**Rapport projet d’optimisation combinatoire**

Contents

[**1)** **Modélisation linéaire** 2](#_Toc57537167)

[**2)** **Traduction d’une instance en modèle** 3](#_Toc57537168)

[**L’utilisation du programme :** 3](#_Toc57537169)

[**Le tableau des résultats de GLPK :** 4](#_Toc57537170)

[**3)** **Algorithme glouton** 5](#_Toc57537171)

[**Implémentation de Glouton :** 5](#_Toc57537172)

[**Le tables des résultats obtenus avec notre algorithme glouton :** 6](#_Toc57537173)

[**4)** **Glouton amélioré** 7](#_Toc57537174)

[**Notre implémentation de Glouton amélioré :** 7](#_Toc57537175)

[**Le tableau des résultats obtenus avec cet algorithme :** 8](#_Toc57537176)

[**5)** **Comparaisons des métaheuristiques** 10](#_Toc57537177)

[**6)** **Comparaisons des métaheuristiques** 12](#_Toc57537178)

# **Modélisation linéaire**

Le problème est la minimisation du coût de déploiement.

La contrainte est que toutes les cibles doivent être couvertes par (au moins) un capteur.

Les informations suivantes nous sont données :

* Il y a M cibles et N capteurs
* Un capteur i couvre une cible j ssi i appartient à Vj
* Vj représente la liste des capteurs couvrants la cible j
* Pour un capteur i, on a son coût unitaire Ci

On peut en déduire le modèle suivant :

Minimiser

S.c :

Explications :

* La variable Xi vaut 1 si le capteur est déployé et 0 sinon (ligne 3).
* Il faut minimiser le coût des capteurs déployés soit XiCi (ligne 1).
* Toutes les cibles doivent avoir au moins un capteur qui les couvre. Donc la somme des capteurs qui couvre une cible doit être supérieure ou égale à 1 et ce pour chaque cible (ligne 2).

# **Traduction d’une instance en modèle**

Maintenant que le modèle linaire a été déterminé, il est possible de convertir les fichiers d’instance en fichier de modèle (.lp).

Le format d’un fichier de modèle est :

Minimize

z : C1 x1 + {…} + Cn xn // Où Ci sera remplacé par sa valeur

Subject To

xV1 + xV2 + xV3 + {…} >= 1 // Où les Vi seront remplacés par leur valeur

Binaries

x1

{…}

xN

End

Le programme lit les fichiers d’instance et déduit les variables :

* M et N qui sont à la première ligne du fichier
* Le coût de déploiement de chaque capteur
* Le nombre de capteurs et leur numéro pour couvrant chaque cible

Voici le fonctionnement des variables du programme :

* M : int m;
* N : int n;
* Ci : int cost[] = (int\*)malloc(n \* sizeof(int));
* Vj : int cibles[][] = (int\*\*)malloc(m \* sizeof(int\*)); cibles[j][] = (int\*)malloc((nbc + 1) \* sizeof(int\*));
* La variable nbc est le nombre de nombre de capteurs et est stockée dans l’emplacement cibles[j][0]

Il n’y a pas de procédures de test mais il programme détecte une erreur et l’affiche.

## **L’utilisation du programme :**

**TradInstance.exe inst41.txt**

TradInstance.exe est le nom de l’exécutable, il est possible de le renommé et il n’est pas nécessaire d’utiliser une extension sous Linux.

Inst41.txt correspond au fichier d’instance, il est possible d’y indiquer le chemin vers le fichier.

Le fichier de sortie aura le même nom que le fichier d’entrer dans le dossier de l’exécutable, seul l’extension change de « .txt » en « .lp ».

## **Le tableau des résultats de GLPK :**

Le terme « score » utilisé dans le sujet est aussi utilisé dans l’heuristique donc Solution Optimale (S.O) sera utilisé pour le remplacer dans les tableaux de résultats.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Instance | S.O | Instance | S.O | Instance | S.O |
| Inst41 | 429 | Inst49 | 641 | Inst58 | 288 |
| Inst42 | 512 | Inst51 | 253 | Inst59 | 279 |
| Inst43 | 516 | Inst52 | 302 | Inst61 | 138 |
| Inst44 | 494 | Inst53 | 226 | Inst62 | 146 |
| Inst45 | 512 | Inst54 | 242 | Inst63 | 145 |
| Inst46 | 560 | Inst55 | 211 | Inst64 | 131 |
| Indt47 | 430 | Inst56 | 213 | Inst65 | 161 |
| Inst48 | 492 | Inst57 | 293 |  |  |

# **Algorithme glouton**

Pour créer le programme Glouton, on réutilise la partie du traducteur d’instance qui permet de lire les fichiers « instances » (ex : inst41.txt).

Nous utilisons une classe que nous avons nommée C\_Solution et qui contient plusieurs membres :

* S\_Cout, le Cout de la solution qui remplace le terme « score »
* S\_nbr\_capteurs, le nombre de capteurs de l’instance
* S\_nbr\_cibles, le nombre de cibles de l’instance
* S\_M qui correspond à M’ dans l’algorithme Glouton
* S\_Tab\_Vij, le tableau des capteurs (j) qui couvrent les cibles (i)
* S\_Tab\_Vji, un tableau des cibles (i) couvertes par les capteurs (j)
* S\_Tab\_Cout, le tableau de cout de déploiement des capteurs
* S\_Xi, la liste des capteurs déployés
* S\_Cibles\_Couvertes, la liste des cibles couvertes

Quant aux Méthodes de la classe :

* CalculeCibleCouvertes() créée et met à jours la liste des cible couverte
* CreationTabVji() créée le tableau S\_Tab\_Vji à partir du tableau S\_Tab\_Vij
* Heuristique() applique notre critère de choix. Ici nous avons opté pour déployer le capteur qui aura le meilleur rapport : cibles non couvertes qui seront couvertes / cout du déploiement.
* CalculCout() calcule le cout de la solution S\_Cout
* AlgorythmeGlouton() met en place l'algorithme Glouton. C’est-à-dire, qu’il va utiliser les méthodes et membres de la classe afin de réaliser l’algorithme.

## **Implémentation de Glouton :**

while(S\_M > 0) {

int i = Heuristique();

S\_Xi.at(i) = 1;

S\_M = S\_nbr\_cibles;

CalculeCiblesCouvertes();

for(int j = 0; j < S\_nbr\_cibles; j++) {

S\_M -= S\_Cibles\_Couvertes.at(j);

}

}

L’heuristique choisi automatiquement le capteur le plus rentable. C’est pourquoi ce morceau de code nécessite de voir l’implémentation de l’heuristique.

CalculeCiblesCouvertes();

std::vector<float> Score;

int Index = 0;

for( int i = 0; i < S\_nbr\_capteurs ; i++){

Score.push\_back(0.0f);

for( int j = 0 ; j < S\_Tab\_Vji.at(i).size() ; j++){

if (S\_Cibles\_Couvertes.at(S\_Tab\_Vji.at(i).at(j) - 1) == 0 ){

Score.at(i)++;

}

}

Score.at(i) = (float) Score.at(i) / S\_Tab\_Cout.at(i);

if (Score.at(i) > Score.at(Index)){

Index = i;

}

}

## **Le tables des résultats obtenus avec notre algorithme glouton :**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Instance | S.O | Gap | Instance | S.O | Gap | Instance | S.O | Gap |
| Inst41 | 463 | 0.08 | Inst49 | 747 | 0.17 | Inst58 | 323 | 0.12 |
| Inst42 | 582 | 0.14 | Inst51 | 289 | 0.14 | Inst59 | 312 | 0.12 |
| Inst43 | 598 | 0.16 | Inst52 | 348 | 0.15 | Inst61 | 159 | 0.15 |
| Inst44 | 548 | 0.11 | Inst53 | 246 | 0.09 | Inst62 | 170 | 0.16 |
| Inst45 | 577 | 0.13 | Inst54 | 255 | 0.05 | Inst63 | 161 | 0.11 |
| Inst46 | 615 | 0.10 | Inst55 | 236 | 0.12 | Inst64 | 149 | 0.14 |
| Indt47 | 476 | 0.11 | Inst56 | 251 | 0.18 | Inst65 | 196 | 0.22 |
| Inst48 | 533 | 0.08 | Inst57 | 326 | 0.11 |  |  |  |

# **Glouton amélioré**

Nous utilisons le programme déjà créé pour glouton et nous y ajouton deux nouveaux membres à notre classe C\_Solution :

* S\_R, le nombre de redondance dans les cibles couvertes par les capteurs déployés.
* S\_Capteurs\_Inutiles, une liste de capteurs dit inutiles. C’est à dire, des capteurs que s’ils sont désactivé, la solution reste valide et le cout est moindre. Tous les capteurs inutiles ne peuvent être retiré en même temps car ils sont dépendants entre-eux.

Quant aux Méthodes ajoutées :

* CalculeRedondance() défini quels capteurs sont inutiles et calcule le nombre de redondance S\_R.
* Heuristique2() choisi le capteur à retirer. On a choisi de retirer le capteur (inutile) ayant le cout de déploiement le plus élevé.
* AlgorthymeGloutonAmeliorer() met en place l’algorithme amélioré en combinant utilisant l’algorithme glouton et les nouveautés ajoutées pour réoptimiser la solution.

## **Notre implémentation de Glouton amélioré :**

AlgorythmeGlouton();

CalculeRedondance();

while(S\_R > 0){

int H2 = Heuristique2();

S\_Xi.at(H2) = 0;

CalculeRedondance();

}

Il nous faut aussi voir le calcule de redondance et la seconde heuristique.

L’heuristique :

int Index = 0;

for (int i = 0; i < S\_Capteurs\_Inutiles.size(); i++) {

if (S\_Tab\_Cout.at(S\_Capteurs\_Inutiles.at(i)) > S\_Tab\_Cout.at(S\_Capteurs\_Inutiles.at(Index))){

Index = i;

}

}

La redondance :

S\_Capteurs\_Inutiles.clear();

for (int i = 0 ; i < S\_nbr\_capteurs ; i++ ) {

if (S\_Xi.at(i) == 1) {

int test = 1;

for (int j = 0; j < S\_Tab\_Vji.at(i).size(); j++) {

int test2 = 0;

for (int k = 0; k < S\_Tab\_Vij.at(S\_Tab\_Vji.at(i).at(j) - 1).size(); k++) {

if (S\_Xi.at(S\_Tab\_Vij.at(S\_Tab\_Vji.at(i).at(j)- 1).at(k) - 1) == 1) {

if(i + 1 != S\_Tab\_Vij.at(S\_Tab\_Vji.at(i).at(j)- 1).at(k)){

test2 = 1;

}

}

}

if(test2 == 0) {

test = 0;

}

}

if(test == 1) {

S\_Capteurs\_Inutiles.push\_back(i);

}

}

}

S\_R = S\_Capteurs\_Inutiles.size();

## **Le tableau des résultats obtenus avec cet algorithme :**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Instance | S.O | Gap | Instance | S.O | Gap | Instance | S.O | Gap |
| Inst41 | 432 | 0.01 | Inst49 | 665 | 0.04 | Inst58 | 311 | 0.08 |
| Inst42 | 529 | 0.03 | Inst51 | 269 | 0.06 | Inst59 | 292 | 0.05 |
| Inst43 | 537 | 0.04 | Inst52 | 330 | 0.09 | Inst61 | 142 | 0.03 |
| Inst44 | 506 | 0.02 | Inst53 | 232 | 0.03 | Inst62 | 156 | 0.07 |
| Inst45 | 518 | 0.01 | Inst54 | 250 | 0.03 | Inst63 | 157 | 0.08 |
| Inst46 | 594 | 0.06 | Inst55 | 212 | 0.00 | Inst64 | 140 | 0.07 |
| Indt47 | 447 | 0.04 | Inst56 | 225 | 0.06 | Inst65 | 186 | 0.16 |
| Inst48 | 525 | 0.07 | Inst57 | 306 | 0.04 |  |  |  |

Il est important à noter que la valeur que nous obtenons peut être légèrement différente que celle d’une autre solution entre-autres celle du sujet. Cependant, les valeurs obtenues par l’algorithme glouton amélioré sont meilleures que celle obtenues par l’algorithme glouton classique.

# **Comparaisons des métaheuristiques**

Méthodes heuristique utilisée :

|  |  |  |
| --- | --- | --- |
| Algorithme glouton  (Amélioré) | Simple, il donne une solution locale rapidement. | L’algorithme ne garantit pas une solution globale du problème |

Méthodes métaheuristique utilisée :

|  |  |  |
| --- | --- | --- |
| Algorithme génétique | Détermine une solution en cas de méthode non définie/complexe, en un temps raisonnable. | À éviter par leur coût en calcul.  Problème de ‘convergence prématurée’. |

Méthodes métaheuristiques proposées :

|  |  |  |
| --- | --- | --- |
|  | Avantages | Défauts |
| Recherche taboue | Bon dans les calculs simples,  Permet de déterminer les extrema locaux | Précision demandée relativement faible (besoin d’une mémoire) |
| Recuit simulé | La complexité réside dans la formule probabiliste, algorithme facile à coder.  Solutions optimales en général. | Nécessité d’une formule probabiliste représentative du problème,  Beaucoup de paramètres définis empiriquement. |
| Optimisation par essaims particulaires | Optimisé pour prédire les mouvements de foules,  ou autre élément dépendant de son voisinage. | Applicable à des cas particuliers suivant une dépendance du voisinage. |
| Algorithme de colonies de fourmis | Adaptation avec les problèmes dynamiques.  Utiliser pour optimiser les trajets routiers. | Pose problème avec les structures sans voisinage.  Peu de théorie, basé sur des expériences réelles.  Aléatoire non négligeable. |

Dans le contexte du projet, il s’agit de problèmes de type minimisation / maximisation. Les différentes métaheuristiques proposées ci-dessus ne sont pas nécessairement adaptées à la tâche.

**Recherche taboue :**

La recherche taboue est ici adaptée du fait des calculs relativement légers. De plus, les résultats attendus ne nécessitent pas de grandes précisions sur l’ordre, de 10^0 à 10^-3. Il faut cependant procéder à une étude qui permet de majorer et minorer la solution afin de s’assurer de l’authenticité de la solution.

*Verdict :* **Applicable**

**Recuit simulé :**

Le fondement de cette métaheuristique impose l’utilisation d’une formule probabiliste basée sur des statistiques et / ou expérimentations. Or, ici le problème ne permet pas à première vue d’établir une telle formule. Cette métaheuristique est donc à mettre de côté dans le cadre de cette application.

*Verdict :* **Pas adapté**

**Optimisations par essaims particulaires** :

Cette métaheuristique est notamment utilisé pour l’étude de foules, où les proches voisins évoluent dynamiquement. Dans le cadre d’une étude de cas indépendante du temps, l’optimisation par essaims particulaires n’est pas adaptée au sujet, malgré les équations fondamentales de l’algorithme proches du glouton.

*Verdict :* **Pas adapté**

**Algorithme de colonies de fourmis :**

Même si le nom en dit long sur son domaine d’étude, il est aussi utilisé pour optimiser d’autres mouvements comme les trajets routiers. Minimiser le temps de trajet ou la distance parcourue. Cet algorithme peut en effet se voir être la source de résultats plus optimisés que ceux obtenus avec les algorithmes gloutons. C’est notamment le cas de problème plus complexe concernant les flux routiers.

*Verdict :* **Résultats Variables**

# **Comparaisons des métaheuristiques**