

Institut polytechnique de paris
École nationale supérieure de techniques avancées

Rapport du projet RO203 : Loopy

Réalisé par :

Mahdi Cheikhrouhou

Marwen Bahri

2ème année Techniques Avancées

02 mai 2021

Table des matières

1	Description du jeu	3
2	Raisonnement pour la résolution du problème	5
2.1	Premier sous-problème	5
2.2	Deuxième sous-problème	5
2.3	Troisième sous-problème	6
2.4	Quatrième sous-problème	7
3	Programme linéaire	9
3.1	Premier sous-problème	9
3.2	Deuxième sous-problème	10
3.3	Troisième sous-problème	11
3.4	Quatrième sous-problème	13
4	Résolution heuristique	14
4.1	Reformulation du problème	14
4.2	Méthode de résolution	15
4.2.1	Première idée	16
4.2.2	Deuxième idée	16
4.2.3	Calcul du nombre de composantes connexes	17
5	Les fonctions principales du code :	18
5.1	Fonctions d'entrée et sortie	18
5.1.1	Fonctions d'entrée	18
5.1.2	Fonctions de sorties	19
5.2	Fonction de génération	21
5.3	Fonction de Résolution	21
5.3.1	Résolution avec la méthode SIMPLEX	21
5.3.2	Résolution avec la méthode heuristique	21
5.3.3	Résolution générale	21

Table des figures

1.1	Exemple d'une grille de problème donnée	3
1.2	Exemple de grille résolue	4
2.1	Graphe avec 2 boucles simples	5
2.2	Graphe initial	6
2.3	Graphe après les changements : 1 source - 7 puits	7
2.4	Les possibilités d'une case de valeur 3	8
3.1	Illustration des variables <i>edgesV</i> et <i>edgesH</i>	9
3.2	Les arcs adjacents d'une case	10
3.3	Les adjacents d'un noeud	11
3.4	Illustration des variables du flot	13
4.1	Les composantes connexes de la graphe	14
4.2	Schéma pour la preuve de P1	15
4.3	Différence avec la vérification du nombre de composantes connexes . .	16
5.1	Exemple de grille initiale dans un fichier txt	18
5.2	Sortie de la fonction DisplayGrid	19
5.3	Résultat de la fonction displaySolution	20
5.4	Diagramme de performance du méthode simplex	20

Chapitre 1

Description du jeu

Le jeu consiste en une grille de taille $n * n$ avec des cases vides et d'autres avec des nombres entre 0 et 3 comme le montre l'exemple suivant :

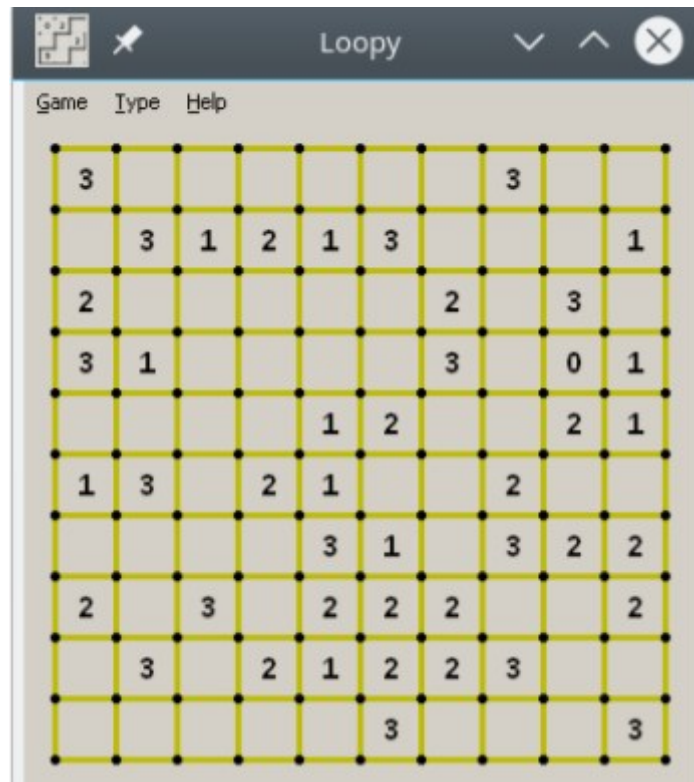


FIGURE 1.1 – Exemple d'une grille de problème donnée

Le but du jeu est de trouver **une boucle** qui doit passer par x côtés d'une carré avec le nombre x . La résolution de la grille donnée en exemple est la suivante :

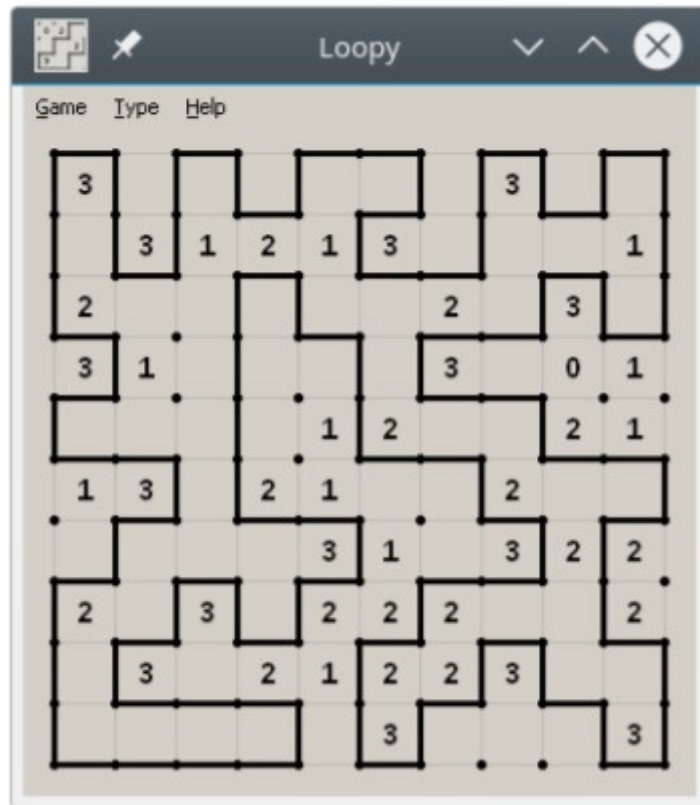


FIGURE 1.2 – Exemple de grille résolue

Chapitre 2

Raisonnement pour la résolution du problème

Le problème peut être décomposé en sous-problèmes plus ou moins simples. Dans ce chapitre, nous allons indiquer les idées. Dans le chapitre suivant, nous décrirons les contraintes du programme linéaire.

2.1 Premier sous-problème

Le premier sous-problème est la condition du problème : Exactement x côtés d'une case avec le nombre x doivent être inclus dans la boucle.

2.2 Deuxième sous-problème

Nous considérons les points que nous allons relier comme un graphe et les côtés sont des liens entre les nœuds. Pour s'assurer que nous avons une boucle simple (pas de croisement), il est nécessaire que le degré de chaque nœud soit égal à 2. C'est effectivement une partie de la solution mais ce n'est pas suffisant car nous pouvons trouver 2 ou plusieurs boucles simples dans le même graphe qui vérifient cette condition comme le montre la figure suivante. Ce problème sera résolu dans le sous-problème suivant..

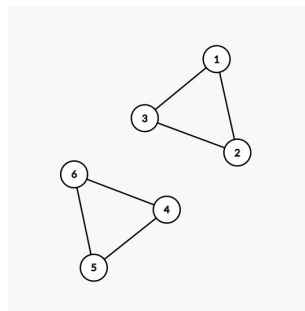


FIGURE 2.1 – Graphe avec 2 boucles simples

2.3 Troisième sous-problème

Pour résoudre le problème de plusieurs boucles, nous allons d'abord commencer par la théorie.

Considérons un graphe non orienté de n nœuds . Pour assurer la connexité du graphe, nous pouvons appliquer l'algorithme *DFS* ou *BFS* à partir de n'importe quel nœud du graphe. Nous concluons que le graphe est connexe si et seulement si le nombre de noeuds visités est égal au nombre de noeuds au total. Mais ces algorithmes ne peuvent pas être codés dans un programme linéaire en tant que contraintes, c'est pourquoi nous avons recours à l'utilisation de l'algorithme de *flot maximum* . Nous allons changer le graphe pour obtenir le résultat. Nous considérons n'importe quel nœud comme source et nous remplaçons les arcs par des arcs orientés avec une capacité infinie. Puis nous connectons tous les nœuds avec un autre nœud artificiel qui est le puits avec une capacité de 1. Nous appliquons maintenant l'algorithme du flot maximal et comparons le résultat à n .Le graphe est connexe si et seulement si $MaxFlow == n$. Le changement du graphe est représenté dans les figures suivantes :

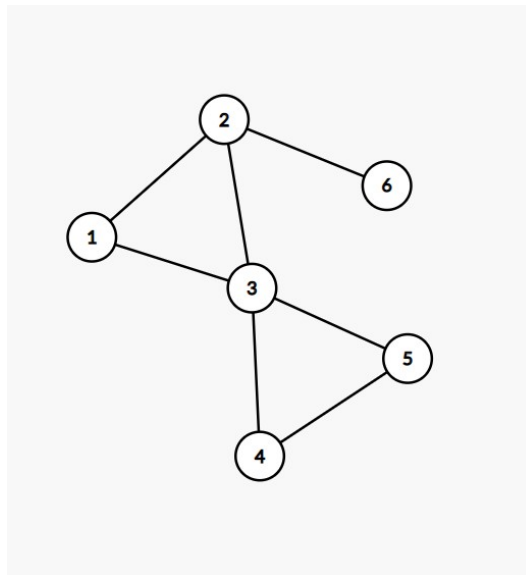


FIGURE 2.2 – Graphe initial

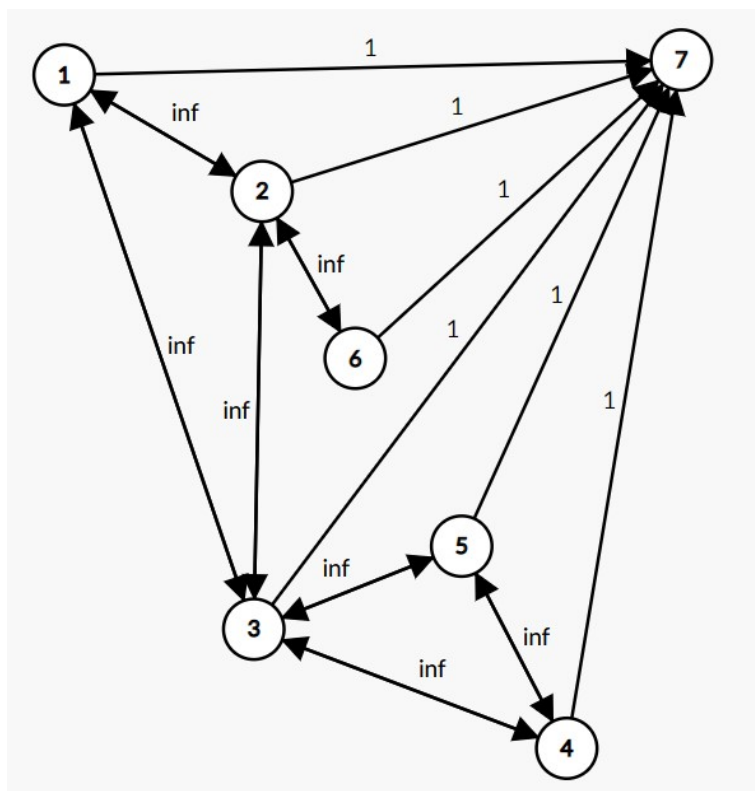


FIGURE 2.3 – Graphe après les changements : 1 source - 7 puits

Nous retournons maintenant pour notre problème. Pour que la solution soit correcte, nous appliquons l'algorithme du flot maximal et il est nécessaire que le flot maximal trouvé soit égal au nombre de nœuds qui se trouvent dans la boucle. C'est à dire le flot maximum égale au nombre des noeuds dont le degré est égal à 2. Mais nous n'avons pas encore fini , il nous reste encore un détail. Dans ce qu'on a expliqué, nous avons considéré un noeud quelconque comme le source. Mais on ne sait pas quels sont les nœuds à prendre dans notre jeu. D'où la création du dernier sous-problème.

2.4 Quatrième sous-problème

Dans ce dernier sous-problème, nous discuterons d'une solution pour le choix de la source. Il y a 2 cas :

- Il y a une case dont la valeur est 3 : Alors nécessairement tous ses coins sont dans le cycle et par conséquent nous prenons n'importe quel point comme source comme représenté dans la figure suivante.
- Sinon : On choisit une case dont la valeur n'est pas nulle. On applique l'algorithme 4 fois. A chaque fois on choisit un coin de la case choisie comme source. Si les 4 ne donnent pas de solution alors la solution n'existe pas.



FIGURE 2.4 – Les possibilités d’une case de valeur 3

Chapitre 3

Programme linéaire

Dans cette partie, nous allons transformer les sous-problèmes définis en un problème linéaire. Nous supposons que la taille de la grille est de $n * n$.

3.1 Premier sous-problème

Nous définissons d'abord les arcs comme deux matrices, une pour les arcs horizontaux et une autre pour les arcs verticaux.

$$edgesH_{i,j} \in \{0, 1\}, 1 \leq i \leq n + 1, 1 \leq j \leq n$$

$$edgesV_{i,j} \in \{0, 1\}, 1 \leq i \leq n, 1 \leq j \leq n + 1$$

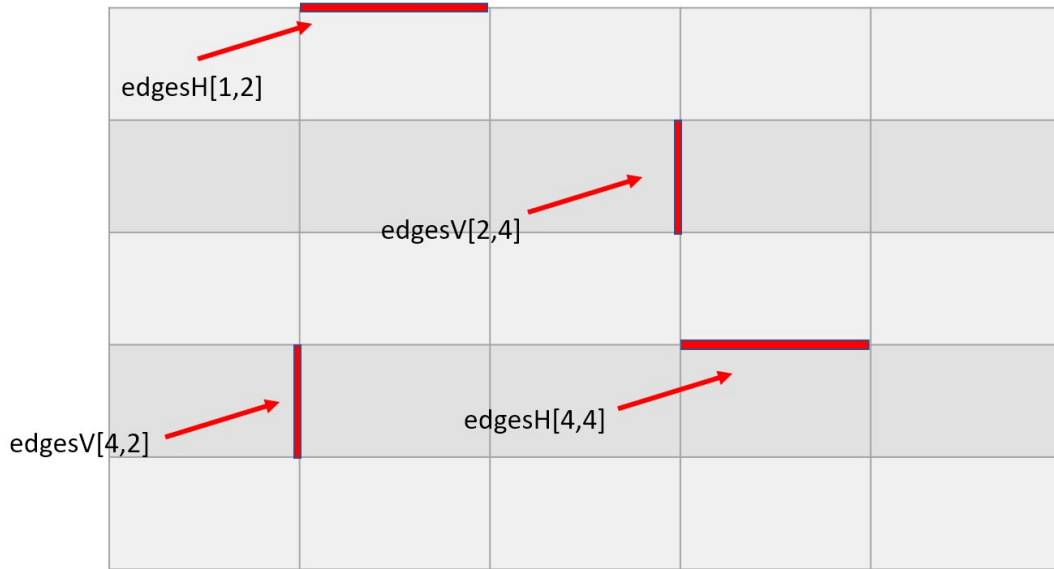


FIGURE 3.1 – Illustration des variables $edgesV$ et $edgesH$

Maintenant, pour la condition , nous bouclons sur les cellules non vides et ajoutons la contrainte suivante : supposons dans la case (i, j) il existe la valeur $grid_{i,j}$.

Les indices dans l'équation sont expliqués dans la figure suivante.

$$edgesH_{i,j} + edgesH_{i+1,j} + edgesV_{i,j} + edgesV_{i,j+1} - grid_{i,j} = 0$$

$$1 \leq i \leq n, 1 \leq j \leq n$$

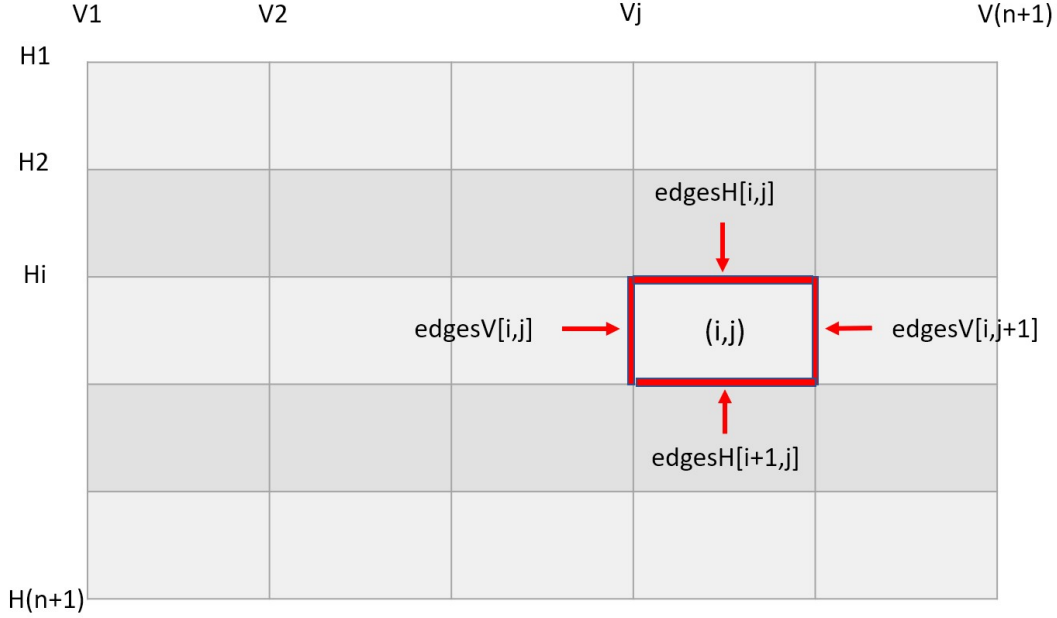


FIGURE 3.2 – Les arcs adjacents d'une case

3.2 Deuxième sous-problème

Dans cette partie nous allons développer les contraintes liées aux boucles. Ces contraintes garantissent que tous les points de degré non nul se trouvent dans une boucle. Pour ce faire, nous définissons d'abord une variable entière *degree* qui représente le degré d'un nœud qui est en fait la somme des arcs adjacents. L'équation est illustrée dans la figure suivante (il y a des cas particuliers au bord de la grille qu'on a tenu en compte seulement dans le code, on va discuter ici seulement le cas générale) :

$$edgesV_{i-1,j} + edgesV_{i,j} + edgesH_{i,j-1} + edgesH_{i,j} - degree_{i,j} = 0$$

$$2 \leq i \leq n, 2 \leq j \leq n$$

Une fois que nous avons calculé le degré, nous créons une variable binaire *degreeCond* qui sera utilisée pour la condition sur le degré comme suit :

$$degree_{i,j} - 2 * degreeCond_{i,j} = 0; 1 \leq i, j \leq n + 1$$

Cette équation nous donne que le degré doit être soit 0 (*degreeCond* = 0) c'est à dire pas d'adjacents donc il n'existe dans aucune boucle, soit 2 (*degreeCond* = 1) et c'est la condition qu'on a donnée au début.

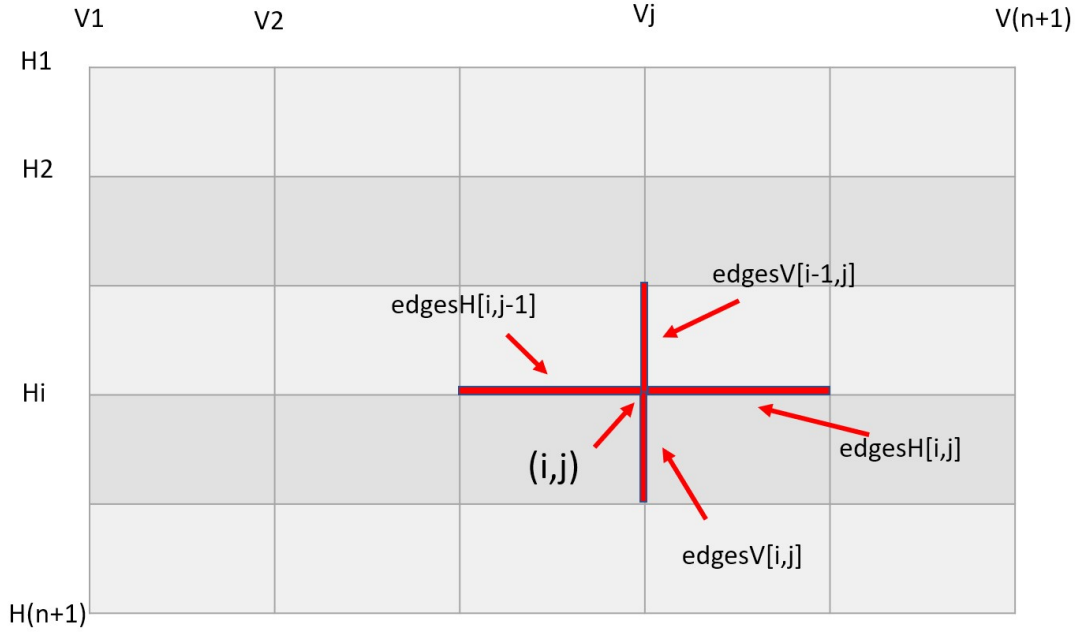


FIGURE 3.3 – Les adjacents d'un noeud

3.3 Troisième sous-problème

C'est le sous problème le plus complexe dans le programme. Dans cette partie nous allons développer la partie flot à partir d'une source vers un puits imaginaire. Pour se faire, nous définissons d'abord quelques variables :

$$flowH_{i,j,k} \in \mathcal{N}; 1 \leq i \leq n+1, 1 \leq j \leq n, 1 \leq k \leq 2$$

$$flowV_{i,j,k} \in \mathcal{N}; 1 \leq i \leq n, 1 \leq j \leq n+1, 1 \leq k \leq 2$$

$$flowSink_{i,j} \in \{0, 1\}; 1 \leq i \leq n+1, 1 \leq j \leq n+1$$

$$capH_{i,j} \in \mathcal{N}; 1 \leq i \leq n+1, 1 \leq j \leq n$$

$$capV_{i,j} \in \mathcal{N}; 1 \leq i \leq n, 1 \leq j \leq n+1$$

$$phi \in \mathcal{N}$$

Description des variables :

- $flowH_{i,j,1}$ correspond au flot qui passe par l'arc $edgeH_{i,j}$ de gauche à droite c'est à dire : de (i, j) à $(i, j+1)$
- $flowH_{i,j,2}$ correspond au flot qui passe par l'arc $edgeH_{i,j}$ de droite à gauche c'est à dire : de (i, j) à $(i, j-1)$
- $flowV_{i,j,1}$ correspond au flot qui passe par l'arc $edgeV_{i,j}$ de haut vers bas c'est à dire : de (i, j) à $(i+1, j)$

- $flowV_{i,j,2}$ correspond au flot qui passe par l'arc $edgeV_{i,j}$ de bas vers haut c'est à dire : de (i, j) à $(i - 1, j)$
- $flowSink_{i,j}$ correspond au flot qui passe du noeud (i, j) vers le puits virtuelle. c'est une variable binaire car le flot maximum qu'on peut envoyer vers le puits égale à 1 à partir d'un seul noeud.
- $capH_{i,j}$ correspond à la capacité de l'arc $edgeH_{i,j}$
- $capV_{i,j}$ correspond à la capacité de l'arc $edgeV_{i,j}$
- Phi correspond au flot dans le puits.

Posons maintenant les coordonnées de la source sont : x_s, y_s

Posons alors notre objective qui est :

$$max(phi)$$

car c'est un problème de calcul du flot maximum. On donne les constraints liées au flot , les contraintes sont illustrées dans la figure suivante :

- **Flot inférieure à la capacité :**

$$flowH_{i,j,k} \leq capH_{i,j}, 1 \leq i \leq n + 1, 1 \leq j \leq n, 1 \leq k \leq 2$$

$$flowV_{i,j,k} \leq capV_{i,j}, 1 \leq i \leq n, 1 \leq j \leq n + 1, 1 \leq k \leq 2$$

- **Flot entrant égale au flot sortant :** on va donner comme le cas de degré , seulement le cas générale , les cas particuliers de bords sont implémentés dans le code directement.

Puisque l'équation est un peu longue on va juste créer des quantités ici pour la clarté des choses :

Pour les cases différentes de la source :

$$flotEntrant = flowV_{i-1,j,1} + flowV_{i,j,2} + flowH_{i,j-1,1} + flowH_{i,j,2}$$

$$flotSortant = flowV_{i-1,j,2} + flowV_{i,j,1} + flowH_{i,j-1,2} + flowH_{i,j,1} + flowSink_{i,j}$$

$$flotEntrant - flotSortant == 0$$

$$2 \leq i, j \leq n, (i, j) \neq (x_s, y_s)$$

Pour la source (dans le cas générale : pas dans le bord) :

$$flotEntrant = phi$$

$$flotSortant = flowV_{x_s-1,y_s,2} + flowV_{x_s,y_s,1} + flowH_{x_s,y_s-1,2} + flowH_{x_s,y_s,1} + flowSink_{x_s,y_s}$$

$$flotEntrant - flotSortant == 0$$

Pour le puits :

$$phi - \sum_{1 \leq i,j \leq n+1} flowSink_{i,j} = 0,$$

- **Condition sur les capacités :** pour les arcs dont leurs valeurs sont nulles , il faut que les capacités soient nulles aussi. Pour cela on ajoute une autre contrainte :

$$capH_{i,j} - 10000 * edgesH_{i,j} \leq 0, 1 \leq i \leq n + 1, 1 \leq j \leq n$$

$$capV_{i,j} - 10000 * edgesV_{i,j} \leq 0, 1 \leq i \leq n, 1 \leq j \leq n + 1$$

Cette condition implique que si l'arc existe alors la capacité est quelconque car $cap \leq 10000$ sinon , alors $cap \leq 0$ implique $cap = 0$ d'où capacité nulle.

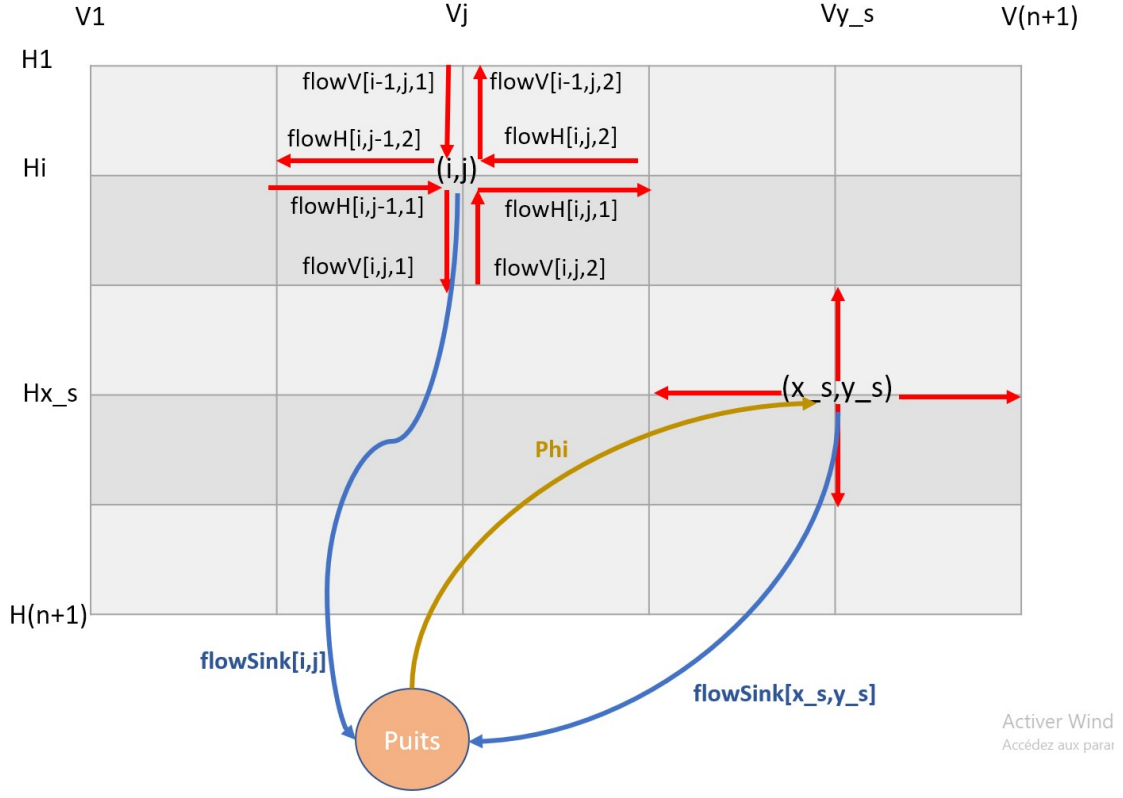


FIGURE 3.4 – Illustration des variables du flot

Maintenant nous ajoutons la condition pour laquelle le flot est égal au nombre de nœuds dont le degré est égal à 2 :

$$phi - \sum_{1 \leq i,j \leq n+1} degreeCond_{i,j} = 0$$

3.4 Quatrième sous-problème

Pour le quatrième sous-problème , il n'y a pas de contraintes supplémentaires. Il suffit de chercher les points possibles d'une source selon les conditions qu'on a cité auparavant et d'appliquer l'algorithme pour chaque source. Le nombre maximum des sources possibles est 4.

Chapitre 4

Résolution heuristique

Pour la résolution heuristique c'était très difficile de trouver une solution de complexité plus ou moins faible qui donnent une solution optimale , la condition du boucle est très difficile à valider en utilisant des algorithmes gloutons ou aléatoires. C'est pourquoi on va reformuler le problème d'une autre façon qui sera le clé d'une solution heuristique plus ou moins acceptable.

4.1 Reformulation du problème

Le problème de trouver une boucle entre les bords des cases peut se transformer en un problème sur les cases : Trouver deux composantes connexes de tel sorte que l'espace extérieur appartient à la graphe . Le schéma suivant explique de plus cette paragraphe :

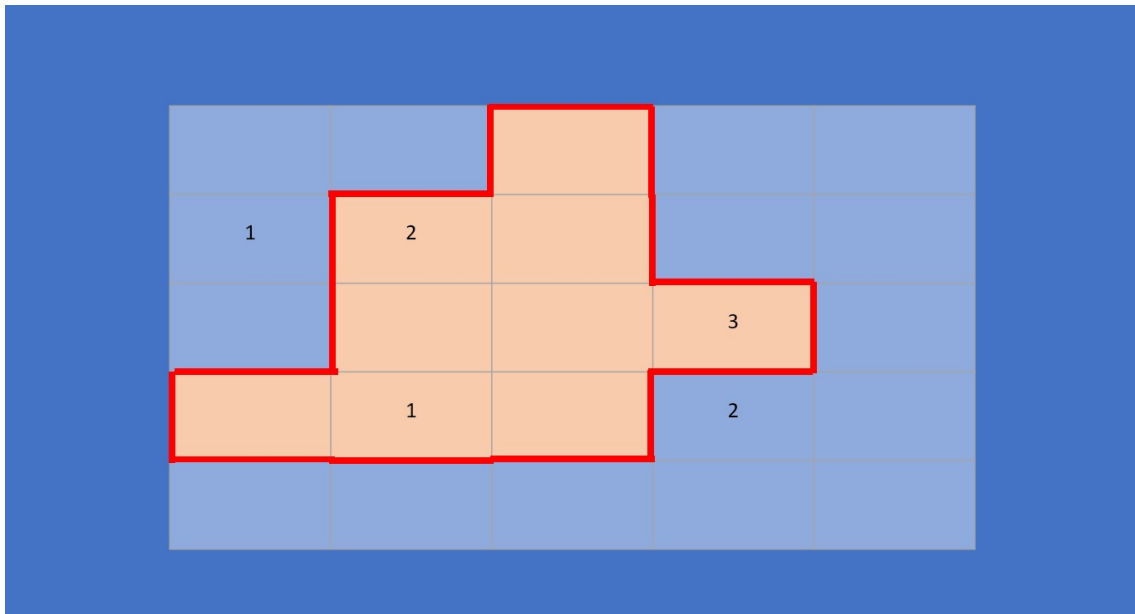


FIGURE 4.1 – Les composantes connexes de la graphe

Dans cette figure on considère la partie entourée par la boucle comme une composante connexe intérieure et tous les autres cases sont dans la même composante

connexe extérieure.

- Partie orangé : composante connexe intérieure
- Partie bleu clair : Les cases de la composante connexe extérieure
- Partie bleu foncé : la partie extérieure (n'existe par réellement dans la graphe)

Maintenant, notre problème c'est de trouver une décomposition de la grille en 2 composantes connexes qui vérifie la condition de chaque case (la boucle doit passer par x cotés d'une case comportant le chiffre x). Mais cette condition maintenant n'a pas de sens car on ne cherche pas une boucle. On peut donc reformuler cette condition aussi : **(P1) Deux cases adjacentes sont dans la même composante connexe si et seulement si il n'y a pas de bord entre les deux c'est à dire le bord qui les sépare n'appartient pas à la boucle du problème initiale.** Par suite la condition sera : chaque case comportant le chiffre x doit avoir $4 - x$ cases adjacents qui sont dans la même composante connexe.

Preuve de (P1) : Supposons que le bord entre A et B existe dans la boucle (représente par l'arc rouge). On va démontrer par l'absurde que A et B sont dans des composantes connexes différentes.

Supposons que A et B sont dans la même composante connexe, alors il existe un chemin dans la grille (vert) de cases connectées. On voit dans le schéma que la boucle va nécessairement couper le chemin vert (orangé ou noir) entre A et B et par suite il ne sont pas connectés par n'importe quel chemin. Absurde!!

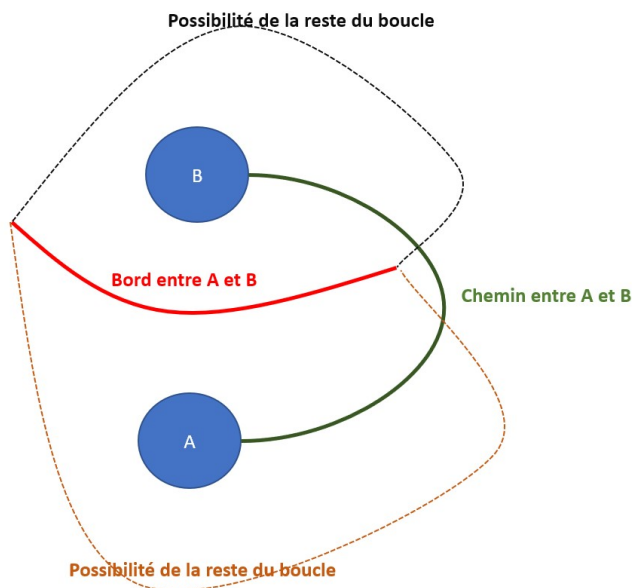


FIGURE 4.2 – Schéma pour la preuve de P1

4.2 Méthode de résolution

Maintenant on a bien reformuler le problème et ses conditions. On va expliquer comment trouver une solution. La solution ne va pas garantir une solution optimale et peut ne pas trouver de solution (complexité élevée).

4.2.1 Première idée

La première idée que nous avons pensé à faire c'est d'essayer toutes les possibilités. On assigne à chaque case soit 0, soit 1 ce chiffre représente la composante connexe à laquelle appartient la case. Mais cette solution comme ça a une complexité de $S_0 = 2^{n*n}$ (cette complexité concerne seulement le nombre de possibilité car la complexité de la validité du graphe générée ne dépend pas du nombre de possibilités).

4.2.2 Deuxième idée

Maintenant on veut diminuer le nombre de possibilités. On ajoute une condition qui n'a pas de sens mathématique mais en regardant plusieurs exemples elle reste valide. La condition : le nombre de case dans la composante extérieure ne doit pas passer le 1/3 du nombre de cases globales. la complexité est maintenant $S_1 = \sum_{k=0}^{(n*n)/3} C_{n*n}^k$. La valeur pour différents n :

- $n = 4$: $S_0 = 65536$, $S_1 = 2516$, $S_0/S_1 = 26$: facteur de 26 d'amélioration
- $n = 5$: $S_0 = 33554432$, $S_1 = 726205$, $S_0/S_1 = 46$: facteur de 46 d'amélioration
- $n = 6$: $S_0 = 68719476736$, $S_1 = 990134947$, $S_0/S_1 = 69$: facteur de 69 d'amélioration

Mais cette méthode on faite génère des graphes où il y a plusieurs composantes connexes . c'est à dire il se peut que deux cases sont supposé dans la même composantes mais il n'a pas de chemin entre eux. Pour éliminer ces cas, à chaque fois on fait une modification on calcule le nombre des composante connexes, si ce dernier est > 2 alors il n'y a pas de solution donc c'est inutile de continuer à développer cette possibilité. On n'a pas aboutit à calculer la complexité exacte de cet algorithme mais un exemple de résolution peut nous donner une idée sur l'importance de ce changement. Cette méthode est appelé *Backtracking with intelligent pruning*.

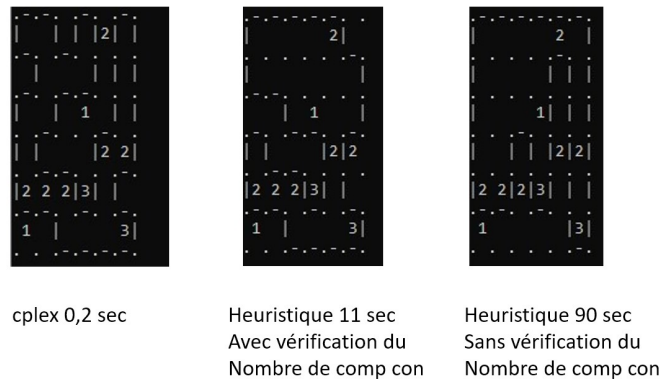


FIGURE 4.3 – Différence avec la vérification du nombre de composantes connexes

4.2.3 Calcul du nombre de composantes connexes

Pour le calcul du nombre des composantes connexes, on a utilisé l'algorithme de *Disjoint Set Union* (DSU) ou *Union Find*.

Description de l'algorithme :

- Chaque noeud constitue un ensemble qui est représenté par lui même
- On boucle sur tous les arcs (a, b) et on unissent l'ensemble qui contient a avec l'ensemble qui contient b .
- Le nombre de composantes connexes est le nombre des ensembles à la fin

Pour la composante extérieure est représentée par la case $(n + 1, n + 1)$ avec n est la taille de la grille.

Chapitre 5

Les fonctions principales du code :

5.1 Fonctions d'entrée et sortie

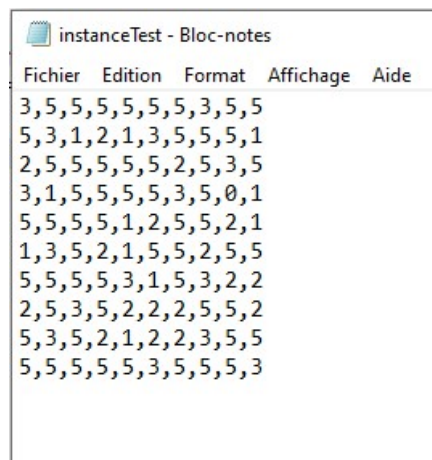
5.1.1 Fonctions d'entrée

On a une seule fonction d'entrée.

readInputFile

Cette fonction prend comme argument une chaîne de caractères qui représente le chemin du fichier où on a généré la grille initiale et retourne un tableau de 2 dimension dont les valeurs sont celles du fichier.

La grille initiale est mémorisé dans un fichier texte (terminaison .txt) dans le répertoire *data* tel est l'exemple de la figure suivante (cette figure représente le jeu dans la figure 1.1).Les cases de la grille sont séparées par des virgules (",") et les cases vides sont représentées par le nombre 5 car c'est impossible de trouver 5 dans la grille et c'est afin de faciliter la récupération des données.



```
instanceTest - Bloc-notes
Fichier  Edition  Format  Affichage  Aide
3,5,5,5,5,5,5,3,5,5
5,3,1,2,1,3,5,5,5,1
2,5,5,5,5,5,2,5,3,5
3,1,5,5,5,5,3,5,0,1
5,5,5,5,1,2,5,5,2,1
1,3,5,2,1,5,5,2,5,5
5,5,5,5,3,1,5,3,2,2
2,5,3,5,2,2,2,5,5,2
5,3,5,2,1,2,2,3,5,5
5,5,5,5,5,3,5,5,5,3
```

FIGURE 5.1 – Exemple de grille initiale dans un fichier txt

5.1.2 Fonctions de sorties

DisplayGrid

Cette fonction à pour but d'afficher une grille non résolu dans le console en lui donnant une chaîne qui représente le chemin du fichier. L'affichage est donnée par la figure suivante.

```
julia> displayGrid("../data/instanceTest.txt")
. . . . .
3 . . . . . 3
. . . . .
. 3 1 2 1 3 . 1
. . . . .
2 . . . . . 2 3
3 1 . . . . 3 0 1
. . . . .
. . . . . 1 2 . 2 1
. . . . .
1 3 . 2 1 . 2
. . . . .
. . . . . 3 1 . 3 2 2
. . . . .
2 . 3 . 2 2 2 . 2
. . . . .
. 3 . 2 1 2 2 3
. . . . .
. . . . . 3 . . 3
```

FIGURE 5.2 – Sortie de la fonction DisplayGrid

displaySolution

Cette fonction à pour but d'afficher une grille résolu et qui est mémoriser dans un fichier txt mais différent que celle de la grille initiale. Ces fichier sont stockés dans des sous répertoires du répertoire *res*. La forme du fichier est la suivante. L'image suivante nous montrent le résultat obtenu :

- Pour les lignes impaires (1,3,5..) il y a n valeurs (n est la taille de la grille) et qui représentent les bords horizontaux. Si la valeur = 1 alors le bord est dans le cycle donc on l'affiche sinon on n'affiche rien.
- Pour les lignes paires (2,4,6,..) il y a $n + (n + 1) = 2 * n + 1$ valeurs.
 - Pour les valeurs dans les colonnes impaires (1,3,5,..) représentent les bords verticaux. Si la valeur = 1 alors le bord est dans le cycle donc on l'affiche sinon on n'affiche rien.
 - Pour les valeurs dans les colonnes paires (2,4,6,..) représentent les valeurs de la grille initiale. Si la valeur = 5 alors la case est vide sinon on affiche la valeur.

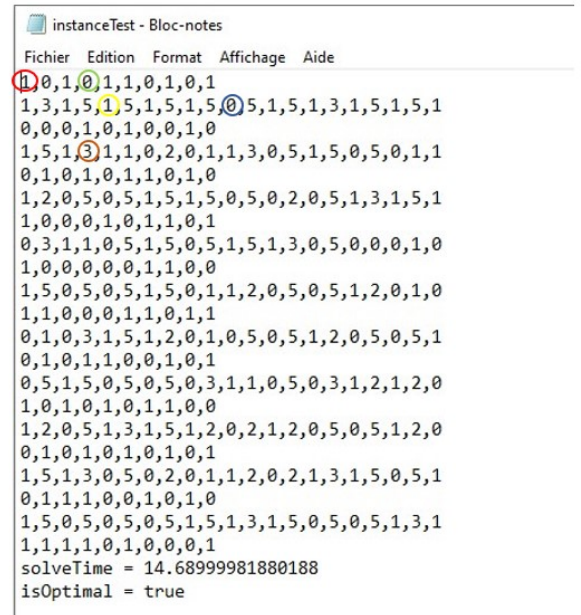
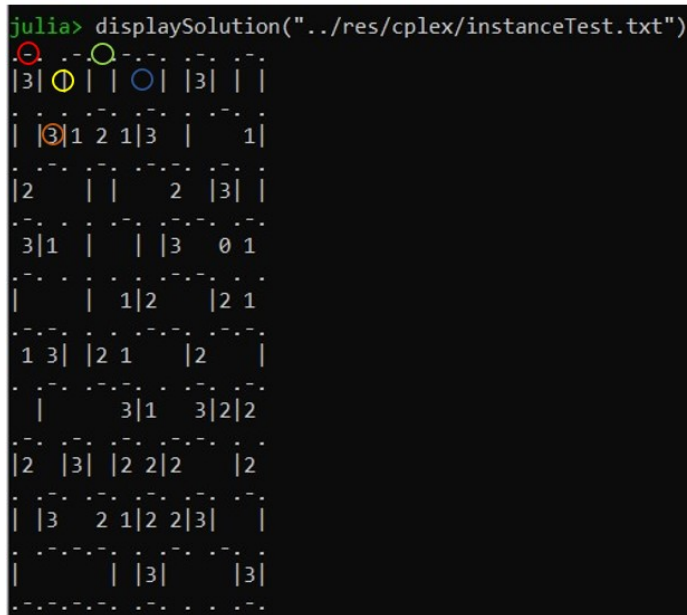


FIGURE 5.3 – Résultat de la fonction displaySolution

performanceDiagram

Le diagramme de performance des méthode simplex et heuristique est le suivant :

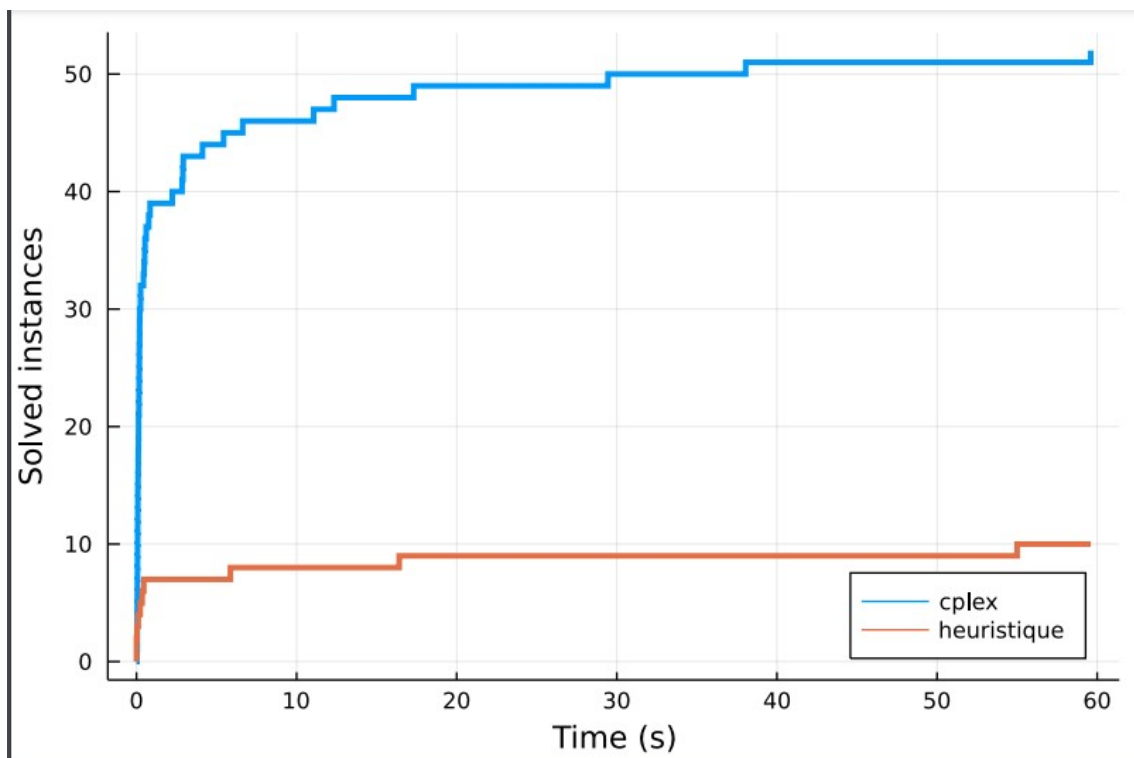


FIGURE 5.4 – Diagramme de performance du méthode simplex

resultsArray

Cette fonction génère un pdf (Latex) où il y a les informations de résolution sur toutes les instances générés.

5.2 Fonction de génération

Pour générer les grilles initiales , on a essayé des densités différentes entre 10% et 20% et pour des tailles différentes. Pour chaque case on choisit un nombre aléatoirement entre 0 et 1, si le nombre est inférieure à la densité on met une valeur dans la case sinon elle reste vide. En moyenne le nombre des cases rempli est égale à $densité * NombreDeCasesTotale$.

5.3 Fonction de Résolution

5.3.1 Résolution avec la méthode SIMPLEX

cplexSolve

Cette fonction prend un tableau 2D comme argument et applique tout le raisonnement qu'on a présenté au début du rapport et retourne les bords qui sont présents dans la boucle (*edgesV* et *edgesH*).

5.3.2 Résolution avec la méthode heuristique

heuristicSolve

Cette méthode reçoit un tableau 2D qui représente la grille. On calcule d'abord le nombre maximum de cases que la composante externe peut contenir. On crée un tableau 2D *mask* de zéros. Les valeurs de *mask* appartiennent à $\{0, 1\}$.

- Si la valeur est 1 : la case appartient à la composante extérieure
- Si la valeur est 0 : la case appartient à la composante intérieure

Après la création de ce tableau , on appelle la fonction *solve*. Si Le résultat obtenu est *true* alors la solution existe est elle est stocké dans *mask* donc on génère *edgesH* et *edgesV* avec la fonction *buildSolution* et on retourne le résultat.

solve

C'est une fonction récursive qui itère sur toutes les possibilités. *nbCells* dans les paramètres représente le nombre maximum de cases qu'on peut ajouter à la composante extérieure.

5.3.3 Résolution générale

solveDataSet

Cette Fonction permet de résoudre toutes les instances dans le répertoire *data* avec plusieurs méthodes possibles (cplex et heuristique) et mémorise les résultat

obtenus dans des sous répertoires du répertoire *res* comme représenté dans la partie `displaySolution` du rapport.

Results Array

Instance	cplex		heuristique	
	Temps (s)	Optimal ?	Temps (s)	Optimal ?
instanceTest.txt	29.45	×	60.0	
instanceTest1.txt	0.26	×	60.0	
instanceTest10.txt	4.12	×	60.0	
instanceTest11.txt	0.85	×	60.0	
instanceTest12.txt	2.92	×	60.0	
instanceTest13.txt	6.63	×	60.0	
instanceTest14.txt	5.44	×	60.0	
instanceTest15.txt	12.34	×	60.0	
instanceTest16.txt	59.6	×	60.0	
instanceTest17.txt	17.31	×	60.0	
instanceTest18.txt	38.05	×	60.01	
instanceTest19.txt	0.19	×	0.46	×
instanceTest2.txt	0.18	×	60.01	
instanceTest20.txt	0.17	×	5.87	×
instanceTest21.txt	0.25	×	60.01	
instanceTest22.txt	0.21	×	60.0	
instanceTest23.txt	11.07	×	60.0	
instanceTest24.txt	0.13	×	60.0	
instanceTest25.txt	0.13	×	60.0	
instanceTest26.txt	0.2	×	60.0	
instanceTest27.txt	0.15	×	60.0	
instanceTest28.txt	0.2	×	60.0	
instanceTest29.txt	0.03		60.0	
instanceTest3.txt	0.11	×	16.41	×
instanceTest30.txt	0.44	×	60.0	
instanceTest31.txt	0.48	×	60.0	
instanceTest32.txt	0.61	×	60.0	
instanceTest33.txt	0.51	×	60.0	
instanceTest34.txt	0.53	×	60.0	

Instance	cplex		heuristique	
	Temps (s)	Optimal ?	Temps (s)	Optimal ?
instanceTest35.txt	2.24	×	60.0	
instanceTest36.txt	0.02		60.0	
instanceTest37.txt	0.77	×	60.0	
instanceTest38.txt	2.9	×	60.0	
instanceTest39.txt	0.05	×	0.04	×
instanceTest4.txt	0.11	×	60.0	
instanceTest40.txt	0.03	×	0.36	×
instanceTest41.txt	0.05	×	0.0	×
instanceTest42.txt	0.05	×	0.0	×
instanceTest43.txt	0.05	×	60.15	
instanceTest44.txt	0.08	×	60.0	
instanceTest45.txt	0.05	×	0.23	×
instanceTest46.txt	0.07	×	55.0	×
instanceTest47.txt	0.06	×	0.11	×
instanceTest48.txt	0.05	×	60.0	
instanceTest49.txt	0.09	×	60.0	
instanceTest5.txt	0.09	×	60.0	
instanceTest50.txt	0.07	×	60.0	
instanceTest51.txt	0.1	×	60.0	
instanceTest52.txt	0.07	×	60.0	
instanceTest53.txt	0.08	×	60.0	
instanceTest6.txt	0.13	×	60.0	
instanceTest7.txt	0.18	×	60.0	
instanceTest8.txt	0.16	×	60.0	
instanceTest9.txt	2.85	×	60.0	
instanceTest.txt	29.45	×	60.0	
instanceTest1.txt	0.26	×	60.0	
instanceTest10.txt	4.12	×	60.0	
instanceTest11.txt	0.85	×	60.0	
instanceTest12.txt	2.92	×	60.0	

Instance	cplex		heuristique	
	Temps (s)	Optimal ?	Temps (s)	Optimal ?
instanceTest13.txt	6.63	×	60.0	
instanceTest14.txt	5.44	×	60.0	
instanceTest15.txt	12.34	×	60.0	
instanceTest16.txt	59.6	×	60.0	
instanceTest17.txt	17.31	×	60.0	
instanceTest18.txt	38.05	×	60.01	
instanceTest19.txt	0.19	×	0.46	×
instanceTest2.txt	0.18	×	60.01	
instanceTest20.txt	0.17	×	5.87	×
instanceTest21.txt	0.25	×	60.01	
instanceTest22.txt	0.21	×	60.0	
instanceTest23.txt	11.07	×	60.0	
instanceTest24.txt	0.13	×	60.0	
instanceTest25.txt	0.13	×	60.0	
instanceTest26.txt	0.2	×	60.0	
instanceTest27.txt	0.15	×	60.0	
instanceTest28.txt	0.2	×	60.0	
instanceTest29.txt	0.03		60.0	
instanceTest3.txt	0.11	×	16.41	×
instanceTest30.txt	0.44	×	60.0	
instanceTest31.txt	0.48	×	60.0	
instanceTest32.txt	0.61	×	60.0	
instanceTest33.txt	0.51	×	60.0	
instanceTest34.txt	0.53	×	60.0	
instanceTest35.txt	2.24	×	60.0	
instanceTest36.txt	0.02		60.0	
instanceTest37.txt	0.77	×	60.0	
instanceTest38.txt	2.9	×	60.0	
instanceTest39.txt	0.05	×	0.04	×
instanceTest4.txt	0.11	×	60.0	

Instance	cplex		heuristique	
	Temps (s)	Optimal ?	Temps (s)	Optimal ?
instanceTest40.txt	0.03	×	0.36	×
instanceTest41.txt	0.05	×	0.0	×
instanceTest42.txt	0.05	×	0.0	×
instanceTest43.txt	0.05	×	60.15	
instanceTest44.txt	0.08	×	60.0	
instanceTest45.txt	0.05	×	0.23	×
instanceTest46.txt	0.07	×	55.0	×
instanceTest47.txt	0.06	×	0.11	×
instanceTest48.txt	0.05	×	60.0	
instanceTest49.txt	0.09	×	60.0	
instanceTest5.txt	0.09	×	60.0	
instanceTest50.txt	0.07	×	60.0	
instanceTest51.txt	0.1	×	60.0	
instanceTest52.txt	0.07	×	60.0	
instanceTest53.txt	0.08	×	60.0	
instanceTest6.txt	0.13	×	60.0	
instanceTest7.txt	0.18	×	60.0	
instanceTest8.txt	0.16	×	60.0	
instanceTest9.txt	2.85	×	60.0	