



# Recherche opérationnelle : Genetic Algorithm for Hub Location Problem

F3B502 : Aide à la décision

Amirath ATANDA & Alban GOUGOUA

September 21, 2020

## 1 Introduction

Le problème de l'emplacement de hubs (*Hub Location Problem (HLP)* en anglais) est un problème populaire dans la littérature et un sujet de recherches pour les entreprises présentes dans le domaine des réseaux et de l'aviation.

Le HLP n'est pas non plus insurmontable car il peut être résolu selon le nombre de nœuds utilisé. En effet, en utilisant un programme linéaire (LP) avec *glpk*, nous avons résolu un HLP de 8 nœuds (voir Annexe). Dans ce rapport, il est question de résoudre le HLP pour 30 nœuds. Alors, nous utiliserons l'algorithme génétique (*Genetic Algorithm* en anglais).

Dans la suite de ce rapport, nous présenterons les différentes étapes de la résolution du HLP par le GA.

## 2 Représentation de la solution

Tout d'abord, il convient de donner la plus bonne représentation aux individus de la population. Nous rappelons que l'une des contraintes de notre HLP est celle de la **structure d'arbre** que doit avoir notre graphe. Evidemment, nous optons pour un **codage de Prüfer** car il représente au mieux les structures de graphe en arbres.

Prenons l'exemple du graphe ci-dessous :

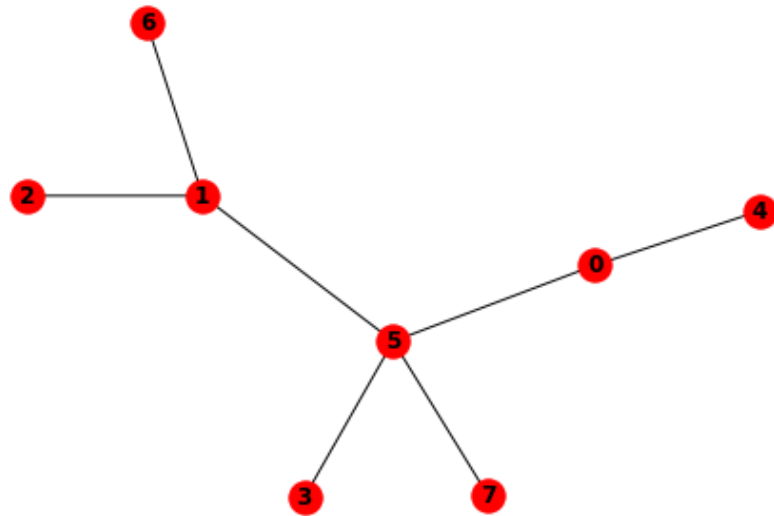


Figure 1: Exemple d'arbre

La séquence de Prüfer de cet arbre est : (1, 5, 0, 5, 1, 5). En effet, le nombre d'éléments dans la séquence de Prüfer est égale au nombre de nœuds du réseau différencié de 2. De plus, les nœuds dans la séquence de Prüfer représente les hubs. Dans ce cas, la séquence de Prüfer compte 6 bits et les hubs sont les nœuds 0, 1 et 5.

Après construction de la séquence de Prüfer, on peut construire l'arbre de la manière suivante : en rangeant les nœuds dans l'ordre croissant des numéros  $\{0, 1, 2, 3, 4, 5, 6, 7\}$  (8 nœuds), on affecte les nœuds ne se trouvant pas dans la séquence (les spokes) en premier au premier nœud de la séquence de Prüfer. Ainsi, on saute dans la liste des nœuds les numéros 0 et 1, et on affecte le nœud 2 au nœud 1. Ensuite, on supprime le premier bit de la séquence de Prüfer et on relie le suivant le nœud 5 au nœud 3, et ainsi de suite. Pour plus d'explications sur la construction d'un arbre à partir d'une séquence de Prüfer, suivre ce [lien](#).

### Code Python 3.1

```
# Créer une séquence de Prüfer
# On travaille sur un arbre de 8 noeuds donc La Longueur de La séquence de Prüfer est de (8-2) = 6

def pruffer() :
    """
    Retourne une séquence de Prüfer : individual (une liste)
    """
    individual = []

    for i in range(NodeNum-2) :
        individual.append(random.randrange(NodeNum))

    random.shuffle(individual)

    return individual
```

Figure 2: Codage de Prüfer pour le GA de HLP : 8 nœuds

```

# Construire un arbre à partir d'une séquence de Prüfer
# Source : FitnessEvaluationHubLocationProblem.py de M. Mehrdad MOHAMMADI

def pruferToTree(a):
    """
    Retourne un arbre : tree

    a : une séquence de Prüfer
    """

    tree = []
    T = range(0, len(a)+2)

    # the degree of each node is how many times it appears in the sequence
    deg = [1]*len(T)
    for i in a: deg[i] += 1

    # for each node Label i in a, find the first node j with degree 1 and add the edge (j, i) to the tree
    for i in a:
        for j in T:
            if deg[j] == 1:
                tree.append((i,j))
                # decrement the degrees of i and j
                deg[i] -= 1
                deg[j] -= 1
                break

    last = [x for x in T if deg[x] == 1]
    tree.append((last[0],last[1]))

    return tree

```

Figure 3: Construction d'arbre à partir d'une séquence de Prüfer

### 3 Représentation de la population

Notre population sera composé d'individus qui sont des séquences de Prüfer. En prenant l'exemple d'un réseau à 8 nœuds, on a  $6^8$  soit plus d'un million de possibilités pour des séquences de Prüfer. Mais, on fera une variation de la taille de la population pour savoir la valeur optimale pour notre problème.

### 4 Evaluation : calcul de la valeur de fitness

L'objectif de notre problème d'emplacement de hubs est de **minimiser les coûts (fixes et variables)** de l'ensemble des transactions effectuées dans le réseau. Cette fonction objective ne diffère pas de celle du programme linéaire ; nous la conservons donc.

Alors, on calcule pour chaque individu de la population le coût total qui correspond à la valeur du fitness. Nous présentons le code Python ci-dessous.

#### Code Python 3.1

```

def individualFitness(individual):
    Tree_Edges = pruferToTree(individual);

    Graph = nx.Graph(Tree_Edges)
    All_Pairs_Path = dict(nx.all_pairs_shortest_path(Graph))

    Hubs = np.unique(individual)
    FixedCost = fixCost[Hubs].sum()

    VarCost = 0
    FlowToHub = np.zeros(NodeNum)
    for i in range(NodeNum):
        for j in range(NodeNum):
            if j > i:
                for x in range(len(All_Pairs_Path[i][j])-1):
                    Route = All_Pairs_Path[i][j]
                    if Route[x] in Hubs and Route[x+1] in Hubs:
                        VarCost = VarCost + alpha * varCost[Route[x],Route[x+1]] * (flow[i,j]+flow[j,i])
                    else:
                        VarCost = VarCost + varCost[Route[x],Route[x+1]] * (flow[i,j]+flow[j,i])

    Fitness = FixedCost + VarCost[0]

    # Calculating the Entering flow to Each Hub #
    for i in range(len(Tree_Edges)):
        if Tree_Edges[i][0] not in Hubs:
            FlowToHub[Tree_Edges[i][1]] = FlowToHub[Tree_Edges[i][1]] + (Origin[Tree_Edges[i][0]]+Destination[Tree_Edges[i][0]])
        elif Tree_Edges[i][1] not in Hubs:
            FlowToHub[Tree_Edges[i][0]] = FlowToHub[Tree_Edges[i][0]] + (Origin[Tree_Edges[i][1]]+Destination[Tree_Edges[i][1]])
    for i in range(len(Hubs)):
        FlowToHub[Hubs[i]] = FlowToHub[Hubs[i]] + (Origin[Hubs[i]]+Destination[Hubs[i]])

    # Feasibility Check: Capacity Constraint #
    Exceed = np.subtract(Cap,FlowToHub)
    ExceedCap = Exceed[Hubs,Hubs]
    if min(ExceedCap) < 0:
        Fitness = np.dot(Fitness,100000000)

    return Fitness

```

Figure 4: Fonction de calcul du fitness

## 5 Sélection

Parmi les différentes méthodes de sélection aléatoire, nous optons pour le [tournoiement selection](#) qui permet de choisir le meilleur de chaque population du tournoi. En effet, cette sélection garantit le caractère aléatoire tout en optimisant le choix : prendre les individus qui ont un meilleur fitness.

### Code Python 3.1

```

# Appliquer Le Tournoiement Selection

def tournamentSelection(generation, Ntour) :
    """
    Retourne une population : populationSelection (une liste de listes)

    generation : une liste de tuples (individu, fitnessValue)
    Ntour : un entier (integer)
    """

    populationSelection = []

    for i in range(len(generation)) :
        populationRandom = []

        for j in range(Ntour) :
            populationRandom.append(generation[random.randrange(len(generation))])

        populationRandom = sorted(populationRandom, key=lambda x : x[1])

        populationSelection.append(populationRandom[0])

    populationSelection = sorted(populationSelection, key=lambda x : x[1])
    a = populationSelection[:Ntour]
    for i in range(len(a)) :
        populationSelection[i] = a[i][0]

    return populationSelection

```

Figure 5: Fonction de Tournament Selection

On remarque le paramètre *Ntour* qui permet de spécifier la taille de la population pour chaque tournoi. Ce paramètre est un paramètre important pour notre solution finale, car en fonction de

sa valeur, on optimise notre résultat.

## 6 Crossover & Mutation & Next Generation

Chaque génération sera composé d'individus qui sont premièrement les élites de leur génération, deuxièmement des croisés (descendants d'individus croisés), troisièmement des mutés (individus ayant subi une mutation). En effet, nous choisissons une stratégie d'élitisme.

- Elites : en passant d'une génération à une autre, on choisit un pourcentage *elitism\_rate* de la génération. Ce taux fait partie aussi des paramètres très importants de notre GA.
- Croisés : de même que précédemment, les générations suivantes seront composées d'un pourcentage *crossed\_rate* d'individus de la génération actuelle croisés. Le taux *crossed\_rate* est lui aussi important dans la détermination de la solution finale.

La fonction de croisement que nous utilisons est le *One-point crossover* : *1X0*.

### Code Python 3.1

```
# Crossover : One-point XO

def crossover1X0(parent1, parent2):
    """
    Retourne deux individus (séquences de Prüfer) : child1 (une liste) et child2 (une liste)

    parent1 : un individu de populationSelection (une liste)
    parent2 : un individu de populationSelection (une liste)
    """

    child1 = []
    child2 = []

    for i in range(len(parent1)):
        if(i < (len(parent1)/2)):
            child1.append(parent1[i])
            child2.append(parent2[i])
        else:
            child1.append(parent2[i])
            child2.append(parent1[i])

    return child1, child2
```

Figure 6: Fonction Crossover1XO

```
# Sélection des croisés
crossed = []
while(len(crossed) < (int(len(populationSelection)*crossover_rate))):
    parent1 = populationSelection[random.randrange(len(populationSelection))]
    parent2 = populationSelection[random.randrange(len(populationSelection))]
    child1, child2 = crossover1X0(parent1=parent1, parent2=parent2)
    crossed.append(child1)
    if(len(crossed) < (len(populationSelection)*crossover_rate)):
        crossed.append(child2)
```

Figure 7: Sélection des croisés

- Mutés : les générations suivantes seront aussi composées d'un pourcentage *mutated\_rate* d'individus de la génération actuelle mutés. Le taux *mutated\_rate* est lui aussi important dans la détermination de la solution finale car il est la différence entre 1 et la somme des taux d'élites et des croisés.

Nous utilisons pour notre problème une mutation simple : *reversion*.

### Code Python 3.1

```

# Mutation : Reversion
def mutationReversion(individual) :
    """
    Retourne un individu (séquence de Prüfer) : individual (une liste)

    individual : un individu de la population
    """

    individual.reverse()

    return individual

```

Figure 8: Fonction de mutation : Reversion

```

# Sélection des mutés
mutated = []
for j in range(int(len(populationSelection)*mutate_rate)) :
    individual = populationSelection[random.randrange(len(populationSelection))]
    mutationReversion(individual=individual)
    mutated.append(individual)

```

Figure 9: Sélection des mutés

## 7 Résultats

Dans cette partie, nous présentons nos résultats pour le HLP avec 8 nœuds en faisant varier les paramètres suivants :

- Taille de la population : le nombre d'individus dans chaque génération
- Taille de la population du tournoi :  $N_{tour}$
- Taux d'élites :  $elitism\_rate$
- Taux des croisés :  $crossed\_rate$

Nous exécuterons le code suivant qui est la fonction de passage d'une génération à une autre. On exécutera 100 itérations de ce code afin d'avoir la solution finale.

### Code Python 3.1

```
#####
#                                     #
#           Genetic Algorithm for HLP           #
#                                     #
#####

def gaHLP(generation, iterations=100, Ntour=3, elitism_rate=0.2, crossover_rate=0.7) :
    """
    Retourne la dernière génération et l'individu qui a le meilleur fitness : generation (liste de tuples)
    et individualBest (un tuple)

    generation : une liste de tuples (individu, fitnessValue)
    iterations : le nombre de générations parcouru avant la dernière descendance (integer)
    Ntour : paramètre de la fonction tournamentSelection (integer)
    elitism_rate : taux des meilleurs pris dans chaque génération (float)
    crossover_rate : taux des individus pris pour croisement (float)
    """

    # Taux des individus pris pour mutation (float)
    mutate_rate = (1 - elitism_rate - crossover_rate)

    for i in range(iterations) :
        # Tournament Selection
        populationSelection = tournamentSelection(generation=generation, Ntour=Ntour)

        # Sélection des élites
        elites = []
        for j in range(int(len(populationSelection)*elitism_rate)) :
            elites.append(populationSelection[j])

        # Sélection des croisés
        crossed = []
        while(len(crossed) < (int(len(populationSelection)*crossover_rate))) :
            parent1 = populationSelection[random.randrange(len(populationSelection))]
            parent2 = populationSelection[random.randrange(len(populationSelection))]
            child1, child2 = crossover1X0(parent1=parent1, parent2=parent2)
            crossed.append(child1)
            if(len(crossed) < (len(populationSelection)*crossover_rate)) :
                crossed.append(child2)

        # Sélection des mutés
        mutated = []
        for j in range(int(len(populationSelection)*mutate_rate)) :
            individual = populationSelection[random.randrange(len(populationSelection))]
            mutationReversion(individual=individual)
            mutated.append(individual)

        # Génération descendance
        population = []
        for individual in elites :
            population.append(individual)
        for individual in crossed :
            population.append(individual)
        for individual in mutated :
            population.append(individual)
        generation = []
        for individual in population :
            generation.append((individual, float(individualFitness(individual))))
        generation = sorted(generation, key=lambda x : x[1])

    individualBest = generation[0]

    return generation, individualBest
```

Figure 10: Fonction du Genetic Algorithm

On obtient le tableau suivant :

| Taille de la population | Ntour | elitism_rate | crossed_rate | Meilleure solution (Coût) |
|-------------------------|-------|--------------|--------------|---------------------------|
| 1000                    | 3     | 0.1          | 0.8          | 6873499.85                |
| 1000                    | 4     | 0.1          | 0.8          | 6482028.75                |
| 1000                    | 5     | 0.1          | 0.8          | 6482028.75                |
| 1000                    | 6     | 0.1          | 0.8          | 6482028.75                |
| 1000                    | 3     | 0.2          | 0.7          | 6482028.75                |
| 1000                    | 4     | 0.2          | 0.7          | 6482028.75                |
| 1000                    | 5     | 0.2          | 0.7          | 6482028.75                |
| 1000                    | 6     | 0.2          | 0.7          | 6482028.75                |

Table 1: Résultats pour 1000 individus

En fixant, la taille de la population à 1000 individus, on obtient la même solution optimale que celle du programme linéaire (voir Annexe) pour de faibles taux d'élites (0,1 ou 0,2) et de forts taux de croisés (0,7 ou 0,8) à partir de  $Ntour \geq 4$ . Afin de valider notre hypothèse sur le paramètre

de la taille de population, on essaie avec 500 individus par population (voir Table 2).

| Taille de la population | Ntour | elitism_rate | crossed_rate | Meilleure solution (Coût) |
|-------------------------|-------|--------------|--------------|---------------------------|
| 500                     | 3     | 0.1          | 0.8          | 7297888.75                |
| 500                     | 4     | 0.1          | 0.8          | 6873499.85                |
| 500                     | 5     | 0.1          | 0.8          | 6873499.85                |
| 500                     | 6     | 0.1          | 0.8          | 7419917                   |
| 500                     | 3     | 0.2          | 0.7          | 7170961.95                |
| 500                     | 4     | 0.2          | 0.7          | 8064516.8                 |
| 500                     | 5     | 0.2          | 0.7          | <b>6482028.75</b>         |
| 500                     | 6     | 0.2          | 0.7          | 7297184.65                |

Table 2: Résultats pour 500 individus

Une seule ligne de la table ci-dessus fournit la solution optimale.

Enfin, on applique cet algorithme pour un réseau de 30 noeuds, avec les paramètres optimaux de la Table 3, on obtient :

| Taille de la population | Ntour | elitism_rate | crossed_rate | Meilleure solution (Coût) |
|-------------------------|-------|--------------|--------------|---------------------------|
| 1000                    | 5     | 0.1          | 0.8          | <b>293967059.9</b>        |

Table 3: Solution optimale pour un réseau de 30 nœuds

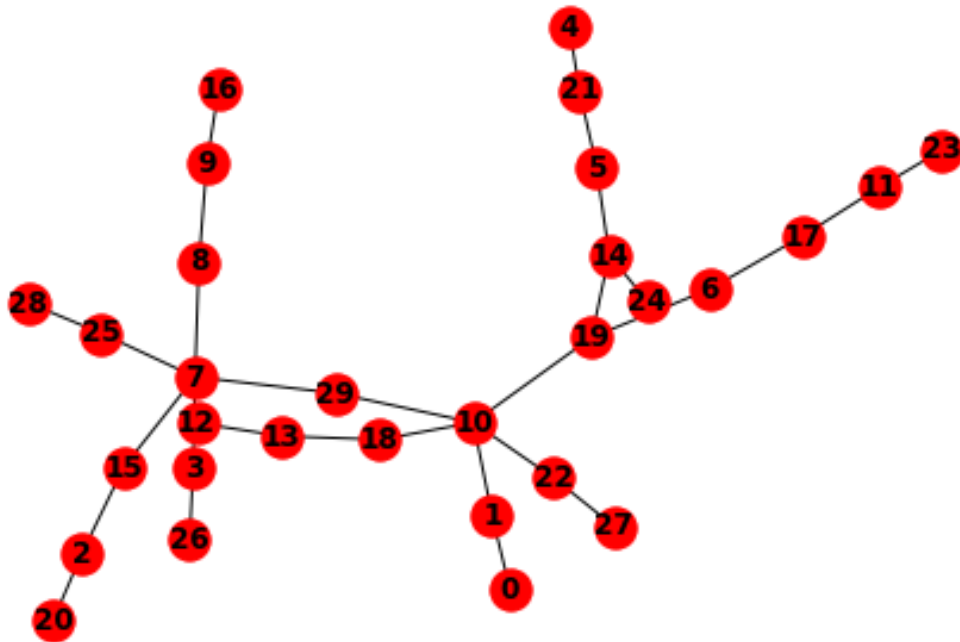


Figure 11: Réseau de hubs pour 30 nœuds

## 8 Conclusion

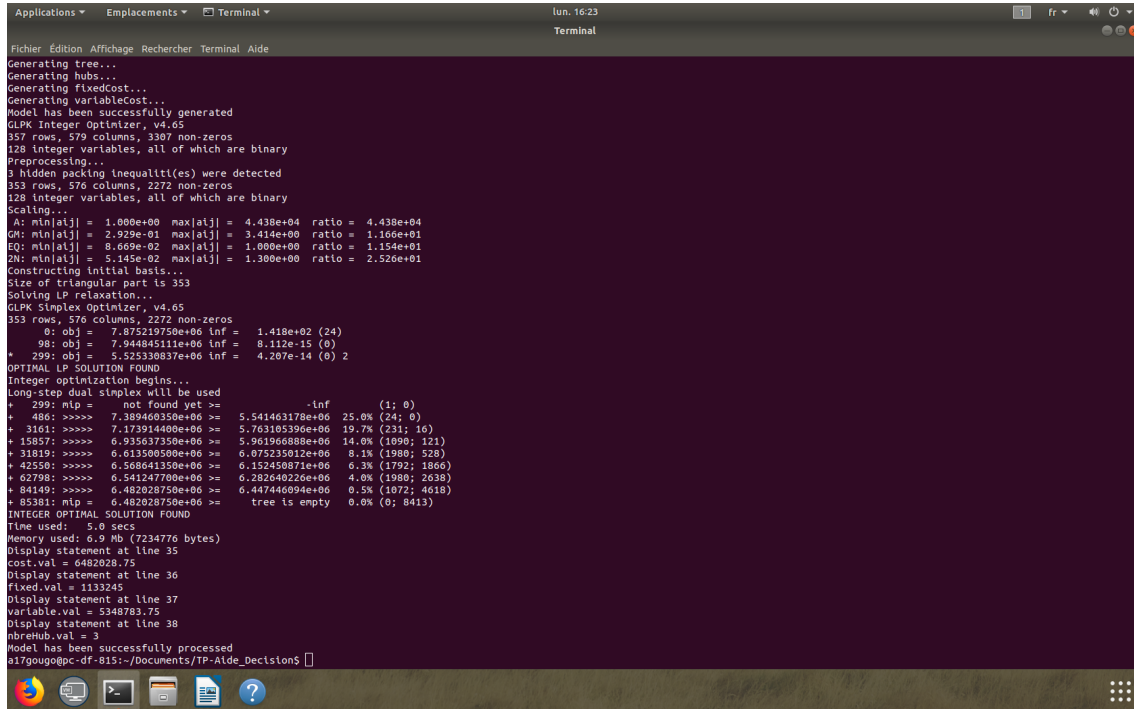
L'algorithme génétique (GA) permet de résoudre des problèmes complexes telles que celui de l'emplacement de hubs (HLP). Cependant, il faut choisir les paramètres optimaux tels que la taille de la population (minimum 1000 individus), la taille de la population de tournoi pour la sélection ( $N_{tour} \geq 4$ ) et respectivement un faible (resp. fort) taux d'élites (resp. de croisés) ; ces



paramètres optimaux garantissent de meilleurs résultats (voir Table 1).

## 9 Annexe

La figure ci-dessous présente le résultat optimal obtenu à partir du programme linéaire pour un réseau de 8 nœuds.



```
Applications Emplacements Terminal
Terminal
Fichier Edition Affichage Rechercher Terminal Aide
Generating tree...
Generating hubs...
Generating fixedCost...
Generating variableCost...
Model has been successfully generated
GLPK Integer Optimizer, v4.65
357 rows, 570 columns, 3307 non-zeros
128 integer variables, all of which are binary
Preprocessing...
3 hidden packing inequality(s) were detected
353 rows, 576 columns, 2272 non-zeros
128 integer variables, all of which are binary
Scaling...
A: min|a[ij]| = 1.000e+00 max|a[ij]| = 4.438e+04 ratio = 4.438e+04
G1: min|a[ij]| = 2.929e-01 max|a[ij]| = 3.414e+00 ratio = 1.166e+01
EQ: min|a[ij]| = 8.669e-02 max|a[ij]| = 1.000e+00 ratio = 1.154e+01
Z1: min|a[ij]| = 5.145e-02 max|a[ij]| = 1.300e+00 ratio = 2.526e+01
Constructing initial basis...
Size of triangular part is 353
Solving LP relaxation...
GLPK Simplex Optimizer, v4.65
353 rows, 576 columns, 2272 non-zeros
* 0: obj = 7.875219750e+06 inf = 1.418e+02 (24)
* 98: obj = 7.94484511e+06 inf = 8.112e-15 (0)
* 299: obj = 5.525330837e+06 inf = 4.207e-14 (0) 2
OPTIMAL LP SOLUTION FOUND
Integer optimization begins...
Long-step dual simplex will be used
* 299: mip = not found yet >= -inf (1; 0)
* 486: >>>> 7.389460350e+06 >= 5.541463178e+06 25.0% (24; 0)
* 3161: >>>> 7.173914400e+06 >= 5.763105396e+06 19.7% (231; 16)
* 15837: >>>> 6.935637350e+06 >= 5.961960880e+06 14.0% (1090; 121)
* 31819: >>>> 6.613500500e+06 >= 6.075235012e+06 8.1% (1980; 528)
* 42550: >>>> 6.568641350e+06 >= 6.152450871e+06 6.3% (1792; 1866)
* 62798: >>>> 6.541247700e+06 >= 6.282640226e+06 4.0% (1980; 2638)
* 84149: >>>> 6.482028750e+06 >= 6.447446094e+06 0.5% (1072; 4616)
* 85381: mip = 6.482028750e+06 >= tree is empty 0.0% (0; 8413)
INTEGER OPTIMAL SOLUTION FOUND
Time used: 5.0 secs
Memory used: 6.9 Mb (7234776 bytes)
Display statement at line 35
cost.val = 6482028.75
Display statement at line 36
fixed.val = 1133245
Display statement at line 37
variable.val = 5348783.75
Display statement at line 38
nbreHub.val = 3
Model has been successfully processed
a17gougo@pc-df-815:~/Documents/TP-Alde_Decision$
```

Figure 12: Résultat du HLP pour 8 nœuds à partir de GLPK