

Algorithmique en résolution de problèmes :
Bipartite Subgraph Problem

J. KOZIK

03-12-2025

1 Introduction

Le *Bipartite Subgraph Problem* est un problème fondamental en algorithmique et en théorie des graphes. Il consiste à extraire, à partir d'un graphe général, un sous-graphe biparti optimisant un critère donné, le plus souvent le nombre d'arêtes conservées. Ce problème est intimement lié à la détection et à l'élimination des cycles impairs, qui caractérisent les graphes non bipartis.

D'un point de vue algorithmique, le Bipartite Subgraph Problem est connu pour être NP-difficile.

2 Modélisation

2.1 État

Un état est défini par deux ensembles d'arêtes :

- $E_c \subseteq E$, l'ensemble des arêtes conservées ;
- $E_r \subseteq E$, l'ensemble des arêtes retirées.

Ces ensembles vérifient les invariants suivants :

$$E_c \cup E_r = E \quad \text{et} \quad E_c \cap E_r = \emptyset.$$

2.1.1 État initial

À l'état initial, toutes les arêtes sont conservées :

$$E_c = E \quad \text{et} \quad E_r = \emptyset.$$

2.1.2 Successeur

Soit un état $s = (E_c, E_r)$ et une arête $e \in E_c$. Le successeur s' obtenu par le retrait de e est défini par :

$$E'_c = E_c \setminus \{e\}, \quad E'_r = E_r \cup \{e\}.$$

2.1.3 État valide

Tout état respectant les invariants précédents est considéré comme valide, car il représente un sous-graphe du graphe initial.

2.1.4 État final

Un état s est final si le graphe induit $G_f = (V, E_c)$ est biparti, c'est-à-dire s'il ne contient aucun cycle impair.

3 Implémentations

Deux versions de l'algorithme de résolution du Bipartite Subgraph Problem ont été développées :

La première version est une approche naïve. Elle explore exhaustivement l'ensemble des états possibles sans appliquer d'heuristiques ni de stratégies particulières. Chaque arête est considérée à chaque étape, ce qui conduit à un espace de recherche exponentiel.

La seconde version intègre plusieurs améliorations destinées à réduire le temps de calcul. En particulier, j'ai introduit :

- Ignorer les arêtes \hat{e} sortantes \hat{z} (sticking-out edges) qui n'ont pas d'incidences dans un cycle impair, car elles ne peuvent jamais empêcher le graphe d'être biparti.
- Effectuer une vérification rapide de la bipartition à chaque état, en identifiant les arêtes \hat{e} problématiques \hat{z} dont les sommets sont assignés à la même couleur.

Ces améliorations permettent à l'algorithme d'explorer plus efficacement l'espace des solutions et d'obtenir des résultats exacts sur des instances plus grandes, tout en restant conceptuellement basé sur le backtracking.

3.0.1 Vérification d'un état terminal

Listing 1 – Vérification de bipartition d'un graphe

```
def is_bipartite(self, G: Graph):
    color = {}

    for v in G.get_vertices():
        if v not in color:
            queue = deque([v])
            color[v] = 0
            while queue:
                u = queue.popleft()
                for n in G.get_neighbors(u):
                    if n not in color:
                        color[n] = 1 - color[u]
                        queue.append(n)
                    elif color[n] == color[u]:
                        return False

    return True
```

Pour déterminer si un état est terminal, il suffit de vérifier si le sous-graphe actuel est biparti. Cette vérification est réalisée avec un parcours BFS.

3.1 Algorithme de backtracking

L'algorithme de backtracking explore récursivement le graphe des états afin de trouver un sous-graphe biparti. Si le graphe est biparti, l'état est immédiatement reconnu comme terminal et retourné comme solution.

Dans le cas contraire, l'algorithme génère l'ensemble des actions possibles, correspondant au retrait des arêtes encore présentes. Pour chaque action, un nouvel état successeur est construit et exploré récursivement.

- Dès qu'une solution est trouvée, l'exploration s'interrompt.
- Si aucune action ne mène à un état terminal, l'algorithme retourne `None`, indiquant l'absence de solution sur la branche courante.

Listing 2 – Algorithme de backtracking

```
def backtrack(self, assignment: Etat):  
    global liste_solution  
    G = Graph(assignment.get_V(), assignment.get_E_c())  
    if self.is_bipartite(G):  
        return assignment  
  
    for edge in assignment.actions():  
        new_assignment = self.succ(assignment, edge)  
        result = self.backtrack(new_assignment)  
        if result is not None:  
            liste_solution.append(result)  
        return result  
  
    return None
```

3.2 Version naive

3.2.1 Actions

Une action correspond au retrait d'une arête du sous-graphe courant. Ainsi, à partir d'un état donné, l'ensemble des actions possibles est constitué des arêtes encore présentes dans E_c .

Listing 3 – Ensemble des actions possibles

```
def actions(self) -> List[Tuple[int, int]]:  
    return self.get_E_c()
```

3.3 Version amélioré

3.3.1 Amélioration par détection des arêtes problématiques

Dans la version améliorée de l'algorithme de backtracking, une tentative de deux-coloration est utilisée afin d'identifier rapidement les arêtes responsables des conflits de bipartition. Contrairement à une vérification complète de bipartition, cette tentative n'interrompt pas l'exploration en cas de conflit, mais fournit une coloration partielle ou potentiellement incorrecte du graphe.

À partir de cette coloration, les arêtes dont les deux sommets se voient attribuer la même couleur sont identifiées comme *arêtes problématiques*. Ces arêtes appartiennent nécessairement à au moins un cycle impair et constituent donc des candidates prioritaires pour le retrait.

Listing 4 – Tentative de deux-coloration et détection des arêtes problématiques

```
def tentative_deux_coloration(self):
    color = {}
    G = Graph(self.get_V(), self.get_E_c())

    for v in G.get_vertices():
        if v not in color:
            queue = deque([v])
            color[v] = 0
            while queue:
                u = queue.popleft()
                for n in G.get_neighbors(u):
                    if n not in color:
                        color[n] = 1 - color[u]
                        queue.append(n)

    return color

def get_problematic_edges(self):
    l = []
    coloration = self.tentative_deux_coloration()
    for e in self.get_E_c():
        u, v = e
        if coloration[u] == coloration[v]:
            l.append(e)
    return l
```

3.3.2 Ignorance des arêtes sortantes

Une autre amélioration apportée à l'algorithme de backtracking consiste à ignorer les *arêtes sortantes* (*sticking-out edges*), c'est-à-dire les arêtes incidentes à au moins un sommet de degré 1. En effet, une telle arête ne peut appartenir à aucun cycle impair. Par conséquent, son retrait n'est jamais nécessaire pour rendre le graphe biparti.

Listing 5 – Détection des arêtes sortantes

```
def is_sticking_out_edge(self, uv: Tuple[int, int]) -> bool:
    for s in uv:
        if len(self.get_neighbors(s)) == 1:
            return True
    return False

def get_sticking_out_edges(self) -> List[Tuple[int, int]]:
    l = []
    for e in self.get_edges():
        if self.is_sticking_out_edge(e):
            l.append(e)
    return l
```

3.3.3 Actions dans la version améliorée

Dans la version améliorée de l'algorithme, l'ensemble des actions possibles est restreint afin de guider plus efficacement la recherche. Plutôt que de considérer toutes les arêtes restantes, seules les arêtes identifiées comme *problématiques* lors de la tentative de deux-coloration sont sélectionnées. Ces arêtes sont directement responsables des conflits de bipartition et leur retrait constitue donc un choix pertinent.

De plus, les arêtes sortantes sont explicitement exclues de cet ensemble.

Listing 6 – Ensemble des actions possibles version améliorée

```
def actions(self) -> List[Tuple[int, int]]:
    l = []
    sticking_out_edges = self.G.get_sticking_out_edges()
    for e in self.get_problematic_edges():
        if e not in sticking_out_edges:
            l.append(e)
    return l
```

4 Instances

4.1 Cas particulier : toy model

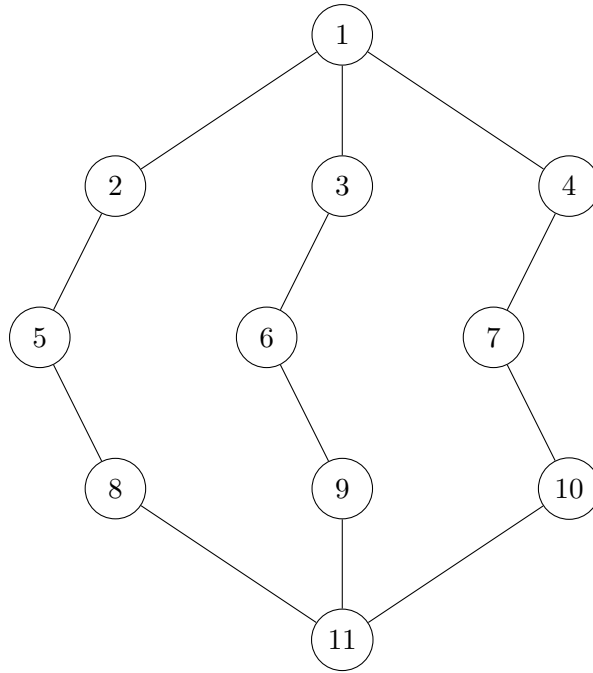


FIGURE 1 – Représentation du graphe étudié

Le graphe est biparti dès l'initialisation, donc aucune arête n'a besoin d'être enlevé. Par conséquent, l'algorithme s'arrête dès l'état initial.

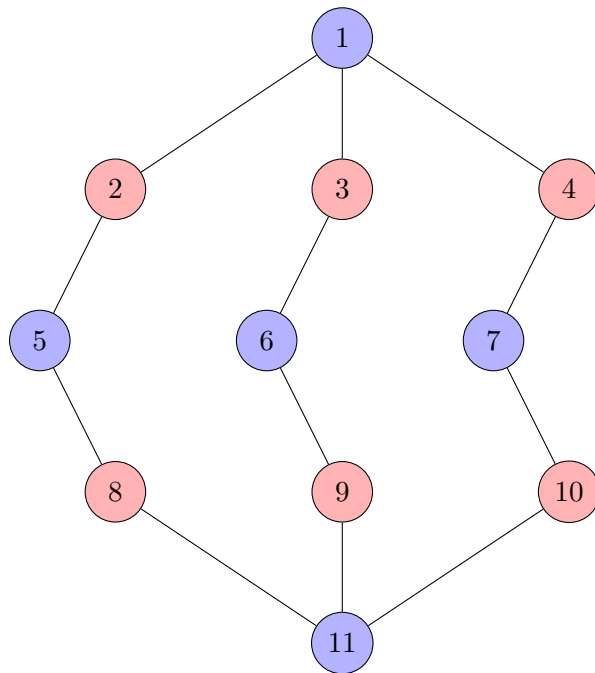


FIGURE 2 – Deux-coloration du graphe (graphe biparti)

4.2 Karate

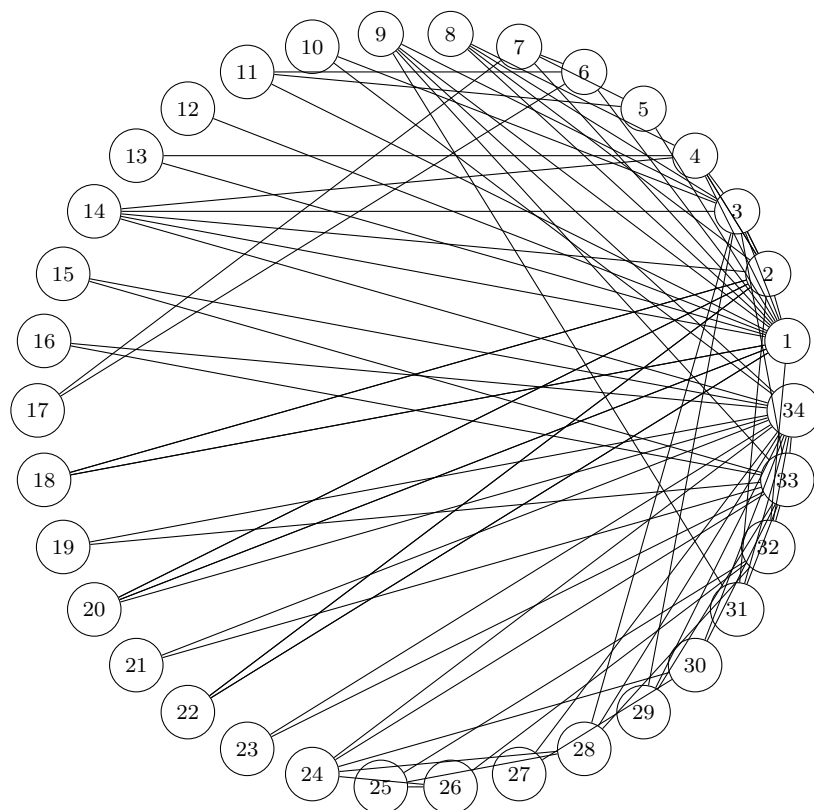


FIGURE 3 – Représentation du graphe Karate

4.2.1 Résultats Version naïve

Cette approche explore exhaustivement toutes les combinaisons possibles d'arêtes à retirer afin d'obtenir un sous-graphe biparti, sans appliquer d'heuristiques particulières ni d'élagage.

- **Nombre d'arêtes retirées : 57**
- **Nombre de solutions trouvées : 58**

Comme attendu, la version naïve explore l'ensemble de l'espace des états, ce qui entraîne un nombre important de solutions et un coût computationnel élevé, particulièrement pour des graphes de cette taille. Cette observation justifie l'introduction de la version améliorée avec heuristiques et élagages.

4.3 Résultats Version améliorée

La version améliorée de l'algorithme de backtracking utilise les heuristiques décrites précédemment, notamment la détection des arêtes problématiques et l'ignorance des arêtes sortantes. Ces améliorations permettent de

réduire le facteur de branchement et de guider la recherche vers les suppressions les plus pertinentes.

— **Nombre d'arêtes retirées : 28**

— **Nombre de solutions trouvées : 29**

Grâce à ces heuristiques, la version améliorée réduit de manière significative le nombre d'arêtes retirées et le nombre de solutions explorées par rapport à la version naïve. Cela illustre l'efficacité des stratégies de détection des arêtes problématiques et de filtrage des arêtes sortantes.

4.4 Conclusion

L'étude comparative des deux versions de l'algorithme de backtracking montre que les heuristiques introduites dans la version améliorée permettent de réduire significativement le nombre d'arêtes retirées et le nombre de solutions explorées. En ciblant les arêtes problématiques et en ignorant les arêtes sortantes, l'algorithme devient plus efficace tout en conservant sa complétude.