# APPROX

Noé Péchereau, Anthony Verkindere, Virgile Hermant, Erwan Guerrier

June 2023

## 1 Énoncé

Une entreprise souhaite surveiller un réseau constitué d'un certain nombre de routeurs connectés entre eux d'une certaine manière. Pour cela, elle peut monitorer des routeurs. Un tel routeur permet de surveiller toutes ses connections. Le souhait de cette entreprise est d'avoir accès a toutes les connections en utilisant un minimum de routeurs.

## 2 Formulation du problème

| | |
|---|---|
| **Input:** | $G = (V, E)$ |
| **Validity:** | $S(I) = \{V' \subseteq V \mid \forall (u, v) \in E, u \in V' \vee v \in V'\}$ |
| **Cost:** | $m(V') = \lvert V' \rvert$ |
| **Objective:** | $O = \min$ |

Où G est notre graphe représentant le réseau, V l'ensemble de ses connexions et E l'ensemble des routeurs.

## 3 Algorithme exact polynomial dans le cas d'un arbre

La fonction *tree_surveillance_rec()* prend en entrée un réseau en forme d'arbre représenté par un objet Graph de la bibliothèque NetworkX. On initialise une variable *surveilles* comme un ensemble vide. Cet ensemble va contenir les identifiants des routeurs qui doivent être surveillés. On effectue ensuite un DFS sur l'arbre. Une fois que le DFS a terminé d'explorer tous les nœuds de l'arbre, on retourne l'ensemble *surveilles* qui contient les identifiants des routeurs à surveiller.

**Algorithm 1:** Surveillance d'arbre

**Entrée:** Un arbre *tree*, un nœud *node*, un nœud *parent*
**Sortie :** Un ensemble *surveilles* contenant les nœuds surveillés

```
1  Function treeSurveillance(tree, node, parent):
2      surveilles ← ensemble vide;
3      return treeSurveillanceRec(tree, node, parent, surveilles);
4  end
5  Function treeSurveillanceRec(tree, node, parent, surveilles):
6      for chaque n dans tree.neighbors(node) do
7          if n ≠ parent then
8              treeSurveillanceRec(tree, n, node, surveilles);
9              if n ∉ surveilles then
10                 Ajouter node à surveilles;
11             end
12         end
13     end
14     return surveilles;
15 end
16 treeSurveillance(tree, node, parent);
```

# 4 Algorithme d'approximation dans le cas général

La fonction *VCGreeedy()* prend en entrée un réseau en forme d'arbre représenté par un objet Graph de la bibliothèque *NetworkX*. L'idée ici est d'avoir accès à la liste des liaisons du graph (une simple liste de liaison ferait l'affaire). L'algorithme utilisé pour trouver un algorithme d'approximation pour le problème de couverture par sommet dans le cas général est "VCGreedy". *VCGreedy* est un algorithme à 2-approximations pour la couverture par sommet.

**Algorithm 2:** VCGreedy

**Entrée:** Un graphe $E$
**Sortie :** Un Vertex Cover $S$

```
1  VCGreedy(E);
2  S ← ∅;
3  E' ← E;
4  while E' ≠ ∅ do
5      Sélectionner une arête (u, v) ∈ E';
6      S ← S ∪ {u, v};
7      E' ← E' \ {e | u ∈ eouv ∈ e};
8  end
9  return S;
```

C'est un algorithme assez naïf qui a pour but de prendre les liaisons non surveillées et de noter un des deux noeuds comme surveillé à ses extrémités

mais qui est satisfaisant si on regarde le facteur d'approximation. Vous pouvez le voir en détail en lançant la partie "comparaison".

# 5 Comparaison des performances théoriques et pratiques de cet algorithme

On remarque que les performances sont identiques. Cependant le temps d'exécution de l'algorithme brute force est beaucoup plus élevé que l'algorithme de la question 2.

# 6 Faire un algorithme exact pour le cas général et comparer les performances théoriques et pratiques de l'algorithme d'approximation dans le cadre général

Dans le cas général, il n'existe pas d'algorithme connu résolvable en temps polynomial pour ce problème. Nous avons donc choisi d'implémenter un algorithme de façon brute force. Voici le pseudo-code:

**Algorithm 3:** Recherche exhaustive du vertex cover optimal

```
   Entrée: Un graphe E
   Sortie : Le vertex cover optimal S*
 1 Function BruteForce(E):
 2 |   S* ← E;
 3 |   foreach subset s of E do
 4 |   |   if s is a valid subset which cover each edge of V then
 5 |   |   |   if |s| < |S*| then
 6 |   |   |   |   S* ← s;
 7 |   |   |   end
 8 |   |   end
 9 |   end
10 |   return S*;
11 end
12 BruteForce(E);
```

En lançant notre liste de tests exhaustive, on aboutit à un facteur d'approximation moyen de 1,55, avec un facteur de 2 dans le pire des cas. Néanmoins, l'algorithme de brute-force ne peut pas être utilisé pour des graphes de plus de 17 noeuds en moyenne, car le temps devient trop conséquent pour la test-suite, ce qui le rend inutilisable dans notre cas, où le but est de surveiller de grands réseaux de routeurs. Par conséquent, l'utilisation de l'algorithme d'approximation est plus pertinent pour des gros graphes.