I Problématique

Avec la généralisation des véhicules électriques, il est important d'installer des bornes de recharge de véhicules électriques accessibles, mais sans en installer trop (ce qui augmente le coût global de leur installation et entretien). On suppose que le responsable de l'aménagement d'une communauté d'agglomération fait appel à nous pour réaliser un logiciel qui permettrait de déterminer quelles villes dans la communauté doivent accueillir un parking équipé de bornes de recharge. Ses contraintes sont les suivantes :

- *(Accessibilité)* Chaque ville doit posséder ses bornes, ou être directement reliée à une ville qui possède des bornes.
- *(Économie)* Le coût du projet doit être le plus bas possible, ce qui signifie que le nombre de bornes à construire doit être le plus petit possible.

La mission qui nous est confiée est de développer un logiciel qui permet :

1) de représenter les villes d'une communauté d'agglomération, et les routes qui les relient ;

2) de simuler la construction de parkings équipés de bornes de recharge dans les villes de la communauté ;

3) de s'assurer que la contrainte d'*accessibilité* est respectée ;

4) de calculer le coût d'une solution (le nombre zones de recharge à construire),

5) de minimiser ce coût.

Nous réaliserons ces tâches de façon incrémentale, afin d'avoir à la fin du semestre un logiciel fonctionnel.

II Villes et Routes

Pour simplifier le problème, nous considérons que toutes les routes sont à double sens, et de même longueur. Une route est donc un point d'accès rapide entre une ville A et une ville B. Par conséquent, une communauté d'agglomération peut-être représentée par un graphe non-orienté, où les noeuds représentent les villes, et les arêtes représentent les routes. Un exemple est représenté en Figure 1.
Une solution naive pour satisfaire la contrainte d'*Accessibilité* consiste à placer une zone de recharge dans chaque ville. On représente les villes possédant une zone de recharge par des noeuds colorés en bleu, comme en Figure 2.
Bien entendu, cette solution ne satisfait absolument pas la contrainte d'*Économie*, puisqu'il serait possible de retirer certaines zones de recharge. La Figure 3 présente une meilleure solution, où seules 4 zones de recharge sont construites.
Cependant, cette solution n'est pas encore optimale. Il est possible de satisfaire la contrainte d'*Accessibilité* avec seulement 3 zones de recharge, comme on le voit en Figure 4a. Pour toute communauté d'agglomération, il existe *au moins une* solution optimale. Il peut en exister plusieurs

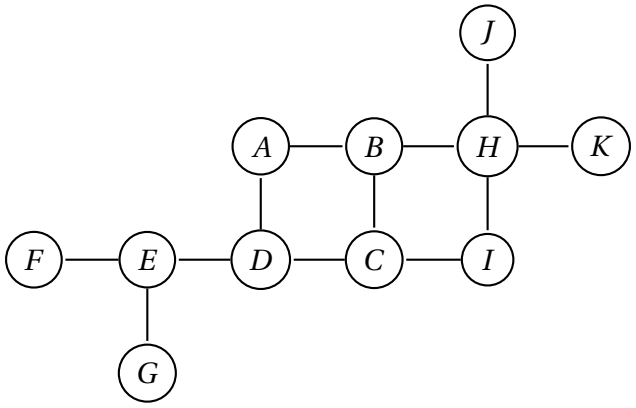


FIGURE 1 – Une communauté d'agglomération avec 11 villes.

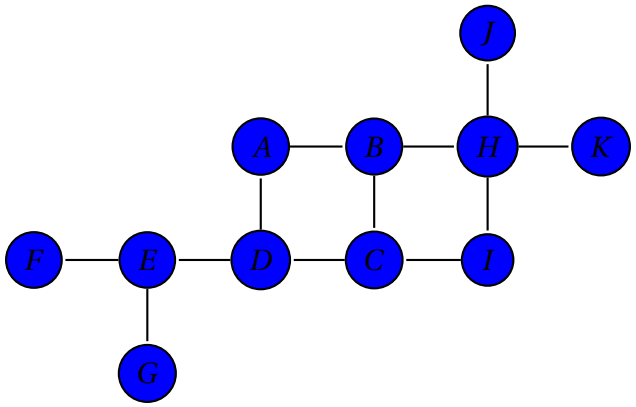


FIGURE 2 – Une communauté d'agglomération avec 11 villes, et 11 zones de recharge.

(c'est-à-dire plusieurs solutions qui ont le même nombre de zones de recharge n, tel qu'aucune solution avec $n-1$ zones de recharge ne satisfait l'*Accessibilité*). C'est par exemple le cas, ici, avec la Figure 4b.

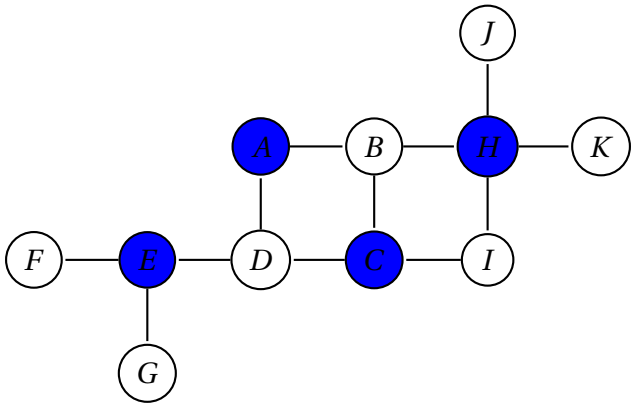
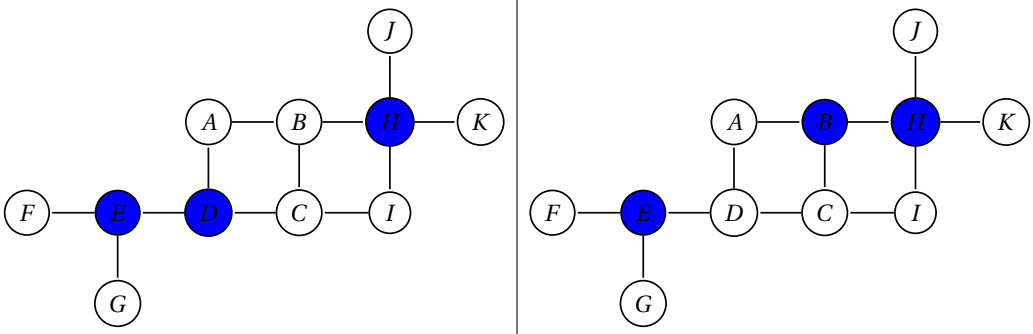


FIGURE 3 – Une communauté d'agglomération avec 11 villes, et 4 zones de recharge.



(a) Une solution optimale

(b) Une deuxième solution optimale

FIGURE 4 – Deux solutions optimales pour la communauté d'agglomération

III Instructions : Phase 1

1. Tâches à réaliser

Pour la première étape du projet, vous devez développer un programme qui permet à un utilisateur de configurer manuellement les zones de recharge d'une communauté d'agglomération. Au démarrage, le programme doit demander à l'utilisateur le nombre de villes. Pour l'instant, nous supposerons que le nombre de villes est inférieur à 26, par conséquent elles peuvent être nommées en fonction des lettres de l'alphabet. Retenez cependant que dans la suite du projet, il pourra y avoir un nombre quelconque de villes, avec des noms quelconques. Cela peut donc être dans votre intérêt d'anticiper cela dans la conception de votre code.
Une fois que le nombre de villes n est fixé, un menu s'affiche avec deux options :

1) ajouter une route ;

2) fin.

Dans le cas où l'option 1 est retenue, on demande à l'utilisateur les villes entre lesquelles il faut ajouter une route, puis on revient au menu précédent. Dans le cas où l'option 2 est retenue, l'utilisateur a terminé de représenter la communauté d'agglomération. Il peut maintenant essayer de trouver manuellement une solution au problème. La communauté d'agglomération est générée avec la solution naive qui consiste à placer une zone de recharge dans chaque ville. Ensuite, l'utilisateur fait face à un menu qui propose trois options :

1) ajouter une zone de recharge ;

2) retirer une zone de recharge ;

3) fin.

Quand l'option 1 est sélectionnée, on demande à l'utilisateur dans quelle ville il souhaite ajouter une zone de recharge, puis le programme ajoute la zone de recharge si c'est possible (c'est-à-dire s'il n'y a pas déjà une zone de recharge dans cette ville).
Quand l'option 2 est sélectionnée, on demande à l'utilisateur dans quelle ville il souhaite retirer une zone de recharge, puis le programme retire la zone de recharge si c'est possible (c'est-à-dire **si ce retrait ne viole pas la contrainte d'*Accessibilité***, et s'il y avait bien une zone de recharge dans cette ville). Dans le cas où le retrait est impossible, un message d'erreur doit indiquer à l'utilisateur pourquoi ce n'est pas possible (c'est-à-dire quelles villes se retrouveraient sans zone de recharge dans leur voisinage).
Après chaque action de l'utilisateur, un message renseigne la liste des villes qui possèdent actuellement une zone de recharge, par exemple la solution représentée en Figure 4b est représentée par :

```
B H E
```

qui indique que les villes B, H et E sont celles où se trouvent des zones de recharge (l'ordre n'a pas d'importance).
Enfin, si l'option 3 est sélectionnée, le programme s'arrête.

2. Remise du projet

Ce projet est à réaliser par groupes de **deux ou trois étudiants**, issus du **même groupe de TD**. Votre code source, correctement documenté, sera à remettre sur Moodle au plus tard le 17 Novembre 2023, sous forme d'une archive zip ou jar (un seul dépôt par binôme/trinôme).
Des conseils sur l'implémentation seront fournis prochainement sur Moodle.

IV Instructions : Phase 2

Le but est maintenant d'étendre le travail réalisé lors de la phase 1 de deux façons. La première est d'extraire une communauté d'agglomération stockée dans un fichier texte (ce qui évite de rentrer celle-ci manuellement). La seconde est de proposer et d'implémenter une algorithme permettant de retourner une solution au problème tout en respectant les contraintes d'accessibilité et d'économie.

1. Entrées-Sorties

Nous souhaitons maintenant pouvoir représenter une communauté d'agglomération sous forme d'un fichier texte. Cela permet d'éviter à l'utilisateur d'entrer manuellement les villes et les routes à chaque utilisation du programme. De plus, cela permet de sauvegarder une solution intéressante. Nous prenons toujours en exemple la communauté d'agglomération donnée en Figure 1.
Cette communauté peut-être représentée par le fichier `exemple.ca` suivant :

```
ville(A).
ville(B).
ville(C).
ville(D).
ville(E).
ville(F).
ville(G).
ville(H).
ville(I).
ville(J).
ville(K).
route(A,B).
route(A,D).
route(B,C).
route(B,H).
route(C,D).
route(C,I).
route(D,E).
route(E,F).
route(E,G).
route(H,I).
route(H,J).
route(H,K).
```

On place d'abord toutes les déclaration de villes (l'ordre n'a pas d'importance), puis les déclarations des routes. L'ordre des routes n'a pas d'importance non plus, ni le sens dans lequel on écrit une route : `route(A,B).` et `route(B,A).` sont équivalents. On peut également représenter le fait qu'une ville contient une zone de recharge. Par exemple, si on considère la communauté représentée en Figure 4b, il suffit d'ajouter au fichier `exemple.ca` les lignes suivantes (après la déclaration des routes) :

```
recharge(B).
recharge(E).
recharge(H).
```

Encore une fois, l'ordre de ces déclarations n'est pas important. Lors de l'extraction de la communauté d'agglomération, pensez à vérifier que le fichier est valide. Si ce n'est pas le cas, vous devez informer l'utilisateur sur la source du problème.

2. Un algorithme naïf

L'Algorithme 1 permet de trouver une solution potentielle du problème. Cependant, cet algorithme (plutôt naïf) ne fournit qu'une solution approximative : il n'y a aucune garantie que cette solution soit optimale. On appelle le *score* d'une solution le nombre de zones de recharge construites. Le but est donc ici de minimiser le score.

Algorithme 1 : Un algorithme d'approximation (naïf)

Entrées : Une communauté CA, un entier k
Output : Une solution naïve du problème
int $i = 0$;
tant que $i < k$ **faire**
    choisir au hasard une ville $v \in CA$;
    si *v a une zone de recharge* **alors**
        retirer la zone de recharge de v;
    sinon
        ajouter une zone de recharge dans v;
    fin
    $i++$;
fin
retourner CA

Cette méthode garantit que l'algorithme s'arrête au bout de k itérations. Cet algorithme peut fournir une bonne base, mais il est (largement) améliorable. Un premier problème est que si k n'est pas assez grand, on risque de s'arrêter sans avoir retiré suffisamment de zones de recharge. On peut proposer l'amélioration suivante (Algorithme 2).

Algorithme 2 : Un algorithme d'approximation (un peu moins naïf)

Entrées : Une communauté CA, un entier k
Output : Une solution (légèrement moins) naïve du problème
int $i = 0$;
int $scoreCourant = score(CA)$;
tant que $i < k$ **faire**
    choisir au hasard une ville $v \in CA$;
    si *v a une zone de recharge* **alors**
        retirer la zone de recharge de v;
    sinon
        ajouter une zone de recharge dans v;
    fin
    si $score(CA) < scoreCourant$ **alors**
        $i = 0$;
        $scoreCourant = score(CA)$;
    sinon
        $i++$;
    fin
fin
retourner CA

Avec cette deuxième version de l'algorithme, on s'arrête lorsqu'on n'a pas été capable d'améliorer

le score k fois de suite. Si k est suffisamment grand (par exemple, k = le nombre de villes de la communauté), on peut envisager de trouver la solution optimale pour de nombreux (petits) problèmes. Cependant, ça ne sera pas suffisant pour résoudre des problèmes de grande taille (c'est-à-dire avec un grand nombre de villes). Soyez imaginatifs pour trouver un meilleur algorithme, qui permettrait de trouver la solution (approximative) avec le score le plus bas possible (voire la solution optimale).

3. Tâches à réaliser

Vous devez implémenter un programme qui prend en entrée, **sur la ligne de commande**, le chemin vers un fichier texte qui représente une communauté d'agglomération. Rappelons que les paramètres de la ligne de commande sont utilisables via le paramètre `String[] args` de la méthode `main`. Le programme propose un menu à l'utilisateur :

1) résoudre manuellement ;

2) résoudre automatiquement ;

3) sauvegarder ;

4) fin.

Dans le cas où la première option est choisie, le programme fonctionne comme celui demandé dans la partie 1 du projet : l'utilisateur ajoute ou retire lui même les zones de recharge. Si le fichier qui décrit la communauté d'agglomération ne contient aucune zone de recharge, l'utilisateur commence avec la solution naïve (une zone de recharge dans chaque ville). Il en va de même si le fichier indique que des zones de recharge existent, mais que cela ne respecte pas la contrainte d'*Accessibilité*. Sinon, on commence avec exactement les zones de recharge qui sont décrites dans le fichier.
Si l'option 2 est choisie, l'algorithme que vous avec implémenté (soit l'Algorithme 1, soit l'Algorithme 2, soit un autre de votre choix), est appelé, avec encore une fois la solution naïve comme point de départ s'il n'y a aucune zone de recharge dans le fichier (ou si le fichier décrit une communauté qui ne respecte pas l'*Accessibilité*), ou avec les zones de recharge décrites dans le fichier.
L'option 3 permet de sauvegarder la dernière solution trouvée (manuellement ou automatiquement). Lorsqu'elle est choisie par l'utilisateur, le programme demande à l'utilisateur le chemin vers le fichier où la solution doit être enregistrée, puis effectue la sauvegarde avant de revenir au menu.
Enfin, la dernière option permet de quitter le programme.

4. Remise du projet

La deuxième partie du projet doit être réalisée par les mêmes binômes/trinômes que la première partie. Votre code source, correctement documenté, sera à remettre sur Moodle au plus tard le 22 Décembre 2023, sous forme d'une archive jar ou zip (un seul dépôt par binôme/trinôme). Avant d'exporter votre projet sous forme d'archive, pensez à le renommer en utilisant vos noms!
En plus du code source, vous devrez fournir un fichier `README` qui précisera :

- quelle classe doit être utilisée pour exécuter votre programme (c'est-à-dire où se trouve la méthode main),
- quelles fonctionnalités vous avez correctement implémentées, et lesquelles sont manquantes ou présentent des problèmes.

Le non-respect de certaines consignes pourra être pénalisé par un malus dans la note du projet.