**Rapport projet d’optimisation combinatoire**

1. **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).

1. **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]

Le tableau des résultats de GLPK :

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 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 |  |  |

1. **Algorithme glouton**

Pour créer le programme Glouton , on a pris une partie du traducteur d’instance (celle qui permet de lire les instance),

Puis nous avons créé une classe C\_Solution qui contient plusieurs Membre :

-S\_Cout qui est le Cout de la solution (Son score).

-S\_nbr\_capteurs qui est le nombre de capteurs présent dans l’instance.

-S\_nbr\_cibles qui est le nombre de cibles présentes dans l’instance.

-S\_M qui jouera le rôle de M’ dans l’algorithme Glouton

-S\_Tab\_Vij qui est le Tableau de capteurs j qui couvrent les cibles i

-S\_Tab\_Vji qui est le Tableau des cibles i couvertes par les capteurs j

-S\_Tab\_Cout qui est le tableau des couts des capteurs.

-S\_Xi qui es la liste des capteurs déployés.

-S\_Cibles\_Couvertes qui est la liste des cibles couvertes

Quant aux Méthodes de la classe :

-CalculeCibleCouvertes() vas nous servir à créer une liste des cible couverte

-CreationTabVji() qui vas nous servir à créer S\_Tab\_Vji à partir de S\_Tab\_Vij

-Heuristique() est notre critere de choix ici nous avons opté pour passer à 1 le capteur qui augmentera le plus S\_Cout

- CalculCout() nous sert à calculer S\_Cout

- AlgorythmeGlouton() comme son nom l'indique c'est l'algorithme Glouton

Les résultats obtenus avec notre algorithme glouton sont répertorier dans ce tableau :

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 |  |  |  |

1. **Glouton amélioré**

Pour améliorer notre algorithme glouton nous avons ajouté deux membres à notre classe C\_Solution :

* S\_R qui comptera le nombre de redondance.
* S\_Capteurs\_Inutiles qui sera une liste de capteur dit inutiles

Quant aux Méthodes ajoutées :

* CalculeRedondance() : qui vas définir quelle capteur est inutile ou non et vas nous calculer S\_R.
* Heuristique2() : qui vas nous choisir le capteur inutile à retirer.
* AlgorthymeGloutonAmeliorer() : qui vas être notre algorithme améliorer en combinant l’algorithme glouton a notre ré-optimisation de la solution.

Les résultats obtenus avec cet algorithme sont répertorier dans ce tableau :

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 |  |  |  |