite, on ajoute les autres sommets un par un. À chaque étape, un nouveau sommet est relié à un sommet déjà présent dans l'arbre (c'est-à-dire un sommet déjà intégré à l'arbre, qu'il ait un père ou qu'il s'agisse de la racine).
- Ce choix est effectué de manière aléatoire parmi les sommets déjà insérés, garantissant une probabilité équitable pour chaque structure d'arbre possible.
- On répète ce processus jusqu'à ce que tous les sommets soient ajoutés à l'arbre. À chaque étape, un nouveau sommet est relié à un sommet déjà présent dans l'arbre. Ce procédé garantit que la structure reste acyclique et connexe. La racine, définie initialement, est le seul sommet qui ne possède pas de parent.

Ce procédé assure la construction d'un arbre couvrant, connexe, et sans cycle, tout en respectant une distribution uniforme sur l'ensemble des arbres possibles.

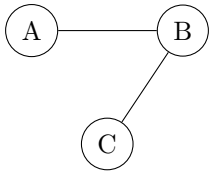
3.1.1 Exemple de génération d'un arbre aléatoire

Voici un exemple d'arbre construit progressivement à partir d'un sommet racine :

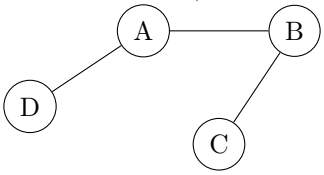
Étape 1 : On commence par choisir le sommet A comme racine.



Étape 2 : On ajoute le sommet B et on le connecte à A.



Étape 3 : On ajoute le sommet C, connecté aléatoirement à B.



Étape 4 : Le sommet D est ajouté et connecté à A.

On obtient ainsi un arbre connexe sans cycle, construit de manière incrémentale et aléatoire.

3.2 Génération d'un graphe orienté acyclique (DAG) connexe

L'objectif est de générer aléatoirement un graphe orienté acyclique (DAG) connexe, de manière uniforme, c'est-à-dire que chaque structure valide ait la même probabilité d'apparaître.

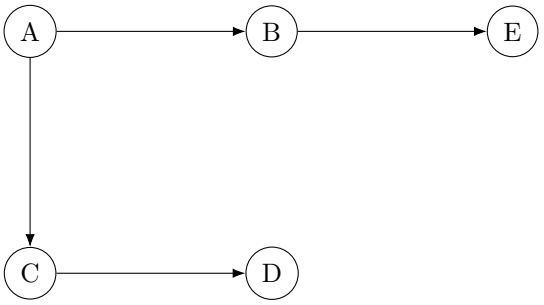
Pour cela, on procède en deux étapes :

— **Étape 1 : génération d'un arbre aléatoire.** On commence par construire un arbre couvrant connexe (voir section précédente). Cet arbre garantit une base structurelle sans cycle et connectée.
— **Étape 2 : ajout d'arcs supplémentaires sans créer de cycles.** Pour enrichir la structure et former un DAG, on considère chaque paire de sommets (u, v) telle que v a été ajouté après u lors de la construction de l'arbre. Avec une certaine probabilité p, on ajoute un arc orienté $(u \to v)$. Ce choix de direction et d'ordre temporel évite toute création de cycle, et permet de conserver l'acyclicité du graphe.

Ce procédé permet de générer un DAG connexe, tout en conservant une distribution aléatoire contrôlée sur l'ensemble des structures possibles.

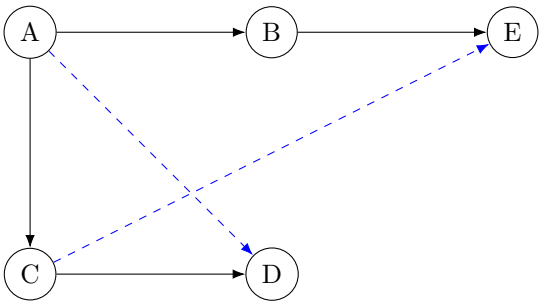
3.2.1 Exemple de génération d'un DAG connexe

Étape 1 : génération d'un arbre aléatoire (structure de base)



Cette structure est un arbre connexe. Chaque sommet a été ajouté un par un, en se connectant à un sommet déjà présent, sans créer de cycle.

Étape 2 : ajout d'arcs orientées pour former un DAG



Les arcs en pointillés bleus sont ajoutées pour enrichir le graphe. Elles respectent l'ordre d'insertion (le sommet source a été ajouté avant le sommet cible), ce qui garantit qu'aucun cycle n'est formé.

4 Cas particulier : propagation dans un arbre orienté

Pour amorcer notre étude, nous avons choisi de nous concentrer sur un cas particulier plus simple : la propagation dans un **arbre orienté**. Cette structure, acyclique par définition, permet une première analyse plus intuitive du phénomène de diffusion et des stratégies d'isolation.

Dans un arbre orienté, chaque sommet (sauf la racine) possède exactement un prédécesseur, et la propagation suit les arcs dirigés vers les successeurs. L'absence de *circuits* garantit qu'une fois qu'un arc est supprimé, les sommets situés en aval de cet arc deviennent définitivement inaccessibles, c'est-à-dire protégés de toute contamination ultérieure.

Ainsi, une stratégie efficace consiste à identifier, à chaque niveau de profondeur de l'arbre, l'arc dont la suppression permet d'isoler le plus grand sous-arbre possible. Supprimer plusieurs arcs à une même profondeur ne présente généralement aucun intérêt, puisque la propagation continue uniquement via les arcs non supprimés vers les niveaux inférieurs.

Cette première approche dans une structure stricte et hiérarchisée nous fournit une base solide avant de généraliser nos algorithmes au cas plus complexe des **graphes orientés acycliques (DAGs)**.

4.1 Algorithme 1 (glouton) — Suppression du plus grand sous-arbre en priorité

Afin d'initier notre réflexion algorithmique, nous avons conçu une première approche simple mais efficace, exploitant la structure hiérarchique des arbres orientés. L'idée centrale consiste à, à chaque étape, supprimer l'arc dont la coupure permet de bloquer la propagation vers le plus grand sous-arbre possible.

Cette stratégie repose sur une logique intuitive : en interrompant la contamination vers les zones les plus étendues dès les premiers niveaux, on maximise le nombre de sommets sauvés. L'algorithme applique ce raisonnement de manière récursive, en ciblant à chaque niveau l'arc le plus « rentable » en termes de sommets isolés.

L'algorithme suivant formalise cette stratégie :

Algorithm 1: PlusGrandSousArbreDAbord(ls, res)

Entrée: ls : liste de sommets
Sortie: res : liste d'arcs maximales

```
1  maxGlobal ← 0;
2  aTraiter ← [ ];
3  arcGlobale ← NIL;
4  Pour chaque sommet s dans ls faire
5      max ← 0;
6      arc ← NIL;
7      Pour chaque fils fs de s faire
8          ajouter fs à aTraiter;
9          nb ← Calcul(fs);
10         Si nb > max alors
11             max ← nb;
12             arc ← (s, fs);
13     Si max > maxGlobal alors
14         maxGlobal ← max;
15         arcGlobale ← arc;
16 Si arcGlobale ≠ NIL alors
17     ajouter arcGlobale à res;
18     retirer le sommet de l'arcGlobale dans aTraiter;
19 Retourner res, maxGlobal + PlusGrandSousArbreDAbord(aTraiter, res);
```

Fonction auxiliaire : Calcul

Cette fonction permet de calculer la taille du sous-arbre enraciné en un sommet donné. Elle est utilisée par l'algorithme précédent pour choisir quel arc supprimer.

Algorithme permettant de calculer le nombre de sommets dans un sous arbre

Algorithm 1: Calcul(s : sommet)

```
[H] Entrée: s : un sommet
    Sortie: Le nombre de sommets dans le sous-arbre enraciné en s
Si s.fils = ∅ alors
    Retourner 1;
nombre ← 1;
Pour chaque fils fs de s faire
    nombre ← nombre + Calcul(fs);
Retourner nombre;
```

Principe de l'algorithme :

- Pour chaque sommet de la liste en cours, on examine ses fils.
- Pour chaque fils, on calcule la taille du sous-arbre dont il est la racine (c'est-à-dire le nombre de sommets descendants).
- L'arc menant au plus grand sous-arbre est identifié comme le meilleur candidat à couper.
- Cet arc est ajouté à la solution, et l'algorithme se poursuit récursivement sur les sous-arbres restants.

Ce fonctionnement récursif permet à l'algorithme de bloquer, étape après étape, les propagations les plus menaçantes, en maximisant à chaque itération le nombre de sommets sauvés.

Complexité temporelle :

La complexité de cet algorithme repose sur deux opérations principales :

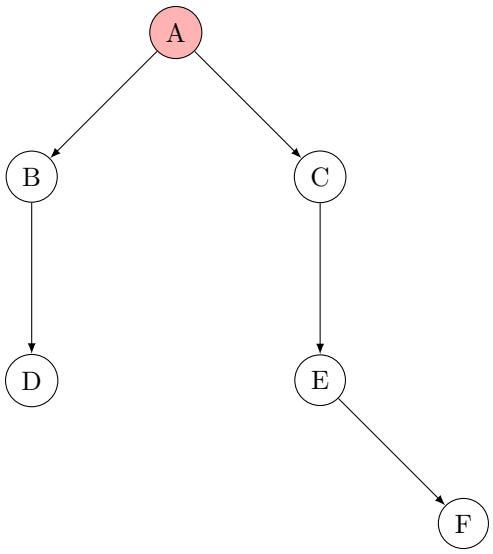
- Le parcours des sommets et de leurs fils, effectué une fois, est en $\mathcal{O}(n)$, avec n le nombre total de sommets.
- La fonction `Calcul(fs)` est appelée pour chaque fils, et parcourt récursivement son sous-arbre pour en compter les sommets, soit jusqu'à $\mathcal{O}(n)$ opérations dans le pire des cas.

Ainsi, dans le pire scénario — par exemple dans un arbre dégénéré (type liste chaînée) — la complexité globale est de l'ordre de $\mathcal{O}(n^2)$.

En pratique, cette approche reste raisonnablement efficace sur des arbres équilibrés ou de taille modérée, ce qui en fait un bon point de départ pour développer des heuristiques plus avancées.

4.1.1 Exemple d'exécution de l'algorithme 1

Considérons l'arbre orienté suivant, où le sommet A est initialement infecté :



L'arbre contient 6 sommets et est orienté de la racine A vers les feuilles. Le sommet A est infecté initialement à l'instant $t = 0$.

Étapes de l'algorithme :

— L'algorithme commence à la racine A.
  — Sous-arbre de B : {B, D} → taille = 2
  — Sous-arbre de C : {C, E, F} → taille = 3
  ⇒ On supprime l'arc **(A, C)** pour sauver C, E et F.
— On traite ensuite le sous-arbre restant contenant A, B, D.
  — À partir de B, le fils D forme un sous-arbre de taille 1.
  ⇒ On supprime l'arc **(B, D)** pour sauver D.

Arcs supprimés : (A, C) et (B, D)

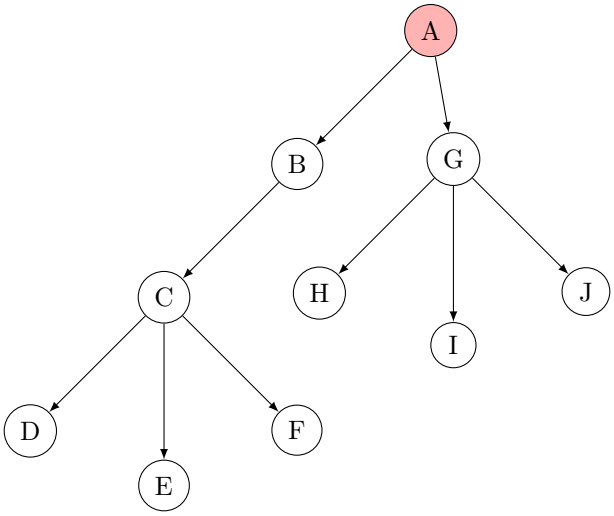
Sommets sauvés de la propagation : C, E, F, D

Sommets infectés au final : A, B

Cet exemple montre que l'algorithme choisit de supprimer en priorité les arcs menant aux sous-arbres les plus vastes. Cela maximise le nombre de sommets protégés dès les premières étapes.

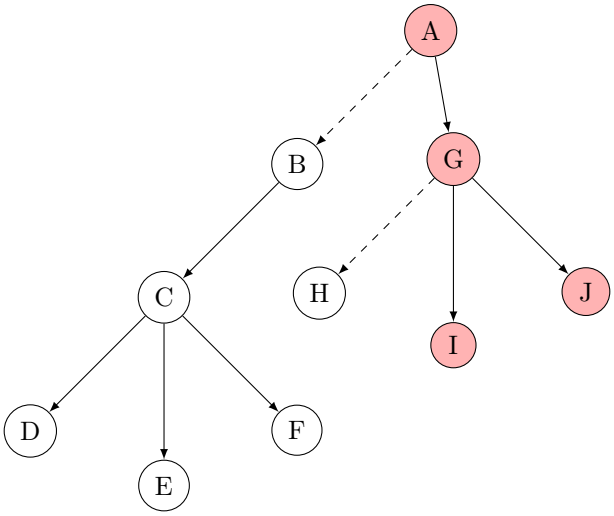
4.1.2 Exemple montrant une limite de l'algorithme 1

Considérons l'arbre orienté suivant :



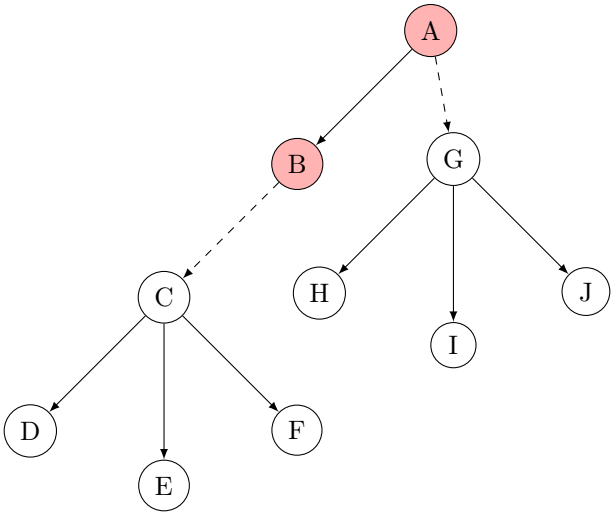
L'arbre est orienté de la racine A vers les feuilles. Le sommet A est initialement infecté à l'instant $t = 0$.

Analyse du choix effectué par l'algorithme :
— Le sous-arbre gauche (linéaire) contient 5 sommets : {B, C, D, E, F}.
— Le sous-arbre droit (compact) contient 4 sommets : {G, H, I, J}.
— L'algorithme 1 choisit donc de supprimer l'arc (A, B), bloquant ainsi la branche gauche (la plus grande en nombre de sommets).
— Il choisit ensuite de couper un des 3 arcs qui vont de G vers H, I, J.



Conséquence du choix de l'algorithme :
— Branche gauche bloquée immédiatement : 5 sommets sauvés (B, C, D, E, F).
— Branche droite bloquée :
  — À $t = 1$, G est infecté.
  — À $t = 2$, 2 des sommets H, I et J (fils de G) sont infectés.
Au total, seuls 7 sommets sont protégés.

Choix alternatif optimal : Si l'algorithme avait supprimé l'arc (A, G) au lieu de (A, B) :
— La branche droite aurait été immédiatement protégée : 4 sommets sauvés (G, H, I, J).
— La propagation dans la branche gauche étant lente (chaîne linéaire), il aurait été possible de supprimer un arc après la contamination, sauvant ainsi 8 sommets au lieu de 7.



Conclusion : L'algorithme 1 privilégie la taille du sous-arbre sans prendre en compte la vitesse de propagation (liée à la structure). Cela peut conduire à des choix non optimaux lorsque des branches peu profondes mais très ramifiées contaminent rapidement plus de sommets que des branches longues mais lentes.

4.2 Algorithme 2 (glouton) — Suppression du plus grand nombre de fils en priorité

Après avoir mis en évidence les limites de l'algorithme précédent, nous proposons un nouvel algorithme glouton qui s'appuie davantage sur la structure locale de l'arbre.

À chaque étape, cet algorithme glouton cible l'arc menant au nœud possédant le plus grand nombre de fils directs, dans l'idée que protéger les branches les plus ramifiées permettra d'isoler rapidement un nombre maximal de sommets. En cas d'égalité sur le nombre de fils, la décision se fait en fonction de la taille totale du sous-arbre concerné.

Le pseudocode est le suivant :

Algorithm 2: PlusGrandNombreDeFilsDabord(ls, res)

Entrée: ls : liste de sommets en cours
Sortie: res : liste d'arcs supprimées

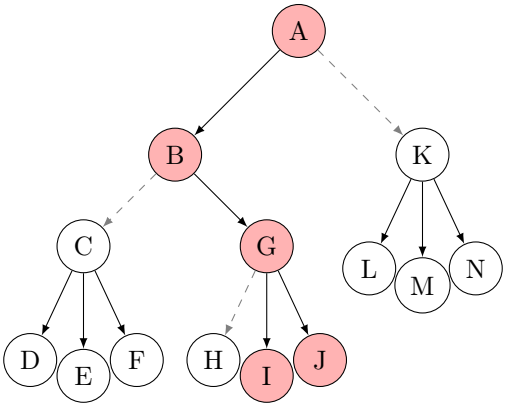
$maxGlobal \leftarrow 0$;
$aTraiter \leftarrow [\,]$;
$arcGlobale \leftarrow$ NIL;
Pour *chaque sommet* s *dans* ls **faire**
  $maxFils \leftarrow 0$;
  $maxSommet \leftarrow 0$;
  $arc \leftarrow$ NIL;
  Pour *chaque fils* fs *de* s **faire**
    ajouter fs à $aTraiter$;
    $nbFils \leftarrow |fs.fils|$;
    $nbSommet \leftarrow$ Calcul(fs);
    Si $nbFils > maxFils$ **alors**
      $maxFils \leftarrow nbFils$, $maxSommet \leftarrow nbSommet$, $arc \leftarrow (s, fs)$;
    sinon si $nbFils = maxFils$ ***et*** $nbSommet > maxSommet$ **alors**
      $maxSommet \leftarrow nbSommet$, $arc \leftarrow (s, fs)$;
  Si $maxSommet > maxGlobal$ **alors**
    $maxGlobal \leftarrow maxSommet$, $arcGlobale \leftarrow arc$;
Si $arcGlobale \neq NIL$ **alors**
  ajouter $arcGlobale$ à res;
  retirer la racine de $arcGlobale$ de $aTraiter$;
Retourner res, $maxGlobal + PlusGrandNombreDeFilsDabord(aTraiter, res)$;

Principe :
- À chaque itération, on regarde pour chacun des sommets en liste le nombre de fils directs.
- On choisit l'arc dont l'enfant a le plus de fils.
- Si deux candidats ont le même nombre de fils, on compare la taille totale de leur sous-arbre.
- On supprime cet arc, puis on réitère sur les sous-arbres non encore protégés.

Complexité : similaire à l'algorithme 1, la boucle principale est en $\mathcal{O}(n)$ et l'appel à `Calcul` peut être $\mathcal{O}(n)$, soit $\mathcal{O}(n^2)$ au pire.

4.2.1 Exemple montrant une limite de l'algorithme 2

Appliquons-le à un arbre :



Conséquence de l'exécution :

— *Étape 1 :* On coupe (A, K), ce qui protège immédiatement tout le sous-arbre droit : K, L, M et N sont sauvés.

— *Étape 2 :* Sur la branche gauche, B a des fils (C et G) qui on le même nombre de fils et les 2 sous-arbres ont la même taille : on coupe (B, C) avant que C ne soit infecté, isolant ainsi C, D, E et F.

— *Étape 2 :* Enfin, sur le sous-arbre de racine G, tous les fils ont le même nombre de successeur (0) et le même taille des sous-arbres sont les mêmes aussi (1) : on coupe (G, H) au hasard.

Au total :

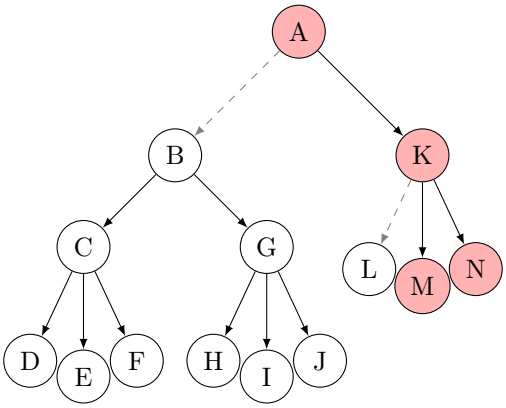
— **Sommets sauvés :** C, D, E, F, H (branche gauche) et K, L, M, N (branche droite) (*9 sommets*)

— **Sommets infectés :** A, B, G, I, J (*5 sommets*)

Choix alternatif optimal : Si l'algorithme avait supprimé l'arc $(A \to B)$ plutôt que $(A \to K)$:

— La branche gauche aurait été immédiatement protégée : 9 sommets sauvés $(B, C, D, E, F, G, H, I, J)$.

— La propagation dans la branche de droite, moins étendue, aurait permis de supprimer un autre arc après contamination, sauvant ainsi 10 sommets au lieu de 9.



Conclusion : On constate que l'algorithme 2 obtient de meilleurs résultats sur les arbres où l'algorithme 1 échoue à atteindre l'optimal (lorsque l'algo 2 trouve la solution optimale), mais qu'il donne en revanche des résultats moins bons sur d'autres arbres pour lesquels l'algorithme 1 était déjà optimal.

4.3 Algorithme 3 (heuristique) — Recuit simulé

Pour pallier l'instabilité des algorithmes gloutons (très performant sur certaines instances, médiocre sur d'autres), nous proposons d'exploiter une méthode méta-heuristique : le recuit simulé. L'idée est de parcourir l'espace des solutions (séquences d'arcs à supprimer) en acceptant parfois des mouvements «pire» pour échapper aux minima locaux, tout en diminuant progressivement la probabilité de ces choix non optimaux.

Principe général :

- On part d'une solution initiale s (par exemple fournie par un heuristique rapide).
- On maintient une «température» T qui décroît lentement (schedule de refroidissement).
- À chaque itération, on génère un voisin s' de la solution courante s (en modifiant un arc dans la séquence).
- On calcule l'amélioration $\Delta = \text{valeur}(s') - \text{valeur}(s)$, où valeur$(\cdot)$ est le nombre de sommets sauvés.
  - Si $\Delta > 0$, on accepte toujours s' (amélioration).
  - Sinon, on accepte s' avec la probabilité $\exp(\Delta/T)$, ce qui permet d'échapper aux minima locaux.
- On met à jour le meilleur solution trouvé $s_{\max}$ si nécessaire.
- On répète jusqu'à ce que T soit trop faible.

Algorithm 3: RecuitSimulé(s, T)

Entrée: s : une solution initiale, T : Température de départ
Sortie: res : Liste de d'arcs et le nombre de sommet sauvé

```
1  solutionMax ← s;
2  valeurMax ← Calcul(solutionMax);
3  valeur ← valeurMax;
4  Tant que T > 1 faire
5      max sans amélioration ← 60;
6      compteur ← 0;
7      Tant que compteur < max sans amélioration faire
8          compteur ← compteur + 1;
9          s' ← ChoisirVoisin(s);
10         valeur' ← Calcul(s');
11         Si valeur' > valeurMax alors
12             s ← s';
13             valeur ← valeur';
14             valeurMax ← valeur';
15             solutionMax ← s';
16             compteur ← 0;;
17         else
18             δ ← valeur' − valeur;
19             proba ← exp δ/T Si random(0, 1) < proba alors
20                 s ← s';
21                 valeur ← valeur'
22     T ← T ∗ 0.99;
23 Retourner solutionMax, valeurMax
```

4.3.1 Fonction auxiliaire : Choisir voisin

Pour explorer l'espace des solutions, on définit un voisinage où chaque solution voisine diffère de la solution courante s par la suppression d'un seul arc différent. Concrètement, on sélectionne au hasard une position dans la séquence d'arcs à supprimer et on la remplace par un autre arc du graphe non présent dans la solution actuelle. Dans le cas des arbres, on prend en compte que les arcs qui changent la valeur de la solution. Par exemple, si on a isolé la partie gauche de l'arbre, alors dans le changement d'arcs, on ne prend pas en compte les arcs dans la partie isolée car ils n'augmenteront pas la solution.

Algorithm 3: ChoisirVoisin(G, s)

Entrée: G : le graphe de départ, s : Un tableau d'arcs qui est une solution particulière
Sortie: *voisin* : Un tableau d'arcs qui est une nouvelle solution

```
Pour chaque arc a dans s faire
    exclure les arcs du sous arbre a[1]
Remplacer un élément de s par un arc aléatoire non exclu du graphe G;
Retourner s
```

Fonctionnement détaillé :

1. *Initialisation :*
$$s \leftarrow \text{solution de départ}, \quad s_{\max} \leftarrow s, \quad T \leftarrow T_0$$
2. *Boucle principale : tant que* $T > 1$
   (a) On fixe un compteur d'itérations sans amélioration (par exemple 60).
   (b) *Boucle interne :* tant que ce compteur n'est pas épuisé :
   — Générer un voisin s' par une petite modification de s (échange ou remplacement d'un arc à supprimer).
   — Calculer $v = \text{valeur}(s)$ et $v' = \text{valeur}(s')$.
   — Si $v' > v$, accepter toujours s' et remettre le compteur à zéro.
   — Sinon, accepter s' avec probabilité $\exp\big((v' - v)/T\big)$ et décrémenter le compteur.
   (c) Réduire la température, par exemple : $T \leftarrow 0.99\,T$.
3. *Retourner* la meilleure solution $s_{\max}$ et sa valeur.

Avantages et limites :
— Le recuit simulé peut échapper aux pièges des minima locaux et trouver une solution proche de l'optimal sur des instances variées.
— Le choix du schéma de refroidissement et de la structure de voisinage est crucial pour l'efficacité.
— Le temps de calcul peut être important si l'on autorise de nombreux essais à haute température.

4.4 Algorithme 4 (heuristique) — Tabu Search

Pour renforcer la stabilité de nos heuristiques gloutonnes, nous adoptons une méta-heuristique de type Tabu Search, spécialement adaptée aux **arbres** (absence de cycles) et tirant parti de leur structure hiérarchique.

L'idée centrale est de maintenir une « liste taboue » des dernières solutions explorées pour éviter les retours immédiats, tout en générant, à chaque itération, un voisinage restreint en deux étapes :

1. **Parcours niveau par niveau** depuis la racine infectée :
— Pour chaque hauteur h de l'arbre, on génère les voisins en remplaçant la coupure actuelle à ce niveau par chaque arc possible du même niveau.
— On calcule le score (nombre de sommets sauvés) de chacun, et l'on ne conserve que ceux qui ne figurent pas dans la liste Tabu (sauf si leur score dépasse le meilleur connu).

2. **Sélection locale** :
— Pour chaque hauteur h, on retient **uniquement** le voisin non tabou le plus améliorant.
— On met à jour la chaîne courante en y substituant ce meilleur mouvement à chaque niveau.

Cette approche « par niveau » présente plusieurs **avantages** :

— **Voisinage réduit** : au lieu d'explorer toutes les substitutions possibles, on ne garde qu'un candidat par profondeur, ce qui accélère considérablement la recherche.

— **Exploitation de l'arborescence** : chaque niveau hiérarchique est traité indépendamment, privilégiant les coupures stratégiques en amont.

— **Évitement des cycles** : la liste Tabu garantit qu'on n'effectue pas de retours inutiles, tout en autorisant, si besoin, une solution taboue si elle dépasse le meilleur score actuel.

Algorithm 4: Tabu Search pour l'isolation de propagation

Entrée: r : sommet infecté
Sortie: suite d'arcs à supprimer

```
1  Chaine_Max ← initialiser_chaine_de_base();
2  Max ← noter(Chaine_Max);
3  Chaine ← Chaine_Max;
4  Note ← Max;
5  Tabu ← file_vide();
6  Enfiler(Tabu, Chaine);
7  Occurence_max ← 10;
8  Occurence ← 0;
9  Tant que Occurence < Occurence_max faire
10     j ← 0;
11     A_visiter ← [r];
12     Tant que taille(A_visiter) > 0 faire
13         n ← taille(A_visiter);
14         Pour de 1 à n faire
15             Sommet ← Defiler(A_visiter);
16             Pour chaque fils f de Sommet faire
17                 Prochaine_Chaine[j] ← arc(Sommet, f);
18                 Prochaine_Note ← noter(Prochaine_Chaine);
19                 Si Prochaine_Chaine ∉ Tabu ou Prochaine_Note > Max alors
20                     Si Prochaine_Note > Note alors
21                         Note ← Prochaine_Note;
22                         Chaine ← Prochaine_Chaine;
23                 Enfiler(A_visiter, f);
24             j ← j + 1;
25     Occurence ← Occurence + 1;
26     Si Note > Max alors
27         Max ← Note;
28         Chaine_Max ← Chaine;
29         Occurence ← 0;
30     Enfiler(Tabu, Chaine);
31     Si taille(Tabu) > 20 alors
32         Defiler(Tabu);
33 Retourner Chaine_Max;
```

Fonctionnement de l'algorithme :

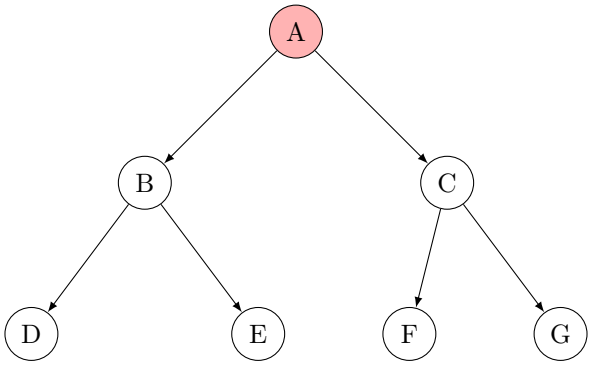
1. **Initialisation :**
   — Construire une chaîne initiale $Chaine\_Max$ et calculer son score Max.
   — Placer cette chaîne dans la liste Tabu (capacité fixe) et fixer le compteur de stagnation $Occurence \leftarrow 0$.
2. **Recherche itérative (tant que $Occurence < Occurence\_max$) :**
   (a) Pour chaque hauteur h de l'arbre :
      — Générer les voisins en remplaçant l'arc de la chaîne courante à la hauteur h.
      — Filtrer ceux non tabous (ou dont le score dépasse Max).
      — Sélectionner le meilleur voisin et l'intégrer à la chaîne.
   (b) Évaluer le nouveau score $Note$ de la chaîne.
3. **Mise à jour et mémoire :**
   — Si $Note > Max$, mettre à jour $Chaine\_Max \leftarrow Chaine$, $Max \leftarrow Note$ et réinitialiser $Occurence \leftarrow 0$.

— Sinon, incrémenter *Occurence*.
— Ajouter la chaîne courante à Tabu et, si besoin, retirer l'élément le plus ancien pour maintenir la taille.

4. **Arrêt :**
— Lorsque $Occurence = Occurence\_max$, retourner la meilleure chaîne $Chaine\_Max$.

4.4.1 Exemple d'exécution de l'Algorithme 4 sur un arbre

Prenons l'arbre suivant, où A est infecté à $t = 0$:



On note les profondeurs (hauteurs) des arcs :

$$\text{niveau 1} : \{A \to B,\ A \to C\}, \quad \text{niveau 2} : \{B \to D,\ B \to E,\ C \to F,\ C \to G\}.$$

1. Initialisation
— Chaîne initiale (arbitraire) :
$$Chaine = \big[(A \to B),\ (B \to D)\big].$$
— Score Max = nombre de sommets sauvés = 3 (on protège la sous-branche B, D, E).
— $Tabu = \{[(A \to B), (B \to D)]\}$, $Occurence = 0$.

2. Première itération (niveau 1 puis 2)

1. *Exploration du niveau 1* (remplacer $(A \to B)$) :
$$\text{Voisin 1} : [(A \to C), (B \to D)] \Rightarrow \text{sauve } \{C, F, G\} \cup \{D\} = 4,$$
(autres remplacements identiques non améliorants).

Le meilleur voisin non tabou est $[(A \to C), (B \to D)]$ avec score 4. On met à jour :
$$Chaine \leftarrow [(A \to C), (B \to D)], \quad Note \leftarrow 4.$$

2. *Exploration du niveau 2* (remplacer $(B \to D)$) :
$$[(A \to C), (B \to E)] \Rightarrow \text{sauve } \{C, F, G\} = 3,$$
$$[(A \to C), (C \to F)] \Rightarrow \{C, F, G\} = 3,$$
$$\dots$$

Aucun voisin n'améliore le score (toujours $3 < 4$), on garde donc $Chaine = [(A \to C), (B \to D)]$.

3. Mise à jour Tabu et stagnation
— Comme $Note = 4 > Max = 3$, on met à jour :
$$Chaine\_Max \leftarrow [(A \to C), (B \to D)], \quad Max \leftarrow 4, \quad Occurence \leftarrow 0.$$
— On ajoute la nouvelle chaîne à Tabu :
$$Tabu = \{[(A \to B), (B \to D)],\ [(A \to C), (B \to D)]\}.$$

Après quelques itérations sans amélioration ou jusqu'à $Occurence\_max$, l'algorithme retourne la meilleure chaîne $[(A \to C), (B \to D)]$ protégeant 4 sommets $\{C, F, G, D\}$.

4.5 Algorithme 5 (heuristique) - Aléatoire pour isolation de propagation

Plutôt que de chercher à « intelligemment » couper les arcs, nous nous interrogeons sur les performances d'une approche purement aléatoire. L'idée est de générer au hasard, pour chaque niveau de l'arbre, un arc à supprimer, puis d'évaluer le nombre de sommets sauvés. Après un nombre fixe d'itérations, on retient la meilleure chaîne rencontrée.

Algorithm 5: Aléatoire pour l'isolation de propagation

Entrée: r : sommet infecté
Sortie: tableau de suite d'arcs

```
1  Chaine_Max ← initialiser_chaine_de_base();
2  Max ← noter(Chaine_Max);
3  Chaine ← Chaine_Max;
4  Note ← Max;
5  Occurence_max ← 100;
6  Occurence ← 0;
7  Tant que Occurence < Occurence_max faire
8      j ← 0;
9      A_traiter ← [r];
10     Tant que taille(A_traiter) > 0 faire
11         A_traiter ← [r];
12         Pour un sommet s aléatoire dans A_traiter faire
13             Pour un fils f_alea aléatoire de s faire
14                 Chaine[j] ← arc(s, f_alea);
15         n ← taille(A_traiter);
16         Pour _ de 1 à n faire
17             Sommet ← Defiler(A_traiter);
18             Pour chaque fils f de Sommet tel que f ≠ f_alea faire
19                 A_traiter.Enfiler(f);
20     j ← j + 1;
21     Occurence ← Occurence + 1;
22     Note ← noter(Chaine);
23     Si Note > Max alors
24         Max ← Note;
25         Chaine_Max ← Chaine;
26 Retourner Chaine_Max;
```

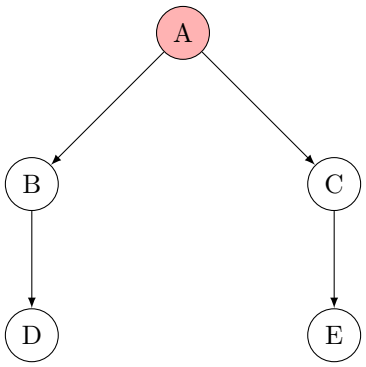
Explication du fonctionnement :

1. *Initialisation*
   — On part d'une chaîne de coupures de référence $Chaine\_Max$ et de son score Max.
   — On fixe le nombre d'itérations aléatoires ($Occurence\_max$).
2. *Génération aléatoire*
   — Tant que la file de niveaux n'est pas vide, on choisit un sommet s au hasard, puis un de ses fils f_{alea} aléatoirement.
   — On inscrit $(s \to f_{\text{alea}})$ dans la chaîne à la position correspondant à la profondeur actuelle.
   — On reconstruit la file pour le niveau suivant en excluant f_{alea}.
3. *Évaluation et mise à jour*
   — Après avoir rempli la chaîne pour tous les niveaux, on calcule son score $Note$.
   — Si $Note > Max$, on met à jour la meilleure chaîne $Chaine\_Max$ et Max.
4. *Arrêt*

— On répète ces tirages aléatoires $Occurence\_max$ fois, puis on retourne la meilleure solution rencontrée.

4.5.1 Exemple d'exécution

Considérons l'arbre ci-dessous, avec A infecté à $t = 0$ et un nombre d'itérations aléatoires fixé à 2 :



On repère deux niveaux d'arcs :

$$\text{niveau 1} : \{A \to B,\ A \to C\}, \quad \text{niveau 2} : \{B \to D,\ C \to E\}.$$

On lance l'algorithme avec `Occurence_max` $= 2$. Chaque essai construit au hasard une chaîne de deux coupures et calcule le nombre de sommets sauvés.

Essai	**Choix niveau 1**	**Choix niveau 2**	**Sommets sauvés**
1	$A \to C$	$C \to E$	$\{C, E\}$ (2)
2	$A \to B$	$B \to D$	$\{B, D\}$ (2)

Les deux essais donnent un score de 2 ; on peut retenir l'une ou l'autre des deux chaînes (par exemple $\big[(A \to C), (C \to E)\big]$).

Remarque : Sur cet arbre très petit, le hasard ne permet pas de dépasser le score 2, mais dès que l'arbre grandit, ce tirage stochastique offrira un large éventail de solutions à comparer.

4.6 Algorithme 6 (heuristique) — Randomisé ε-greedy

Spécialisé pour les **arbres** (absence de cycles) et non applicable directement aux **DAGs**, cet algorithme s'inspire des méthodes d'Ant Colony Optimization pour concilier exploration aléatoire et apprentissage adaptatif. Il réalise un grand nombre de simulations, et à chaque position j de la chaîne de coupures il alterne entre :

— *exploration* (choix aléatoire d'un arc avec probabilité ε),

— *exploitation* (sélection de l'arc dont la moyenne des scores passés $\frac{W_{j,a}}{N_{j,a}+1}$ est la plus élevée).

À l'issue de chaque simulation, le score obtenu est injecté dans les compteurs $W_{j,a}$ et $N_{j,a}$ pour guider progressivement les choix futurs, et la meilleure chaîne rencontrée est mémorisée.

À la fin de chaque simulation, le score obtenu est ajouté aux compteurs $W_{j,a}$ et $N_{j,a}$ pour guider progressivement les futurs choix, et la meilleure chaîne rencontrée est mémorisée.

Algorithm 6: Randomisé ε-greedy

Entrée: r : sommet infecté
Sortie: $\text{Chaine}_{\max}$: suite d'arcs à supprimer

Pour $j = 0, \ldots, j_{\max}$ **faire**
  foreach $a \in E$ **do**
    $N_{j,a} \leftarrow 0$;
    $W_{j,a} \leftarrow 0$;
$Max \leftarrow -\infty$;
$\text{Chaine}_{\max} \leftarrow [\,]$;
Pour $it = 1, \ldots,$ `max_iter` **faire**
  $Chaine \leftarrow [\,]$;
  $A_{\text{traiter}} \leftarrow [\,r\,]$;
  Pour $j = 0, \ldots,$ `profondeur_max` $- 1$ **faire**
    $Actions \leftarrow \{(x, f) \mid x \in A_{\text{traiter}},\ f \in \text{fils}(x)\}$;
    Si $Actions = \emptyset$ **alors**
      Break
    Si `rand()` $< \varepsilon$ **alors**
      $(x_s, f_s) \leftarrow$ choix_uniforme($Actions$);
    sinon
      $(x_s, f_s) \leftarrow \arg\max_{(x,f) \in Actions} \frac{W_{j,(x,f)}}{N_{j,(x,f)} + 1}$;
    $Chaine[j] \leftarrow (x_s, f_s)$;
    $A_{\text{temp}} \leftarrow [\,]$;
    foreach $y \in A_{\text{traiter}}$ **do**
      foreach $f \in \text{fils}(y)$ ***si*** $(y, f) \neq (x_s, f_s)$ **do**
        $A_{\text{temp}} \leftarrow A_{\text{temp}} \cup \{f\}$;
    $A_{\text{traiter}} \leftarrow A_{\text{temp}}$;
  $score \leftarrow \text{noter}(Chaine)$;
  Pour $k = 0, \ldots, |Chaine| - 1$ **faire**
    $a_k \leftarrow Chaine[k]$;
    $N_{k,a_k} \leftarrow N_{k,a_k} + 1$;
    $W_{k,a_k} \leftarrow W_{k,a_k} + score$;
  Si $score > Max$ **alors**
    $Max \leftarrow score$;
    $\text{Chaine}_{\max} \leftarrow Chaine$;
return $\text{Chaine}_{\max}$;

Principe de fonctionnement :

1. *Initialisation des statistiques* Pour chaque profondeur j et chaque arc $a \in E$, on met $N_{j,a} = W_{j,a} = 0$.
2. *Boucle de simulations* (`max_iter` itérations) :
   — On construit une chaîne *Chaine* position par position :
      — On rassemble toutes les actions possibles (x, f) depuis la file de nœuds à traiter.
      — Avec probabilité ε, on choisit (x, f) uniformément au hasard (*exploration*), sinon on prend $\arg\max_{(x,f)} \frac{W_{j,(x,f)}}{N_{j,(x,f)}+1}$ (*exploitation*).
      — On met à jour la file pour la profondeur suivante en excluant le fils choisi.
   — On évalue la chaîne (nombre de sommets sauvés) et on met à jour :

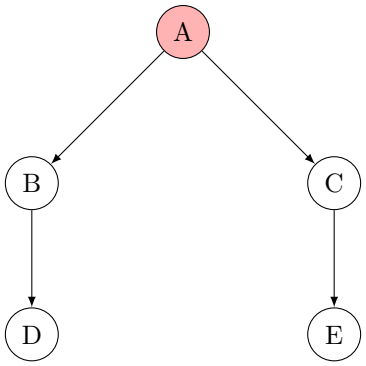
$$N_{j,a} \leftarrow N_{j,a} + 1, \quad W_{j,a} \leftarrow W_{j,a} + \text{score}$$

   pour chaque arc a utilisé en position j.
   — Si le score dépasse le meilleur Max, on remplace $\text{Chaine}_{\max}$ et Max.
3. *Résultat* Après `max_iter` simulations, on retourne la chaîne $\text{Chaine}_{\max}$ ayant obtenu le meilleur score.

Avantages et limites :
— ε-greedy permet d'exploiter progressivement les arcs les plus prometteur tout en conservant une part d'exploration pour sortir des minima locaux.
— Le paramètre ε et le nombre de simulations `max_iter` contrôlent le compromis exploration/exploitation.
— Coût $\mathcal{O}(\texttt{max\_iter} \times n \times d)$, avec d la moyenne des degrés (nombre d'actions possibles par niveau).

4.6.1 Exemple d'exécution de l'Algorithme 6

Considérons l'arbre suivant (A infecté à $t = 0$) :



On a deux niveaux :
— niveau 0 : $\{A \to B,\ A \to C\}$
— niveau 1 : $\{B \to D,\ C \to E\}$
Soit $\varepsilon = 0.5$ et `max_iter` $= 3$. On initialise tous $N_{j,a} = W_{j,a} = 0$.

Itération 1 (exploration forcée car tout vaut 0) :
— Niveau 0 : choix uniforme, disons $(A \to C)$.
— Niveau 1 : choix uniforme, disons $(C \to E)$.
— Chaîne : $[(A \to C), (C \to E)]$, score $= 2$.
— Mise à jour :

$$N_{0,(A\to C)} = 1,\ W_{0,(A\to C)} = 2, \quad N_{1,(C\to E)} = 1,\ W_{1,(C\to E)} = 2.$$

Itération 2 (exploitation partielle) :

— Niveau 0 :

$$\frac{W_{0,(A\to B)}}{N_{0,(A\to B)}+1}=0, \quad \frac{W_{0,(A\to C)}}{N_{0,(A\to C)}+1}=1.$$

Avec prob. $1-\varepsilon=0.5$ on choisit $(A\to C)$ (meilleur).

— Niveau 1 :

$$\frac{W_{1,(B\to D)}}{N_{1,(B\to D)}+1}=0, \quad \frac{W_{1,(C\to E)}}{N_{1,(C\to E)}+1}=1.$$

De même, on prend $(C\to E)$.

— Chaîne : $[(A\to C),(C\to E)]$, score $=2$.

— Mise à jour :

$$N_{0,(A\to C)}=2,\ W_{0,(A\to C)}=4, \quad N_{1,(C\to E)}=2,\ W_{1,(C\to E)}=4.$$

Itération 3 (exploitation forte) :

— Niveau 0 : $\frac{W_{0,(A\to C)}}{N_{0,(A\to C)}+1}=4/3\approx 1.33$ contre $0/1=0$, donc $(A\to C)$.

— Niveau 1 : $\frac{W_{1,(C\to E)}}{N_{1,(C\to E)}+1}=4/3\approx 1.33$ contre $0/1=0$, donc $(C\to E)$.

— Chaîne : $[(A\to C),(C\to E)]$, score $=2$.

— Mise à jour (mais pas de nouveau best) : $N_{0,(A\to C)}=3,\ W_{0,(A\to C)}=6,\ N_{1,(C\to E)}=3,\ W_{1,(C\to E)}=6$.

Au terme des 3 simulations, la meilleure chaîne retenue est

$$\big[(A\to C),(C\to E)\big]$$

avec un score de 2. Cet exemple illustre comment l'algorithme ε-greedy privilégie progressivement les arcs qui ont donné les meilleurs retours tout en gardant une part d'exploration.

5 Cas général : propagation dans un graphe orienté acyclique (DAG)

Dans le cas des arbres, on supposait qu'il suffisait de supprimer exactement un arc par niveau (hauteur) pour couper toute voie unique menant à chaque feuille, puisque chaque sommet n'est accessible que par un seul chemin. En revanche, dans un DAG, plusieurs chemins disjoints peuvent relier un même couple de sommets A et B.

- La suppression d'un seul arc à une hauteur donnée n'est plus garante d'isoler entièrement le sous-graphe descendant, car d'autres routes parallèles peuvent exister.
- On ne connaît plus *a priori* le nombre d'arcs à retirer : il dépend de la multiplicité des chemins et de leur chevauchement.
- Il faut donc rechercher une *séquence ordonnée* d'arcs à supprimer, dont la longueur n'est plus fixée par la hauteur, mais déterminée par la topologie du DAG.

Nous devons donc passer d'une stratégie « un arc par niveau » à une formalisation qui renvoie une suite (dynamique) d'arcs à supprimer, choisies en fonction des chemins multiples et de leur impact combiné sur la propagation.

5.1 Algorithme 7 (glouton) - Plus grand sous DAG

À la manière des arbres, afin d'obtenir une solution rapide, on va regarder le nombre de sommets des DAG voisins des sommets infectés et couper l'arc qui mène vers la plus grande valeur.

Algorithm 7: PlusGrandSousDAGDAbord(ls, res)

Entrée: ls : liste de sommets
Sortie: res : liste d'arcs maximales

```
1  maxGlobal ← 0;
2  aTraiter ← [ ];
3  arcGlobale ← NIL;
4  Pour chaque sommet s dans ls faire
5      max ← 0;
6      arc ← NIL;
7      Pour chaque fils fs de s faire
8          ajouter fs à aTraiter;
9          nb ← Calcul(fs);
10         Si nb > max alors
11             max ← nb;
12             arc ← (s, fs);
13     Si max > maxGlobal alors
14         maxGlobal ← max;
15         arcGlobale ← arc;
16 Si arcGlobale ≠ NIL alors
17     ajouter arcGlobale à res;
18     retirer le sommet de l'arcGlobale dans aTraiter;
19 Retourner res, maxGlobal + PlusGrandSousDAGDAbord(aTraiter, res);
```

Fonction auxiliaire : Calcul

Cette fonction permet de calculer la taille du sous-DAG en un sommet donné. Elle est utilisée par l'algorithme précédent pour choisir quel arc supprimer.

Algorithme permettant de calculer le nombre de sommets dans un sous DAG

Algorithm 7: Calcul(s : sommet)

```
[H] Entrée: s : un sommet
    Sortie: Le nombre de sommets dans le sous-DAG en s
Si s.fils = ∅ alors
    Retourner 1;
nombre ← 1;
Pour chaque fils fs de s faire
    nombre ← nombre + Calcul(fs);
```

5.2 Algorithme 8 (heuristique) — Recuit simulé

Pour le premier algorithme, nous avons simplement réutilisé l'heuristique du recuit simulé : en adaptant la fonction de génération de voisin pour tenir compte de la structure d'un DAG, celle-ci devient directement applicable à ces graphes acycliques.

Principe général :

- On part d'une solution initiale s (par exemple fournie par un heuristique rapide).
- On maintient une «température» T qui décroît lentement (schedule de refroidissement).
- À chaque itération, on génère un voisin s' de la solution courante s (en modifiant un arc dans la séquence).
- On calcule l'amélioration $\Delta = \text{valeur}(s') - \text{valeur}(s)$, où valeur$(\cdot)$ est le nombre de sommets sauvés.
  - Si $\Delta > 0$, on accepte toujours s' (amélioration).
  - Sinon, on accepte s' avec la probabilité $\exp(\Delta/T)$, ce qui permet d'échapper aux minima locaux.
- On met à jour le meilleur solution trouvé $s_{\max}$ si nécessaire.
- On répète jusqu'à ce que T soit trop faible ou qu'aucune amélioration n'ait lieu pendant un certain nombre d'essais.

Algorithm 8: RecuitSimulé(s, T)

Entrée: s : une solution initiale, T : Température de départ
Sortie: res : Liste de d'arcs et le nombre de sommet sauvé

```
1  solutionMax ← s;
2  valeurMax ← Calcul(solutionMax);
3  valeur ← valeurMax;
4  Tant que T > 1 faire
5      max sans amélioration ← 60;
6      compteur ← 0;
7      Tant que compteur < max sans amélioration faire
8          compteur ← compteur + 1;
9          s' ← ChoisirVoisin(s);
10         valeur' ← Calcul(s');
11         Si valeur' > valeurMax alors
12             s ← s';
13             valeur ← valeur';
14             valeurMax ← valeur';
15             solutionMax ← s';
16             compteur ← 0;;
17         else
18             δ ← valeur' − valeur;
19             proba ← exp δ/T Si random(0, 1) < proba alors
20                 s ← s';
21                 valeur ← valeur'
22     T ← T ∗ 0.99;
23 Retourner solutionMax, valeurMax
```

5.2.1 Fonction auxiliaire : Choisir voisin

Pour explorer l'espace des solutions, on définit un voisinage où chaque solution voisine diffère de la solution courante s par la suppression d'un seul arc différente. Concrètement, on sélectionne au hasard une position dans la séquence d'arcs à supprimer et on le remplace par un autre arc du graphe non présent dans la solution actuelle.

On dispose aussi d'un autre voisinage qui est une rotation de deux arcs choisis aléatoirement dans la solution courante concaténer aux arcs du graphe.

Algorithm 8: ChoisirVoisin(G, s)

Entrée: G : le graphe de départ, s : Un tableau d'arcs qui est une solution particulière
Sortie: $voisin$: Un tableau d'arcs qui est une nouvelle solution

```
1  Remplacer un élément de s par un arc aléatoire du graphe G;
2  Retourner s
```

5.3 Algorithme 9 — Tabu Search étendu aux DAGs

Pour rendre la Tabu Search utilisable sur des **DAG** (et non plus seulement sur des arbres), nous avons adapté la génération de voisinage et le critère d'arrêt. Plutôt que de parcourir les niveaux d'un arbre, on travaille directement sur la « chaîne » d'arcs de longueur *taille_Chaine* et on définit un voisin par simple **échange** de deux positions dans cette chaîne. La liste taboue empêche les retours immédiats, et le pivot j tourne à chaque itération pour balayer toutes les positions.

Algorithm 9: Tabu Search pour l'isolation de propagation

Entrée: r : sommet infecté
Sortie: suite d'arcs à supprimer

```
1  Chaine_Max ← initialiser_chaine_de_base();
2  taille_Chaine ← taille(Chaine_Max);
3  Max ← noter(Chaine_Max);
4  Chaine ← Chaine_Max;
5  Note ← Max;
6  Tabu ← file_vide();
7  Enfiler(Tabu, Chaine);
8  Occurence_max ← taille_Chaine;
9  Occurence ← 0;
10 j ← 1;
11 Tant que Occurence < Occurence_max faire
12     Pour i de 1 à taille_Chaine faire
13         tmp_Chaine ← Chaine;
14         Si i ≠ j alors
15             tmp_Chaine[i] ← Chaine[j];
16             tmp_Chaine[j] ← Chaine[i];
17             tmp_Note ← noter(tmp_Chaine;
18             Si tmp_Chaine ∉ Tabu ou tmp_Note > Max alors
19                 Si tmp_Note > Note alors
20                     Note ← tmp_Note;
21                     Chaine ← tmp_Chaine;
22     Occurence ← Occurence + 1;
23     Si Note > Max alors
24         Max ← Note;
25         Chaine_Max ← Chaine;
26         Occurence ← 0;
27     Enfiler(Tabu, Chaine);
28     Si taille(Tabu) > 20 alors
29         Defiler(Tabu);
30     j ← (j mod taille_Chaine) + 1
31 Retourner Chaine_Max;
```

Fonctionnement détaillé :

1. **Initialisation**
   - Chaine_Max ← chaîne de base ; taille_Chaine ← |Chaine_Max| ; Max ← noter(Chaine_Max).
   - Copier dans Chaine et Note.
   - Tabu ← file vide, on y enfile Chaine.
   - Occurence_max ← taille_Chaine, Occurence ← 0, $j \leftarrow 1$.
2. **Boucle principale (pivot tourant)**

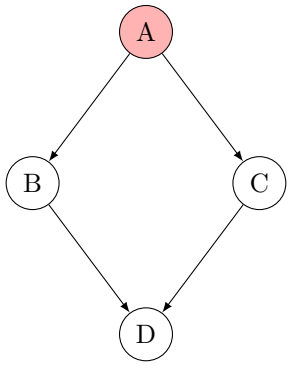
Tant que Occurence < Occurence_max :

— Pour $i = 1, \ldots,$ taille_Chaine $(i \neq j)$:
  (a) Échanger les éléments i et j de Chaine pour obtenir tmp_Chaine.
  (b) Calculer tmp_Note = noter(tmp_Chaine).
  (c) Si tmp_Chaine $\notin$ Tabu ou tmp_Note > Max, et tmp_Note > Note, alors Chaine $\leftarrow$ tmp_Chaine, Note $\leftarrow$ tmp_Note.
— Occurence $\leftarrow$ Occurence $+ 1$.
— Si Note > Max : Max $\leftarrow$ Note, Chaine_Max $\leftarrow$ Chaine, Occurence $\leftarrow 0$.
— Enfiler Chaine dans Tabu ; si |Tabu| > 20, défiler l'ancien.
— Mettre à jour le pivot $j \leftarrow (j \bmod$ taille_Chaine$) + 1$.

3. **Résultat** Chaine_Max est la meilleure suite d'arcs supprimés pour isoler la propagation dans le DAG.

5.3.1 Exemple d'exécution de l'Algorithme 9 — Tabu Search étendu aux DAGs

Considérons ce **DAG** où deux chemins convergent vers D :



On fixe la chaîne et on initialise :

Chaine = $[(A \to B), (B \to D), (A \to C), (C \to D]$, Chaine_Max = Chaine, Max = Note = noter(Chaine) = 0,

Tabu = {Chaine}, Occurence_max = 2, Occurence = 0, $j = 1$, taille_Chaine = 4.

Itération 1 $(j = 1)$:
— $i = 1$: on passe (car $i = j$).
— $i = 2$: on échange positions 1 et 2 :

$$\text{tmp\_Chaine} = [(B \to D), (A \to B), (A \to C), (C \to D].$$

Comme noter(tmp_Chaine) $= 0 \not> 0$, on ne change pas Chaine.
— $i = 3$: on échange positions 1 et 3 :

$$\text{tmp\_Chaine} = [(A \to C), (B \to D), (A \to B), (C \to D].$$

Comme noter(tmp_Chaine) $= 2 > 0$, on change Chaine pour [(A $\to$ C), (B $\to$ D), (A $\to$ B), (C $\to$ D].
— $i = 4$: on échange positions 1 et 4 :

$$\text{tmp\_Chaine} = [(C \to D), (B \to D), (A \to C), (A \to B)].$$

Comme noter(tmp_Chaine) $= 1 \not> 2$, on ne change pas Chaine.
— Fin : Occurence $\leftarrow 1$, on enfile Chaine dans Tabu, max_Chaine $\leftarrow$ Chaine car noter(Chaine) > Max $j \leftarrow (1 \bmod 2) + 1 = 2$.

Itération 2 $(j = 2)$:
— $i = 1$: on échange à nouveau, on retombe sur tmp_Chaine = $[(B \to D), (A \to C), (A \to B), (C \to D]$, pas d'amélioration.

— $i = 2$: on passe.
— ...

Itération 3 ($j = 3$) : ...

— ...
— Fin : Occurence $\leftarrow 2 =$ Occurence_max, on arrête.

La meilleure solution gardée est

$$\text{Chaine\_Max} = [(A \rightarrow C),\ (B \rightarrow D),\ (A \rightarrow B),\ (C \rightarrow D)]$$

qui, sur ce DAG, bloque la propagation vers C puis D et protège donc 2 sommets.

5.4 Algorithme 10 — Randomisé ε-greedy

Pour rendre cette méthode compatible avec les DAG, nous étendons les statistiques à **toutes** les positions de la chaîne et à **tous** les arcs possibles. Autrement dit, pour chaque indice j de la séquence et pour chaque arc $a \in E$, on conserve deux compteurs $N_{j,a}$ et $W_{j,a}$, de façon à estimer la qualité moyenne de chaque coupure en fonction de sa position.

À la fin de chaque simulation, le score obtenu est ajouté aux compteurs $W_{j,a}$ et $N_{j,a}$ pour guider progressivement les futurs choix, et la meilleure chaîne rencontrée est mémorisée.

Algorithm 10: Randomisé ε-greedy pour DAG

Entrée: r : sommet infecté
Sortie: $\text{Chaine}_{\max}$: suite d'arcs à supprimer

$Stats = []$ **Pour** $j = 0, \dots, j_{\max}$ **faire**
  Ajouter($structureTrie$) dans $Stats$;
  foreach $a \in E$ **do**
    $Stats[j][N_{j,a}] \leftarrow 0$;
    $Stats[j][W_{j,a}] \leftarrow 0$;
$\text{Chaine}_{\max} \leftarrow$ initialisation_chaine();
$Max \leftarrow$ noter($\text{Chaine}_{\max}$);
Pour $it = 1, \dots,$ `max_iter` **faire**
  $Chaine \leftarrow [\,]$;
  Pour $j = 0, \dots, j_{\max}$ **faire**
    Si `rand()`$<\varepsilon$ **alors**
      $(x_s, f_s) \leftarrow$ choix_uniforme($\text{Chaine}_{\max}$);
    sinon
      $(x_s, f_s) \leftarrow \arg\max_{(x,f)\in \text{Chaine}_{\max}} \dfrac{Stats[j][W_{j,(x,f)}]}{Stats[j][N_{j,(x,f)}]+1}$;
      $top \leftarrow 1$;
      Tant que $(x_s, f_s) \in Chaine$ **faire**
        $top \leftarrow top + 1$;
        $(x_s, f_s) \leftarrow \arg\max_{Stats(top)} \dfrac{Stats[j][W_{j,(x,f)}]}{Stats[j][N_{j,(x,f)}]+1}$;
    $Chaine[j] \leftarrow (x_s, f_s)$;
  $score \leftarrow$ noter($Chaine$);
  Pour $k = 0, \dots, |Chaine| - 1$ **faire**
    $a_k \leftarrow Chaine[k]$;
    $Stats[k][N_{k,a_k}] \leftarrow N_{k,a_k} + 1$;
    $Stats[k][W_{k,a_k}] \leftarrow W_{k,a_k} + score$;
  Si $score > Max$ **alors**
    $Max \leftarrow score$;
    $\text{Chaine}_{\max} \leftarrow Chaine$;
return $\text{Chaine}_{\max}$;

Étapes détaillées :

1. **Initialisation** Pour chaque position $j = 0, \dots, j_{\max}$ (typiquement $|E| - 1$) et chaque arc $a \in E$, on pose

$$N_{j,a} \leftarrow 0, \quad W_{j,a} \leftarrow 0.$$

On initialise également $\text{Chaine}_{\max}$ par une chaîne de référence et Max = noter($\text{Chaine}_{\max}$).

2. **Boucle de simulations** ($it = 1$ à `max_iter`)

— On vide la chaîne courante *Chaine* et on initialise l'ensemble Rem $= E$ des arcs disponibles.
— Pour chaque position $j = 0, 1, \ldots$ jusqu'à Rem vide :
  — 1. Avec probabilité ε, on tire a_s uniformément dans Rem (*exploration*).
  — 2. Sinon, on calcule pour chaque $a \in$ Rem le ratio $r_{j,a} = \dfrac{W_{j,a}}{N_{j,a}+1}$, puis on prend l'arc a_s qui maximise $r_{j,a}$ (*exploitation*).
  — 3. Si cet arc a déjà été placé dans *Chaine* (pour éviter les collisions), on passe au second meilleur, puis au troisième, etc., jusqu'à trouver un arc non encore choisi.
  — 4. On ajoute a_s à *Chaine*$[j]$ et on retire a_s de Rem.

3. **Évaluation**
— On calcule le score de la chaîne finale : $score =$ noter(*Chaine*).
— Pour chaque position k et l'arc $a_k =$ *Chaine*$[k]$, on met à jour

$$N_{k,a_k} \mathrel{+}= 1, \quad W_{k,a_k} \mathrel{+}= score.$$

— Si $score >$ Max, on remplace Max $\leftarrow score$ et $\text{Chaine}_{\max} \leftarrow$ *Chaine*.

4. **Retour** Après `max_iter` itérations, $\text{Chaine}_{\max}$ est la permutation d'arcs qui a obtenu le meilleur score.

Pourquoi ça marche ?

— Les compteurs $(N_{j,a}, W_{j,a})$ estiment la qualité moyenne de chaque arc a quand il est utilisé à la position j.
— L'exploration aléatoire (ε) permet de découvrir de nouvelles combinaisons,
— L'exploitation favorise peu à peu les arcs qui ont donné de bons résultats,
— L'obligation de permuter sans répétition (on retire a_s de Rem) garantit que la chaîne couvre bien tous les arcs du DAG.
— C'est une méthode heuristique, sans garantie d'optimalité, mais qui apprend progressivement quelles positions bénéficient le plus de quels arcs.

6 Résultats obtenus

6.1 Résultats moyens sur 100 simulations

Nous comparons cinq méthodes :
— Aléatoire pur
— ε-greedy
— Recuit simulé
— Tabu rapide (pivot cyclique + K=5)
— Tabu complet (liste taboue taille 20)

Pour chacun des trois DAGs, nous affichons la courbe du meilleur score cumulé (record global) en fonction du nombre d'itérations.

6.1.1 DAG 1

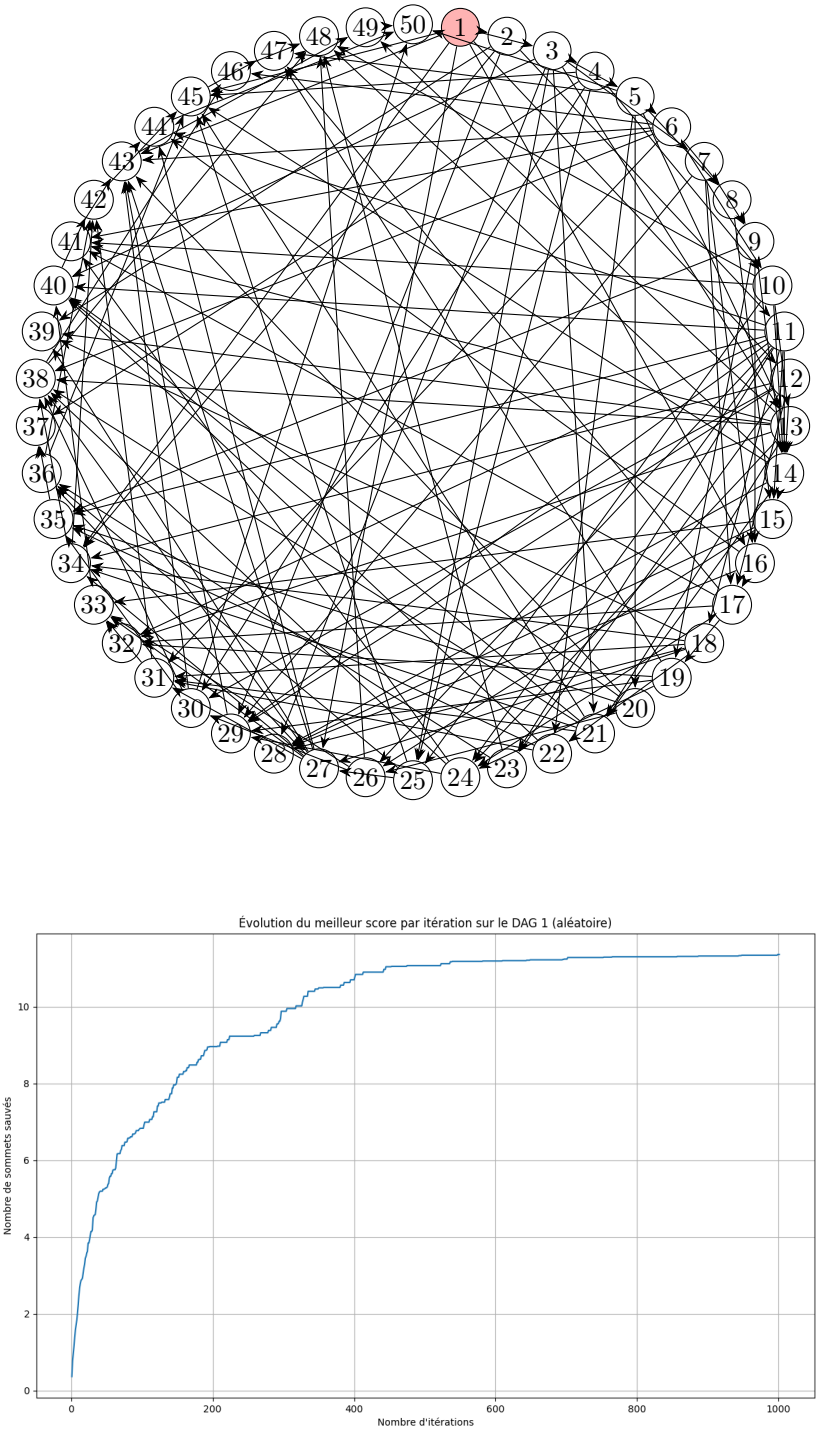


FIGURE 1 – Aléatoire pure

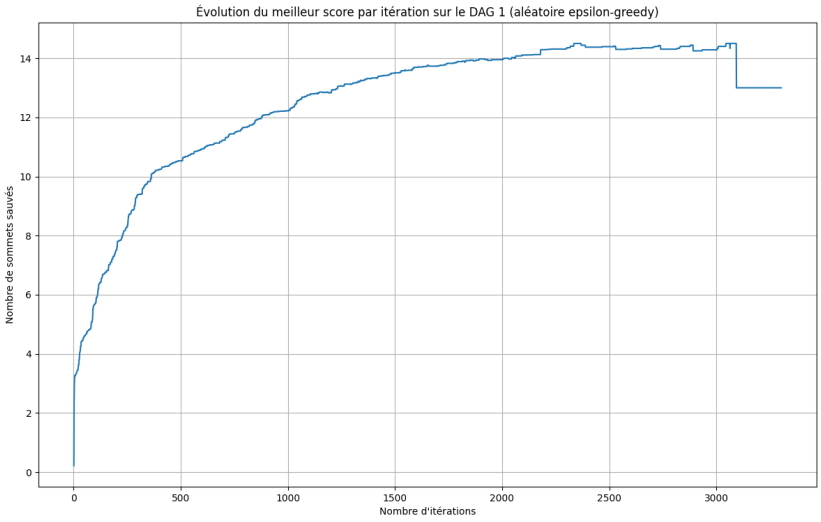


FIGURE 2 – ε-greedy

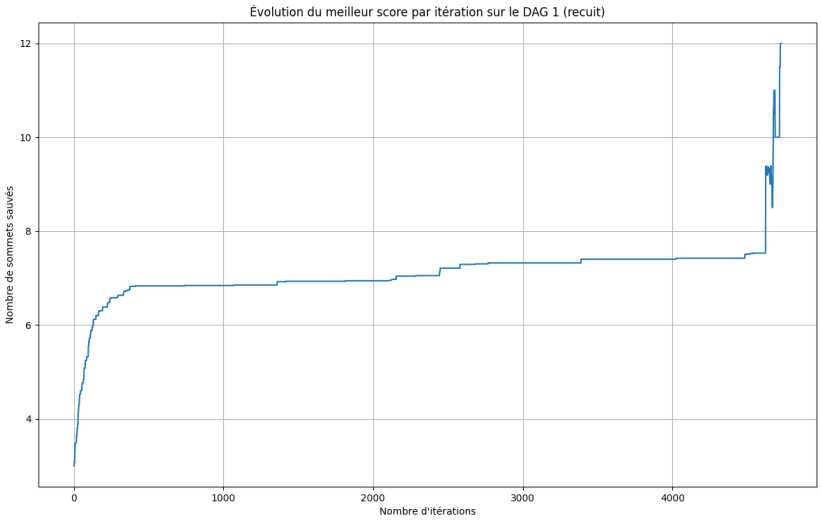


FIGURE 3 – Recuit simulé

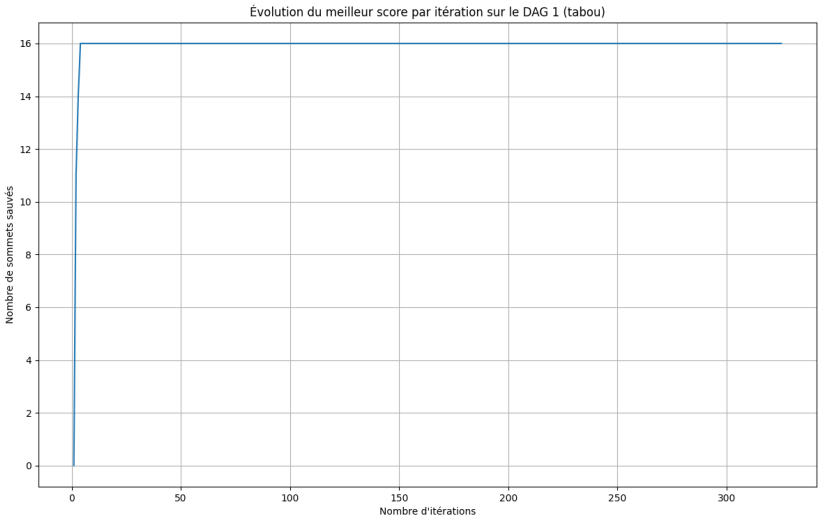
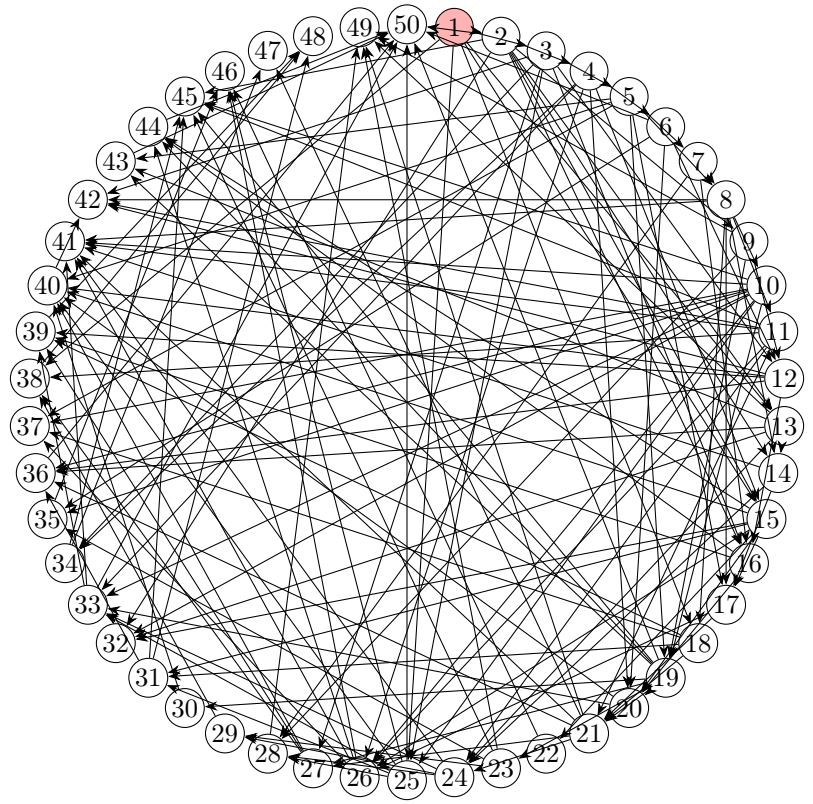


FIGURE 4 – Tabou

TABLE 1 – Comparaison des scores moyens

Méthode	Score moyen atteint
Aléatoire pur	$\simeq$11
ε-greedy	$\simeq$13
Recuit simulé	12
Tabou	16

6.1.2 DAG 2



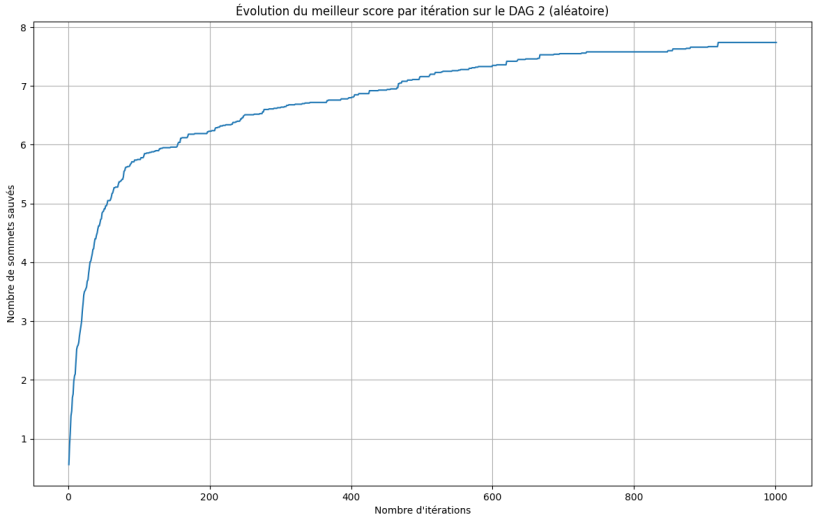


FIGURE 5 – Aléatoire pure

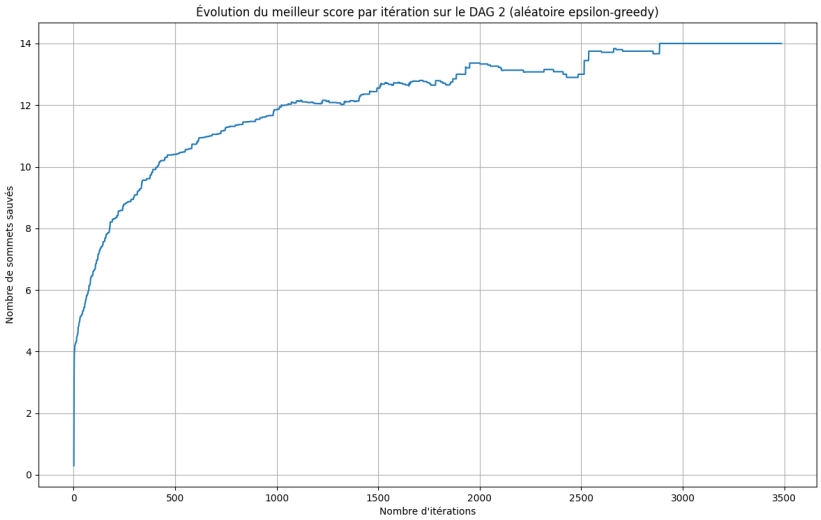


FIGURE 6 – ε-greedy

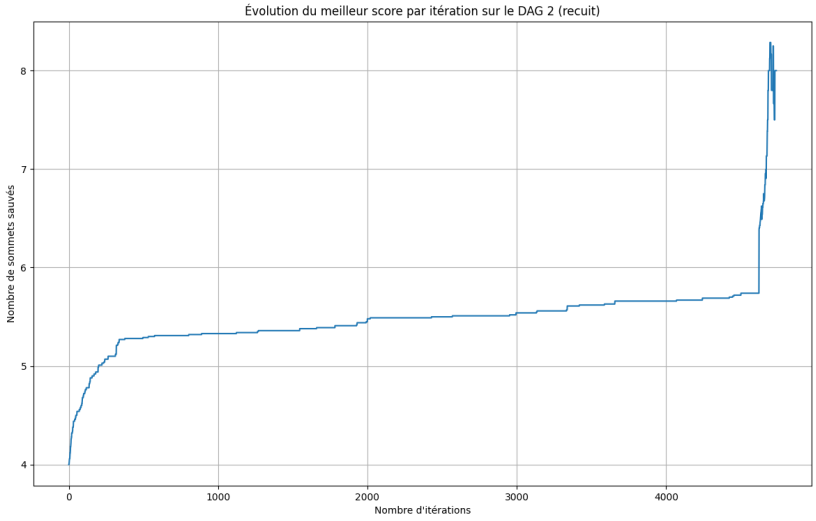


FIGURE 7 – Recuit simulé

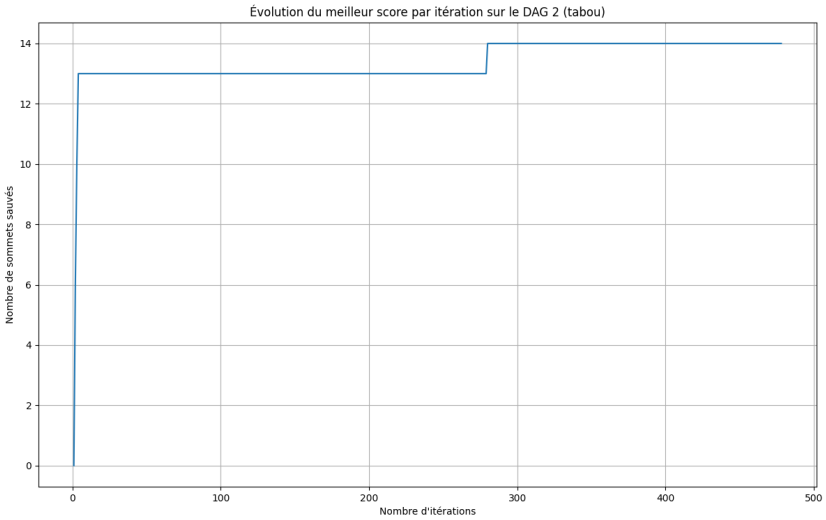
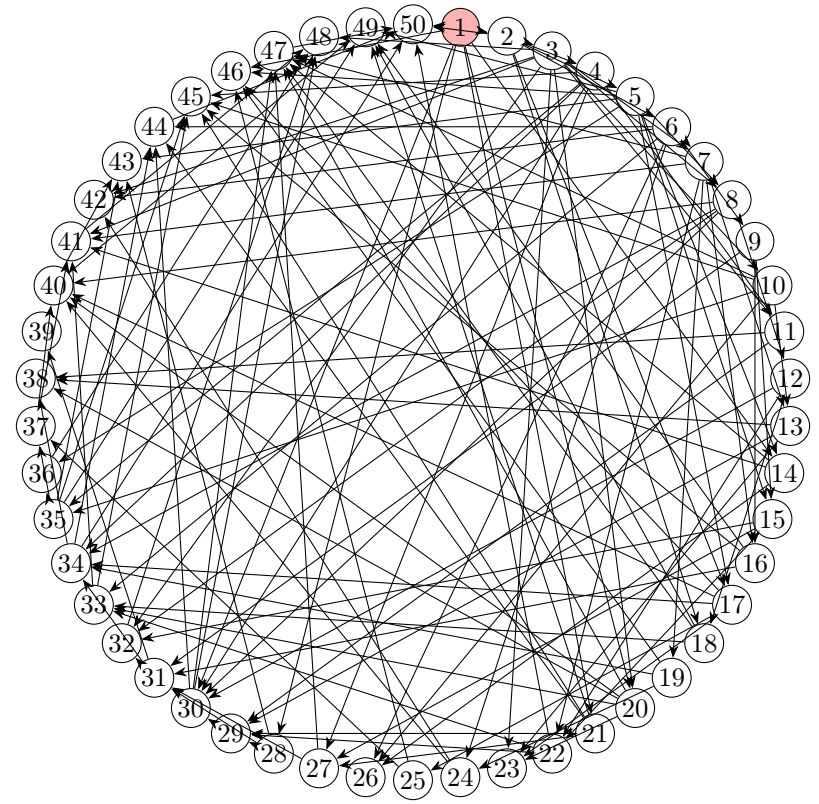


FIGURE 8 – Tabou

TABLE 2 – Comparaison des scores moyens

Méthode	Score moyen atteint
Aléatoire pur	$\simeq 8$
ε-greedy	$\simeq 14$
Recuit simulé	$\simeq 8$
Tabou	14

6.1.3 DAG 3



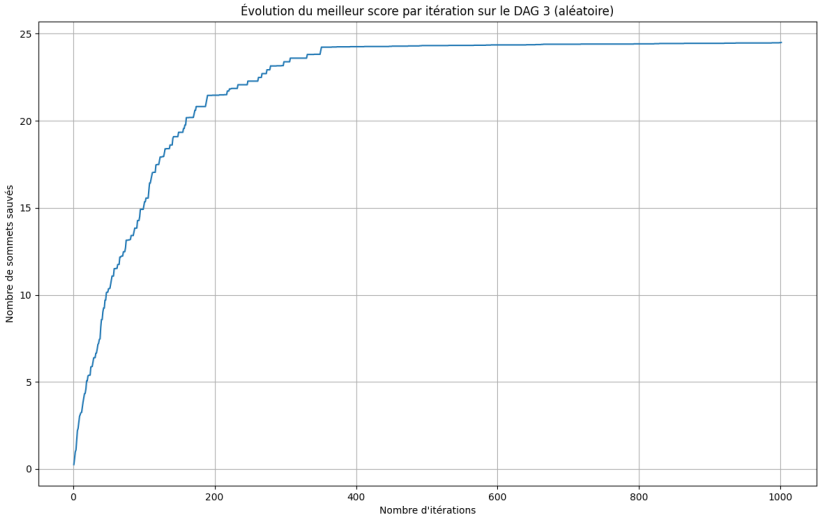


FIGURE 9 – Aléatoire pure

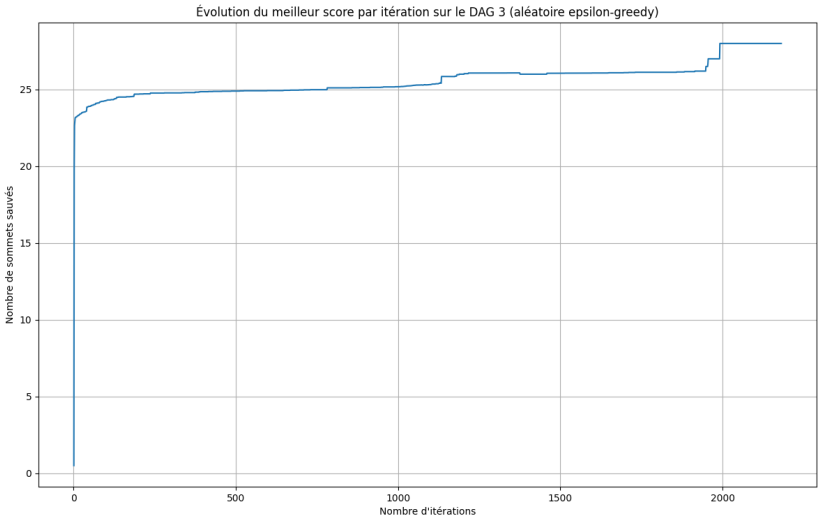


FIGURE 10 – ε-greedy

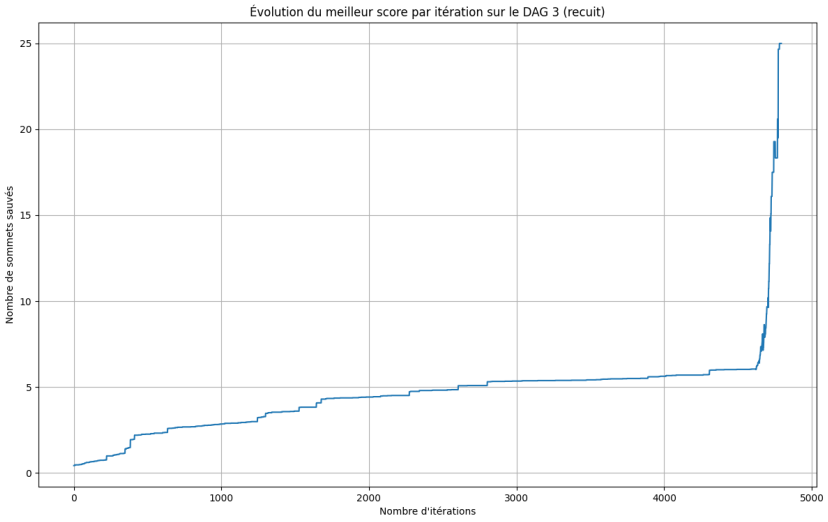


FIGURE 11 – Recuit simulé

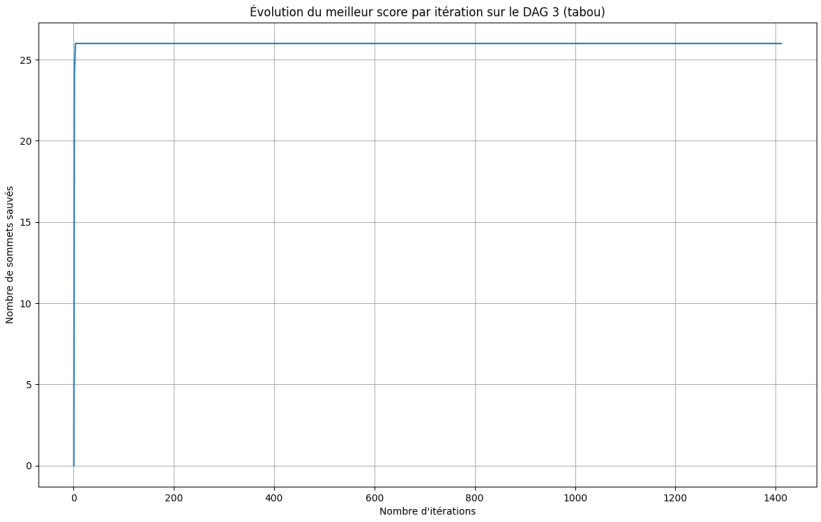


FIGURE 12 – Tabou

TABLE 3 – Comparaison des scores moyens

Méthode	Score moyen atteint
Aléatoire pur	$\simeq$24
ε-greedy	$\simeq$25
Recuit simulé	25
Tabou	25

6.2 Conclusion des expérimentations

Les performances moyennes finales en 100 simulations pour chaque DAG sont les suivantes :

TABLE 4 – Comparaison des scores moyens pour chaque DAG

Méthode	DAG 1	DAG 2	DAG 3
Aléatoire pur	$\simeq$11	$\simeq$8	$\simeq$24
ε-greedy	$\simeq$13	$\simeq$14	$\simeq$25
Recuit simulé	12	$\simeq$8	25
Tabou	16	14	25

1. Aléatoire pur atteint des scores respectables (8–24) mais sans garantie de progression : il dépend entièrement du hasard.

2. ε-greedy améliore systématiquement l'exploration par apprentissage positionnel, franchissant les scores de l'aléatoire sur les trois DAGs (jusqu'à +6 sur DAG 2).

3. Recuit simulé propose une montée plus régulière qu'ε-greedy, mais reste en retrait sur DAG 2 et égal sur DAG 3.

4. Tabu complet fournit les meilleurs résultats moyens sur tous les cas, creusant l'écart sur DAG 1 (+3 par rapport à ε-greedy) et égalant le meilleur score possible sur DAG 3.

Recommandation finale

Pour maximiser la protection dans ces graphes, l'heuristique Tabou (à liste taboue taille 20) est la méthode la plus robuste et la plus efficace, alliant exploration prolongée et exploitation optimale de l'espace des solutions.

7 Conclusion générale du TER

Au terme de cette étude, la méthode Tabou s'est révélée la méthode la plus performante pour isoler la propagation sur des DAG de taille comparable. En moyenne, elle atteint la meilleure solution en moins de 10 itérations, puis confirme qu'aucune amélioration n'est possible.

Cet effet ultra-rapide ouvre la voie à une variante « rapide » de la méthode Tabou : il suffit de mettre fin à la recherche dès qu'on observe 5 itérations consécutives sans amélioration du record global.

Cette approche combine les avantages suivants :
— Gain de temps : arrêt anticipé dès convergence, sans balayer inutilement tout le voisinage.
— Efficacité : conservation de la qualité optimale, puisque le meilleur score est atteint et validé.
— Simplicité de paramétrage : un seul paramètre de « stagnation » à calibrer (ici recommandé à 5).

Note : des tests plus poussés devront être conduits sur d'autres instances et avec une plage d'itérations élargie pour confirmer qu'aucune amélioration supplémentaire n'est possible au-delà de ces 5 itérations sans progrès.