TP3 : Algorithmes gloutons

L'objectif de ce TP est d'implanter l'algorithme glouton pour SETCOVER dans le plan vu en cours. On fournit pour cela :

— un fichier `tests.cpp` qui contient le `main` ;
— deux fichiers `Affichage.h/cc` pour l'affichage graphique ;
— deux fichiers `SetCover.h/.cc` (`SetCover.cc` **est l'unique fichier à modifier**) ;
— un `Makefile`.

On représente les coordonnées d'une maison par une `structure` `coord`, qui contient deux champs entiers `x` (abscisse) et `y` (ordonnée). La liste des maisons est un tableau de `coord`, une maison étant identifiée par son indice. On indique le placement des émetteurs grâce à un tableau de booléens `emetteurs`, qui contient `true` en case `i` s'il y a un émetteur sur la maison `i`, et `false` sinon.
Le fichier de test produit des représentations graphiques dans les fichiers `Maison.svg`, `Emetteurs.svg` et `EmetteursOpt.svg`.

À vous de jouer !

1. Compléter la fonction `coord* maisonsAleatoires(int n, int l, int h)` qui génère n maisons aléatoires dans le rectangle $[10, l-10] \times [10, h-10]$ (c'est-à-dire le rectangle de largeur `l` et de hauteur `h`, privé d'une bande d'épaisseur 10 de chaque côté).
   Test. *On peut utiliser l'option* `1. Réutiliser une graîne` *et vérifier si le résultat est identique à la figure 1.*

FIGURE 1 – Dix maisons obtenues avec la graîne 2021.

2. Compléter la fonction `bool couvre(int i, int j, coord* maisons, int dcouv)` qui retourne vrai si, et seulement si, les maisons i et j se situent à moins de `dcouv` l'une de l'autre.
   Remarque. *Ne pas utiliser de bibliothèque mathématique pour cette question. En particulier, trouver une solution qui n'utilise pas de calcul de racine carrée !*
3. Compléter la fonction `bool** graphe(int n, coord* maisons, int dcouv)` qui contruit le graphe G dont les sommets sont les maisons, deux maisons i et j étant reliées par une arête si et seulement si elles sont à distance inférieure ou égale à `dcouv`. Le graphe G est représenté par une *matrice d'adjacence* `G` telle que `G[i][j] = true` s'il y a une arête entre i et j, et `G[i][j] = false` sinon.
   Test. *Les graphes des exemples fixés de taille* 5, 10 *et* 15 *sont représentés sur la figure 2.*
4. Compléter la fonction `int prochaineMaison(int n, bool** G, bool* couvertes)` qui retourne l'indice de la prochaine maison sur laquelle placer un émetteur. Le tableau `couvertes` indique quelles maisons sont déjà couvertes.

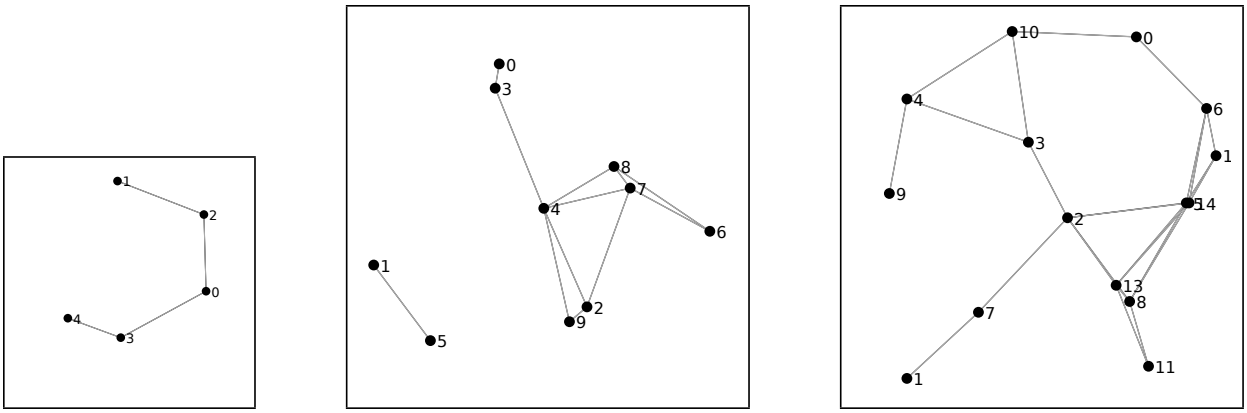

FIGURE 2 – Graphe des exemples fixés à cinq, dix et quinze maisons.

Test. *Pour tester cette fonction, on peut vérifier que le premier émetteur à placer pour l'exemple fixé à dix maisons est sur la maison* 4*. Pour un test complet, il faut avoir effectué la question suivante.*

Attention. *Si l'on ne crée pas le graphe avant de chercher le premier émetteur à placer, cela provoque une erreur puisque le graphe passé en paramètre de la fonction est un pointeur nul ! Cette remarque s'applique pour les questions suivantes : il faut toujours commencer par construire le graphe.*

5. Compléter la fonction `int placementGlouton(int n, bool** G, bool* emetteurs)` qui trouve un ensemble d'émetteurs qui couvre toutes les maisons, avec l'algorithme glouton vu en cours. La fonction complète le tableau `emetteurs` et renvoie le nombre d'émetteurs placés.

Test. *Les émetteurs trouvés pour les exemples fixés de taille* 5*,* 10 *et* 15 *sont représentés sur la figure 3, à comparer avec les fichiers* `Emetteurs.svg` *produit par votre code. Pour les exemples fixés de taille* 20*,* 30 *et* 50*, l'algorithme glouton trouve des solutions de taille* 4*,* 10 *et* 8 *respectivement.*

Efficacité. *Essayer d'appliquer votre code sur des exemples aléatoires de grande taille. Par exemple, il doit être possible d'obtenir le résultat pour 1500 maisons en une ou deux secondes.*

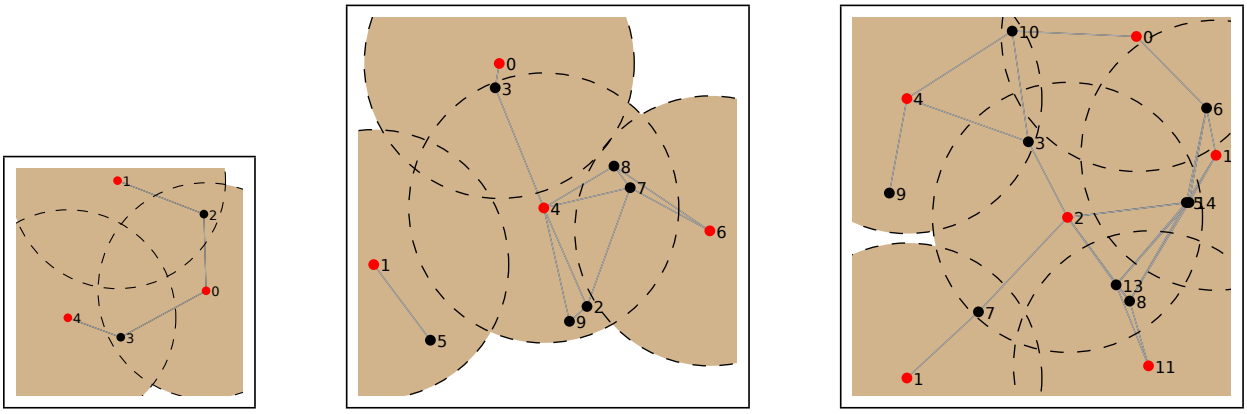

FIGURE 3 – Émetteurs trouvés par l'algorithme glouton pour les exemples fixés à cinq, dix et quinze maisons.

6. Compléter la fonction `placementOptimal(int n, bool** G, bool* emetteurs)` qui renvoie une solution optimale. Pour cela, on peut effectuer une recherche exhaustive en testant tous les sous-ensembles possibles et en gardant la meilleure solution. Chaque sous-ensemble est représenté par un tableau de n booléens, qu'on peut interpréter comme l'écriture en base 2 d'un entier (par exemple {`true, false, true, true`} est interprété comme $1101_2 = 13$). On peut tous les parcourir en partant de 0 (tableau de `false`) et en incrémentant progressivement cet entier jusqu'à arriver à $2^n - 1$ (tableau de `true`).

Remarques. *Cette fonction demande un peu d'initiative, ne pas hésiter à chercher un peu sur internet si nécessaire. D'autre part, il est très recommandé d'écrire des fonctions auxiliaires pour simplifier la programmation, par exemple une fonction qui teste si un ensemble d'émetteurs est suffisant, c'est-à-dire couvre toutes les maisons.*

Test. *La figure 4 représente des solutions optimales pour les exemples fixés à cinq, dix et quinze maisons. L'exemple fixé à vingt maisons possède une solution à quatre émetteurs et celui à trente sommets une solution à sept émetteurs. Je ne sais pas quelle est la solution optimale de l'exemple à cinquante sommets !*

Efficacité. *Votre code doit pouvoir traiter des exemples jusqu'à une vingtaine de sommets en moins d'une seconde.*

Défi. *Pour celles et ceux que cela intéresse, l'objectif est de trouver une implantation aussi rapide que possible de la fonction* `placementOptimal`*. Dans les techniques possibles à utiliser, on peut parcourir les sous-ensembles par taille croissante, partir de la valeur renvoyée par* `placementGlouton` *pour s'aider, accélerer le parcours des sous-ensembles avec des codes de Gray* [1]*, changer la représentation du graphe et utiliser des listes d'adjacence, etc. Pour information, mon code permet de calculer la solution optimale pour l'exemple fixé à trente sommets en env. 2,5 secondes.* ***Attention : si vous avez un code qui fonctionne, pensez à le sauvegarder avant de l'optimiser pour pouvoir revenir en arrière si votre optimisation casse tout !***

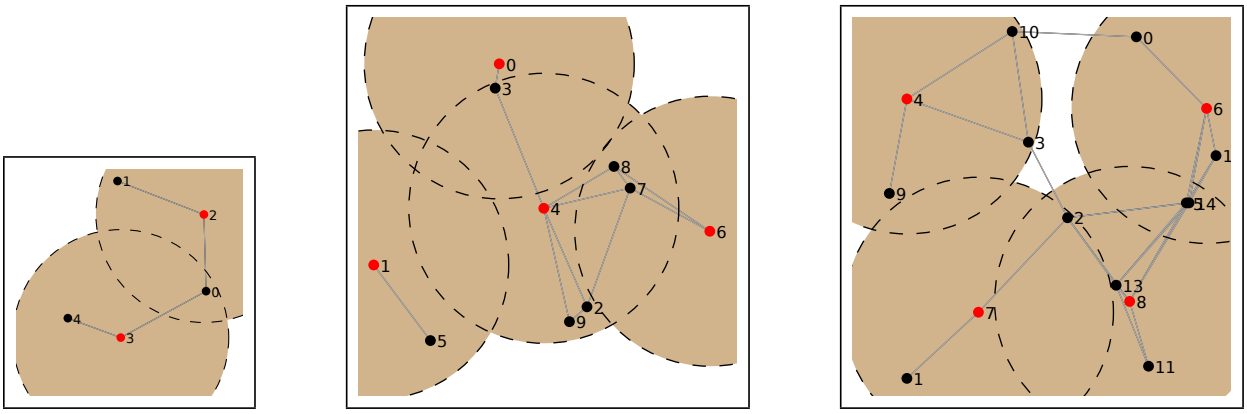

FIGURE 4 – Solutions optimales des exemples fixés à cinq, dix et quinze maisons.

1. `https://fr.wikipedia.org/wiki/Code_de_Gray`