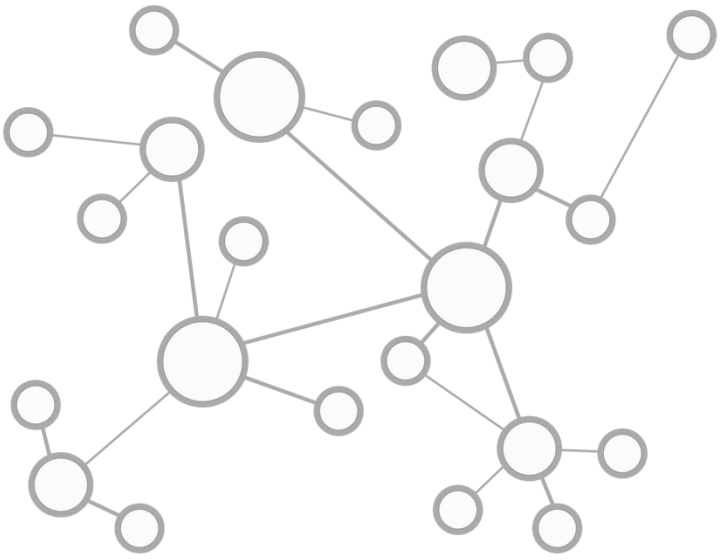
Analyse d'algorithme – Dossier #2

Remplissage par diffusion – Backtracking – Dijkstra



Baptiste Thomas – Cédric Klodzinski – Kevin Valente
ISIMA, F2, FEVRIER 2022

ISIMA

Table des matières

Table des figures

I – Remplissage par diffusion

Kevin V. (Programme) & Cédric K. (analyse)

A) Principe et intérêt

L'algorithme que nous abordons ici permet le remplissage d'une zone par diffusion d'une couleur, sur une grille de pixels. Il s'agit de partir d'un pixel de la grille et de diffuser la valeur qu'il possède à ses voisins, jusqu'à heurter un contour. On peut assimiler ce procédé à un parcours d'arbre.

Ce genre d'algorithme est utilisé dans des applications de dessin ou d'infographie (l'outil remplissage par exemple), ainsi que dans certains jeux basés sur les associations d'objets de même couleur.

B) Langage et dépendance du code

Le programme fourni est écrit en C#, avec le Framework .NET. Il respecte la charte de style préconisée par Microsoft.

C) Analyse du code

Type de programme : Application console
Entrée : un tableau de pixel représentant une image avec sa taille et sa largeur, un pixel cible, une couleur cible, une couleur finale et une tolérance.
Sortie : Le tableau de pixel avec remplissage de la zone correspondante.

Le code dispose d'une classe utilitaire Coordinate, qui représente les coordonnées d'une case de la grille de pixels que l'on traite. On stocke ainsi les deux coordonnées X et Y :

```
internal class Coordinate
{
	public int X { get; set; }
	public int Y { get; set; }

	public Coordinate(int x, int y)
	{
		this.X = x;
		this.Y = y;
	}
}
```

On dispose aussi d'une fonction d'affichage de la grille dans la console, en $o\big(length \times (height + 2)\big)$.

```
static void displayTab(int[,] tabPixel, int lenght, int height)
{
	int i, j;
	string line = "+-";

	for (i = 0; i < lenght * 2; i++)
	{
		line += "-";
	}
	line += "+";

	Console.WriteLine(line);

	for (i = 0; i < lenght; i++)
	{
		Console.Write("| ");
		for (j = 0; j < height; j++)
		{
			Console.Write($"{tabPixel[i, j]} ");
		}
		Console.WriteLine("|");
	}

	Console.WriteLine(line);
}
```

Le cœur de l'algorithme est défini dans la fonction fill, qui applique le remplissage à la grille (cf page suivante). Cette fonction prend en paramètres :

- La grille de pixels, avec sa largeur et hauteur
- Les coordonnées du pixel de départ
- La valeur cible, qui détermine la zone autour du pixel de départ et qui est remplacée à la fin
- La valeur à appliquer, qui remplace la valeur cible
- Une tolérance, qui détermine les valeurs de pixel qui sont considérées comme des contours de la zone à remplacer

```
static void fill(int[,] tabPixel, int width, int height, int pixelLine,
int pixelColumn, int targetColor, int finalColor, int tolerance)
{
	Coordinate xy;
	Stack<Coordinate> stack = new Stack<Coordinate>();

	int west, east, line, i;
	int targetMin = targetColor - tolerance;
	int targetMax = targetColor + tolerance;

	stack.Push(new Coordinate(pixelColumn, pixelLine));

	while (stack.Count > 0)
	{
		xy = stack.Pop();

		west = xy.X;
		east = xy.X;
		line = xy.Y;

		while ((west > 0) &&
				(tabPixel[line, west - 1] >= targetMin) &&
				(tabPixel[line,  west - 1] <= targetMax))
		{
			west--;
		}

		while ((east < width-1) &&
				(tabPixel[line, east + 1] >= targetMin) &&
				(tabPixel[line, east + 1] <= targetMax))
		{
			east++;
		}

		for (i = west; i <= east; i++)
		{
			tabPixel[line, i] = finalColor;

			if ((line != 0) &&
				(tabPixel[line - 1, i] != finalColor) &&
				(tabPixel[line - 1, i] >= targetMin) &&
				(tabPixel[line - 1, i] <= targetMax))
			{
				stack.Push(new Coordinate(i, line - 1));
			}

			if ((line != height-1) &&
				(tabPixel[line + 1, i] != finalColor) &&
				(tabPixel[line + 1, i] >= targetMin) &&
				(tabPixel[line + 1, i] <= targetMax))
			{
				stack.Push(new Coordinate(i, line + 1));
			}
		}
	}
}
```

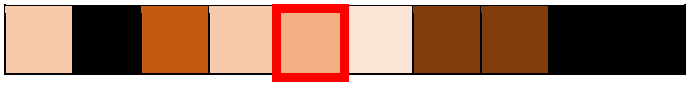
Une pile de coordonnées permet de nous aider dans le parcours de la zone de pixels. On commence par y ajouter le pixel de départ. On détermine aussi les valeurs limites qui représentent l'intervalle de valeurs de la zone à délimiter, d'après la valeur cible et la tolérance. On a donc l'intervalle de valeurs

$$I = [cible - tolerance\,;cible + tolerance]$$

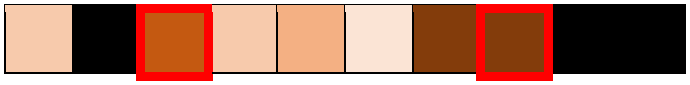
Ensuite, tant que des pixels sont dans la pile, on récupère la ligne du pixel en tête de pile (coordonnée Y). Puis on détermine, sur cette ligne, les pixels contigus ayant des valeurs dans l'intervalle déterminé au début de l'algorithme, en la parcourant vers l'ouest et l'est depuis le pixel en tête de pile. On obtient donc la zone à recolorier sur cette ligne :

Soit l'intervalle de couleurs suivant :

Soit la ligne suivante, avec le pixel orange encerclé le pixel en tête de pile :



On détermine les limites de la zone vers l'ouest et vers l'est, selon les valeurs de l'intervalle :



La zone sélectionnée par la diffusion est donc la zone en vert :

Figure 1 : explication du parcours est-ouest

Puis, pour chaque pixel sur cette portion de ligne, on remplace la valeur par la valeur à appliquer, et on regarde les voisins verticaux de ces pixels. Si leur valeur est dans l'intervalle et qu'elle n'est pas égale à la valeur à appliquer, on les ajoute à la pile. Ne pas vérifier que la valeur finale est déjà présente conduit à une boucle infinie. On recommence le processus pour chaque pixel de la pile.

Ici, on illustre les étapes de l'algorithme, ligne par ligne, avec comme pixel de départ celui en haut à gauche. L'intervalle de la zone est celui donné précédemment en exemple et la couleur de destination est le vert :

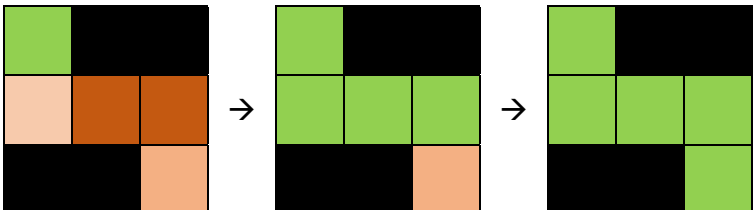


Figure 2 : explication du parcours nord-sud

À la fin, la zone est recoloriée.

D) Test

Pour tester l'algorithme, nous allons utiliser les cas suivants :

- Cas « normal », avec $Tolerance \in [0; \infty[$ et pixel de départ avec valeur valide
  Pour la grille de départ suivante, en partant du pixel central de valeur 1, avec pour valeur cible 1, pour valeur à appliquer 1 et pour tolérance 2 :

0	0	1	2	1
1	5	0	1	1
1	1	1	1	1
3	0	1	6	6
0	1	1	0	1

  On obtient la grille :

1	1	1	1	1
1	5	1	1	1
1	1	1	1	1
1	1	1	6	6
1	1	1	1	1

  Ce qui est attendu.

- $Tolerance = 0$
  Pour la grille de départ suivante, en partant du pixel central de valeur 1, avec pour valeur cible 1, pour valeur à appliquer 8 et pour tolérance 0 :

0	0	1	2	1
1	5	0	1	1
1	1	1	1	1
3	0	1	6	6
0	1	1	0	1

  On obtient la grille :

0	0	1	2	8
8	5	0	8	8
8	8	8	8	8
3	0	8	6	6
0	8	8	0	1

  Ce qui est attendu.

- $Tolerance = \infty$

  Pour la grille de départ suivante, en partant du pixel central de valeur 1, avec pour valeur cible 1, pour valeur à appliquer 7 et pour tolérance 10 (soit plus que la valeur maximale dans la grille) :

0	0	1	2	1
1	5	0	1	1
1	1	1	1	1
3	0	1	6	6
0	1	1	0	1

  On obtient la grille :

7	7	7	7	7
7	7	7	7	7
7	7	7	7	7
7	7	7	7	7
7	7	7	7	7

  Ce qui est attendu.

- Pixel de départ avec une valeur non valide

  Pour la grille de départ suivante, en partant du pixel central de valeur 1, avec pour valeur cible 7, pour valeur à appliquer 7 et pour tolérance 2 :

0	0	1	2	1
1	5	0	1	1
1	1	1	1	1
3	0	1	6	6
0	1	1	0	1

  On obtient la grille :

0	0	1	2	1
1	5	0	1	1
1	1	7	1	1
3	0	1	6	6
0	1	1	0	1

E) Limites de l'algorithme

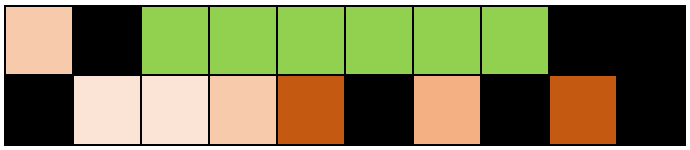
On remarque qu'une première optimisation a été faite via le parcours des lignes vers l'ouest et l'est. On évite ainsi de nombreux ajouts et suppressions dans la pile.

On peut toutefois noter des calculs qui peuvent être redondants dans certains cas.

L'algorithme de recherche d'un voisin vertical valide, pour prolonger la zone à recolorier, est prévu pour gérer des lignes où les valeurs valides ne seraient pas contigües, comme dans l'exemple suivant :

Soit l'intervalle de couleurs suivant :

Soient les lignes d'une image, avec la ligne en vert celle qui vient d'être traitée :



On ajoute donc les pixels suivants à la pile :

Figure 3 : illustration des limites de l'algorithme

Ce comportement permet d'obtenir toute la zone contigüe. Cependant, il provoque par la même occasion l'ajout de plusieurs pixels valides par ligne, ce qui peut provoquer plusieurs calculs de zone contigüe par ligne, comme dans l'exemple précédent (3 pixels sur la zone, donc 3 traitements identiques).

II – Algorithme de Backtrack (appliqué au sudoku)

Cédric K. (Programme) & Baptiste T. (analyse)

A) Principe et intérêt

La famille d'algorithmes « retour arrière » est notamment utilisée pour la résolution de problèmes impliquant la satisfaction d'un certains nombres de contraintes (comme le sudoku). Le principe est de tester de manière systématique la totalité des affectations possibles d'une variable du problème. À partir de cette affectation, on teste de manière récursive les affectations qui en découlent. Si aucune solution n'est trouvée, on revient en arrière. Cela s'apparente à un parcours en profondeur d'un arbre.

Appliqué au sudoku cela revient à tester pour chaque case, chaque possibilité.

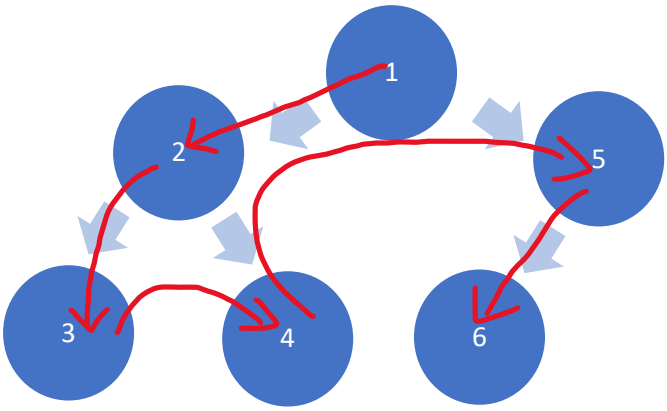


Figure 4 : illustration de l'algorithme de bactrack

B) Dépendance externe du code

Le programme fournit est écrit en C#, avec le Framework .NET. Il respecte la charte de style préconisée par Microsoft. Le programme nécessite la version 6 de .NET au minimum, car il y a eu des changements de structure qui empêchent la rétrocompatibilité.

C) Analyse du code

Type de programme : application console
Entrée : un tableau représentant une grille de sudoku à compléter.
Sortie : Une solution possible de la grille passée en entrée.

Le code utilise la classe suivante pour représenter une case vide, à la position (X;Y) d'une grille de sudoku.

```
internal class EmptyCell
{
	public int X { get; set; }
	public int Y { get; set; }
	public int Possibilities { get; set; }
}
```

Voici la fonction principale :

```
bool Solve(int[,] grid)
{
	// Initializes global arrays to false
	for (int i = 0; i < 9; i++)
		for (int j = 0; j < 9; j++)
			existsOnLine[i, j] = existsOnColumn[i, j]
					= existsInBlock[i, j] = false;

	// Saves already present values in arrays
	int k;
	for (int i = 0; i < 9; i++)
		for (int j = 0; j < 9; j++)
			if ((k = grid[i, j]) != 0)
				existsOnLine[i, k - 1] = existsOnColumn[j, k - 1]
					= existsInBlock[3 * (i / 3) + (j / 3), k - 1]
					= true;

	// Creates the list of cells, sorted by number of possibilities
	positions = new List<EmptyCell>();

	for (int i = 0; i < 9; i++)
		for (int j = 0; j < 9; j++)
			if (grid[i, j] == 0)
				positions.Add(new EmptyCell() { Y = i, X = j,
				Possibilities = GetPossibilities(grid, i, j) });

	positions = positions.OrderBy(p => p.Possibilities).ToList();

	// Runs the backtracking algorithm from the start of the list
	bool ret = IsValid(grid, 0);

	return ret;
}
```

On initialise d'abord les différents tableaux de vérification de la présence d'un numéro sur une ligne, une colonne, et un bloc. On va ensuite les mettre à jour, avec les cases de la grille qui sont déjà remplies.

On crée maintenant les objets associés aux cases vides et on leur attribue une valeur possible.
On vérifie que la grille est valide à partir de ces objets.
Fonction GetPossibilities :

```
int GetPossibilities(int[,] grid, int i, int j)
{
	int count = 0;
	// For each value, checks if it is valid in the cell
	for (int k = 0; k < 9; k++)
		if (!existsOnLine[i, k] && !existsOnColumn[j, k] &&
		!existsInBlock[3 * (i / 3) + (j / 3), k])
			count++;

	return count;
}
```

Cette fonction retourne une valeur possible, par rapport aux valeurs déjà présentes sur la même ligne, même colonne et même bloc.

Fonction de validation :

```
bool IsValid(int[,] grid, int position)
{
	if (position >= positions.Count || positions[position] == null)
	// If no more cells, valid branch
		return true;

	int j = positions[position].X, i = positions[position].Y;
	// Gets the coordinates of the cell in the grid

	for (int k = 0; k < 9; k++)
	{
		// For each value, check if it doesn't exist
		if (!existsOnLine[i, k] && !existsOnColumn[j, k] &&
			!existsInBlock[3 * (i / 3) + (j / 3), k])
		{
			// Add k to saved values
			existsOnLine[i, k] = existsOnColumn[j, k] =
					existsInBlock[3 * (i / 3) + (j / 3), k] = true;

			if (IsValid(grid, position + 1))
			{
				// Saves the valid choice in the grid
				grid[i, j] = k + 1;
				return true;
			}
			// Removes k from saved values
			existsOnLine[i, k] = existsOnColumn[j, k]
				= existsInBlock[3 * (i / 3) + (j / 3), k] = false;
		}
	}

	return false ;
}
```

C'est la fonction la plus importante. C'est ici qu'on vérifie que la grille est valide et que l'on réalise la récursivité.

Si la liste de cases vides est vide, on considère la grille comme valide. Sinon, on parcourt la liste des cases vides de manière récursive. Si la valeur est valide, on passe à la possibilité suivante.

D) Tests

Pour cet algorithme, la vérification de l'efficacité a été faite en comparant les résultats avec la grille de sudoku et la solution présente dans les journaux. Difficile donc de l'illustrer dans ce rapport, mais les tests ont été concluants. De plus, l'utilisation de divers vérificateurs de grilles en lignes ont confirmé la justesse des résultats.

III – Algorithme de Djikstra

Baptiste T. (Programme) & Kevin V. (analyse)

A) Principe et intérêt

L'algorithme de Dijkstra est un algorithme particulièrement connu, qui a pour but d'apporter une solution au problème du plus court chemin. Il s'agit un problème de théorie des graphes dont le but est, comme son nom l'indique, de trouver une solution optimale permettant d'aller d'un point A à un point B dans un réseau de nœuds.

En termes d'application, cet algorithme est, entre autres, utilisé dans le domaine de la recherche opérationnelle. Il permet également d'aider des entreprises, comme les transporteurs, à planifier des routes.

L'algorithme de Dijkstra repose sur le parcours en largeur d'un graphe, dont en voici une explication animée :

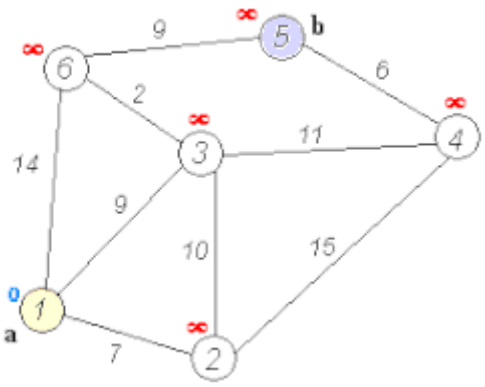


Figure 5 : animation représentant l'algorithme de Dijkstra

Source image : Wikipédia - https://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra#Applications, consultée le 25/02/2022 - License : domaine public.

B) Langage et dépendance du code

Le programme fourni est écrit en C#, avec le Framework .NET. Le programme nécessite .NET 6 pour tourner. Il respecte la charte de style préconisée par Microsoft.

C) Analyse du code

Type de programme : application console
Entrée : un graphe
Sortie : la distance minimale entre la source et chaque sommet

Tout d'abord, un graphe est représenté sous la forme d'un tableau en deux dimensions, selon la logique suivante : le croisement entre une ligne i et une colonne j représente un arc entre le sommet i et le sommet j. Un 0 pour cette valeur représente l'absence d'arc entre les points i et j.

Par exemple, voici une correspondance entre un tableau et un graphe :

	A	B	C	D
A	0	1	0	5
B	1	0	2	0
C	0	2	0	1
D	5	0	1	0

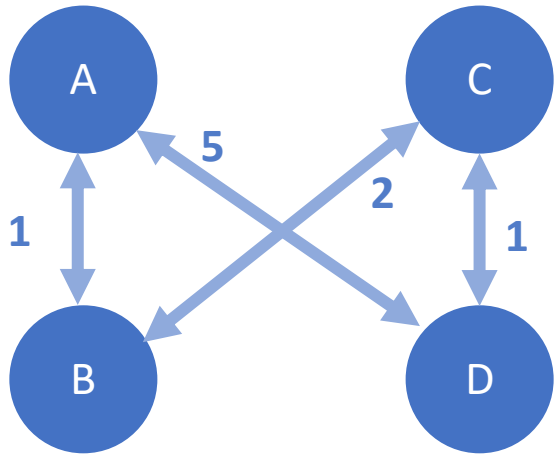


Figure 6 : représentation d'un graphe

Concernant le code, il se sépare en plusieurs fonction. Tout d'abord, une fonction **minDistance** qui trouve le chemin le moins long entre deux points :

```
int minDistance(int[] dist, bool[] sptSet)
{
      int min = int.MaxValue, min_index = -1;

      for (int v = 0; v < V; v++)
      if (sptSet[v] == false && dist[v] <= min)
      {
            min = dist[v];
            min_index = v;
       }
       return min_index;
}
```

On notera que pour cette fonction, et les suivantes, V est une variable globale contenant la taille du graphe, autrement dit, le nombre de sommets.

Ensuite, une courte fonction d'affichage pour le résultat :

```
private void printSolution(int[] dist, int n)
{
      Console.Write("Point\t\tDistance de la source\n");
      for (int i = 0; i < V; i++)
            Console.Write(i + " \t\t " + dist[i] + "\n");
}
```

Enfin, l'algorithme en lui-même se trouve ci-dessous. Les commentaires (en gris) permettent d'expliciter le déroulé de l'algorithme, pour faciliter la compréhension.

```
public void dijkstra(int[,] graph, int src)
{
      int[] dist = new int[V]; // distances stockées

      // bool[i] a 'True' si le sommet i appartient au chemin le plus cours
      bool[] sptSet = new bool[V];

      for (int i = 0; i < V; i++)
      {
            dist[i] = int.MaxValue;
            sptSet[i] = false;
       }

      // La distance du début au début est toujours 0
      dist[src] = 0;

      // Boucle pour trouver le chemin le plus court parmi les sommets
      for (int count = 0; count < V - 1; count++)
      {
            // on prend le poids min
            int u = minDistance(dist, sptSet);

            //on marque le sommet comme visité
            sptSet[u] = true;

            // On met à jour avec les valeurs suivante
            for (int v = 0; v < V; v++){
                  // On met à jour seulement si : pas déjà dans sptSet, lien entre
                  u et v existant et poids inférieur à la valeur actuel de dist[v]
                  if (!sptSet[v] && graph[u, v] != 0 &&
                        dist[u] != int.MaxValue && dist[u] + graph[u, v] < dist[v])
                  {
                       dist[v] = dist[u] + graph[u, v];
                  }
       }

       printSolution(dist, V); // on affiche
}
```

D) Test

Tout d'abord, prenons l'exemple plus haut (figure 6) :

Le programme devrait sortir :

B 1 – C 3 – D 4

La sortie est la suivante :

```
Point         Distance de la source
0               0
1               1
2               3
3               4
```

Ce qui est correct.

Ensuite, afin de tester l'algorithme, nous avons utilisé un solveur, disponible en ligne via le lien suivant : http://www.jakebakermaths.org.uk/maths/dijkstrasalgorithmsolverv9.html (consulté le 25/02/2022).

Le graphe représenté par le tableau suivant a été utilisé :

```
int[,] graph = new int[,] {
      { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
      { 4, 0, 8, 0, 0, 0, 0, 11, 0 },
      { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
      { 0, 0, 7, 0, 9, 14, 0, 0, 0 },
      { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
      { 0, 0, 4, 14, 10, 0, 2, 0, 0 },
      { 0, 0, 0, 0, 0, 2, 0, 1, 6 },
      { 8, 11, 0, 0, 0, 0, 1, 0, 7 },
      { 0, 0, 2, 0, 0, 0, 6, 7, 0 }
};
```

Les distances calculées étaient correctes.

Les deux tests que nous avons choisis d'illustrer ici sont représentatifs des autres tests que nous avons réalisé sur l'algorithme, mais étant donné leur caractère répétitif, nous n'avons pas jugé bon de tous les présenter.

Nous avons tout de même voulu tester quelques cas particuliers, notamment le cas d'un tableau rempli de 0. Le programme retourne alors des valeurs incohérentes, le cas n'étant pas traité. Nous avons également testé les cas où la symétrie n'était pas respectée dans la représentation du graphe. Dans ce cas, la valeur prise en compte par le programme est uniquement celle du côté gauche de la matrice.