Analyse comparative des optimisations locales et globales

Corentin POUPRY
corentin.poupry@edu.esiee.fr

Néo JONAS
neo.jonas@edu.esiee.fr

Résumé

Ce rapport compare les valeurs de stratégies d'**optimisation locale** (souvent appelées stratégies gloutonnes) avec les valeurs de stratégies d'**optimisation globale** calculées par programmation dynamique.

Introduction

Problèmes traitées

Pour pouvoir comparer les comportements des deux stratégies, on s'appuiera sur cinq problèmes d'algorithmique différents, chacun détaillé dans sa section respective

Table des matières

Sources

Tous les programmes utilisés seront fournis en annexe dans ce document mais sont aussi consultable sur le dépot Git du projet, accessible ici.

Méthodologies

Toutes les données énoncées dans ce rapport sont disponible dans le dépot Git du projet, accessible ici, dans le dossier `data`. Les données statistiques sont calculées par GNU Octave au moyen des fonctions énoncées dans la section 26.1 Descriptive Statistics et présentes dans le script `stats.m`.

Pour former les histogrammes avec un échantillon de données représentatif, on exécute soit 5000 ou 10000 runs pour obtenir les données les plus représentatif des performances réelles des programmes.

Technologies

Les histogrammes présent dans ce document sont calculés lors de la compilation du rapport LaTeXavec les données bruts présentes dans `data`. Les différents codes dans ce rapport ont été compilés et exécutés avec succès sous OpenJDK v18.0.1.1.

1 Problème du chemin le moins coûteux du robot

Ce problème est tiré de l'exercice 6 du TD 5 de l'unité `IGI-2102`.

Détails

Un petit robot est place en case $(l, c) = (0, 0)$ d'une grille à L lignes et C colonnes. Le robot doit atteindre la case $(l, c) = (L - 1, C - 1)$. Ses mouvements possibles sont Nord (N), Nord-Est (NE), Est (E).

Le robot étant sur la case $(l-1, c)$, le déplacement N le conduit en case (l, c) avec un coût $n(l-1, c)$. Le robot étant sur la case $(l-1, c-1)$ le déplacement NE le conduit en case (l, c) avec un coût $ne(l-1, c-1)$. Le robot étant sur la case $(l, c-1)$ le déplacement E le conduit en case (l, c) avec un coût $e(l, c-1)$.

Afin que le robot ne sorte pas de la grille, les mouvements N depuis une case située en ligne $l = L-1$ sont de coût infini. De même : les mouvements NE depuis une case située en ligne $l = L-1$ ou colonne $c = C-1$, et les mouvements E depuis une case située en colonne $c = C-1$.

Résolution

Optimisation globale La stratégie optimale a été étudiée en TD et n'est pas détaillé dans ce rapport. Cependant, les commentaires du code sont disponibles en annexe de ce rapport pour la compréhension de la résolution du problème par programmation dynamique.

Optimisation locale Le robot choisira toujours le mouvement le moins coûteux entre ceux qui lui sont possibles. Pour cette stratégie, on s'attend déjà à avoir un énorme écart entre l'optimisation locale et globale tant la stratégie est d'une naïveté extrême.

Analyse statistique

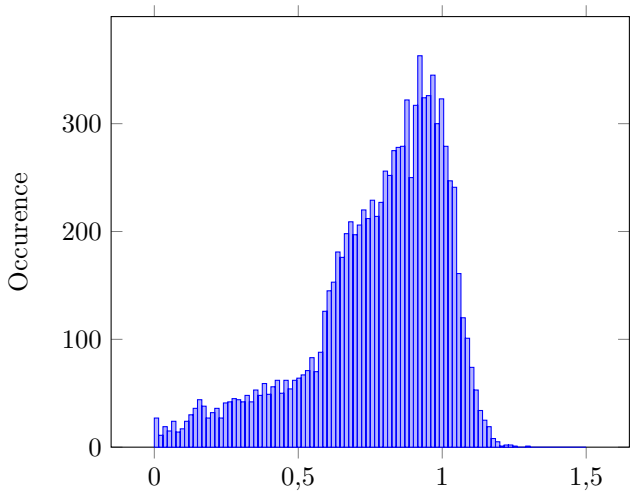

FIGURE 1 – Différence relative entre stratégie optimal et naïf pour le chemin le moins coûteux du robot

On notera qu'on est ici sur une *minimisation* et non maximisation, d'où l'utilisation de la formule suivante

$$\frac{g - v^*}{v^*}$$

Stratégie gloutonne	Moyenne	Médiane	Variance	Écart type
Par choix de la case minimum	0.77633	0.82928	0.056029	0.23671

TABLE 1 – Statistiques pour le problème du chemin minimum du robot

On remarquera qu'une majorité d'occurrences se font au niveau du 1.

$$\begin{aligned}\frac{g - v^*}{v^*} = 1 &\Leftrightarrow g - v^* = v^* \\ &\Leftrightarrow g = 2v^*\end{aligned}$$

On peut alors dire que pour une majorité des valeurs, *la stratégie gloutonne est deux fois moins performante que la stratégie optimale dans ce cadre du problème en terme de valeur retournée.*

2 Problème des sacs de valeur maximum

Détails

On est en présence d'un ensemble de n objets, un objet i possédant une taille et une valeur notées respectivement t_i et v_i. La question étant, pour un sac de contenance C, calculer un sous-ensemble d'objet de taille inférieure ou égale à la contenance C du sac et de valeur maximum.

Résolution

Optimisation globale La stratégie optimale a été étudiée en cours et n'est pas détaillé dans ce rapport. Cependant, les commentaires du code sont disponibles en annexe de ce rapport pour la compréhension de la résolution du problème par programmation dynamique.

Optimisation locale Pour organiser les objets et établir un ordre d'insertion dans le sac de contenance C. Pour cela, on va on peut s'intéresser au ratio valeur / poids des objets. Pour un objet i, on lui associe le ratio suivant

$$\text{ratio}(i) = \frac{v_i}{t_i}$$

Mais on peut aussi imaginer un façon beaucoup plus simpliste d'ordonner l'ensemble des objets, à savoir uniquement sur la base de la valeur v_i ou de la taille t_i d'un objet i pour ensuite essayer de mettre les objets les plus intéressants dans le sac et ainsi de suite.

Analyse statistique

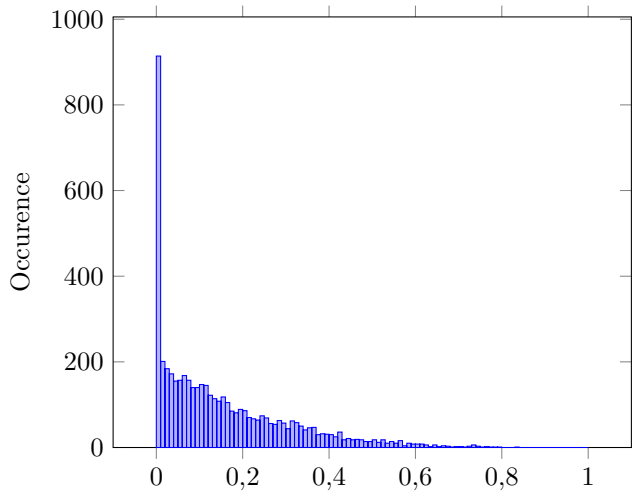

FIGURE 2 – Différence relative entre chemin optimal et naïf (par valeur) pour le sac de valeur maximale

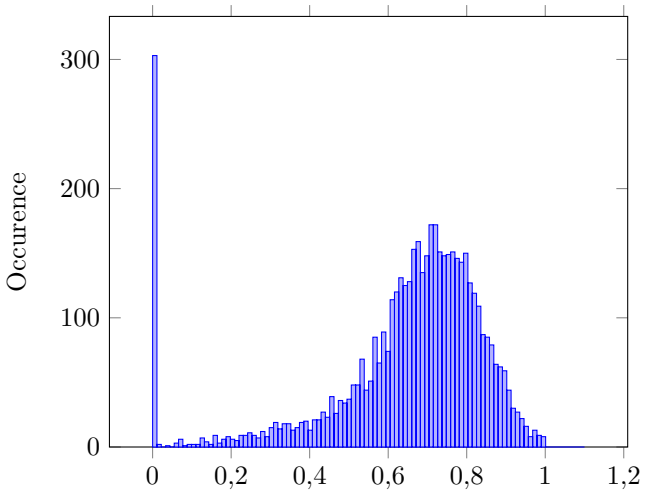

FIGURE 3 – Différence relative entre chemin optimal et naïf (par taille) pour le sac de valeur maximale

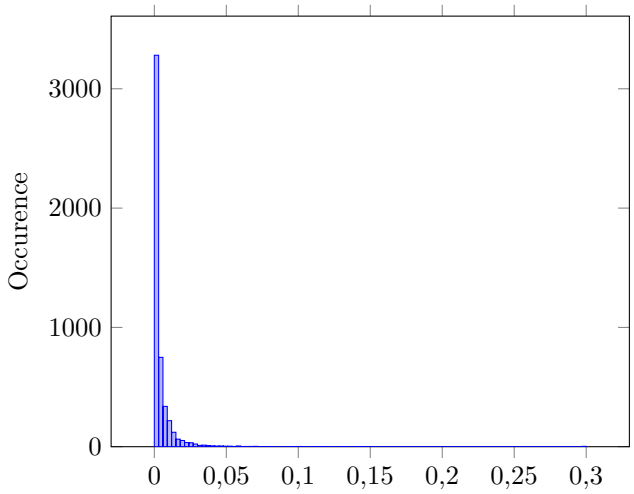

FIGURE 4 – Différence relative entre chemin optimal et naïf (par ratio) pour le sac de valeur maximale

Stratégie gloutonne	Moyenne	Médiane	Variance	Écart type
Naïve par ratio	0.0042197	0.0010018	0.00014425	0.01201
Naïve par valeur	0.15141	0.10767	0.022986	0.15161
Naïve par taille	0.63943	0.69247	0.049271	0.22197

TABLE 2 – Statistiques pour le problème du sac de valeur maximale

On voit très clairement que la différence de résultat entre les trois stratégies naïves est grande. Cependant, la stratégie par ratio vient donner des résultats plutôt proches de la stratégie optimale (visible sur le graphe par la faible dispersion), venant supporter l'hypothèse que *la stratégie gloutonne par ratio est ici adaptée pour donner une résultat approximant correctement le meilleur résultat possible.*

Digression sur Java

Pour pouvoir trier le tableau T contenant les objets Java représentant donc nos véritables « objets » du problème, on pourrait d'aborder penser à implémenter l'interface `Comparable` sur ces objets. Bien que très facile à mettre en place, cette méthode pose un problème : en effet, il ne sera d'implémenter qu'une seule stratégie de trie. Pour pallier à ça, Java 8 introduit la fonction statique `Comparator.comparing`, permettant de créer un méthode de comparaison entre deux objets et de pouvoir l'appliquer avec `Arrays.sort`.

3 Problème de la répartition optimale d'un temps de travail

Détails

Supposons qu'un étudiant dispose de H heures de révision et que pour chaque unité i, ledit étudiant a estimé la note qu'il obtiendrait en consacrant t heures de révision à cette unité. Ce problème consiste à trouver la répartition optimale des H heures afin de maximiser la moyenne obtenue (ce qui revient à maximiser la somme des notes des i unités).

Résolution

Optimisation globale La stratégie optimale a été étudiée en cours et n'est pas détaillé dans ce rapport. Cependant, les commentaires du code sont disponibles en annexe de ce rapport pour la compréhension de la résolution du problème par programmation dynamique.

Optimisation locale Pour résoudre ce problème naïvement, on peut proposer un algorithme heuristique simple : soit $e(i,t)$ l'estimation de la note pour l'unité i avec un temps de révision t. Supposons qu'il nous reste $h \leq H$ heures de révisions à placer, on choisira de placer une heure supplémentaire de révision dans l'unité qui maximisera la valeur suivante

$$e(i,t+1) - e(i,t)$$

Ainsi, on pourra s'assurer localement que travailler cette unité est la meilleure décision sur le moment.

Analyse statistique

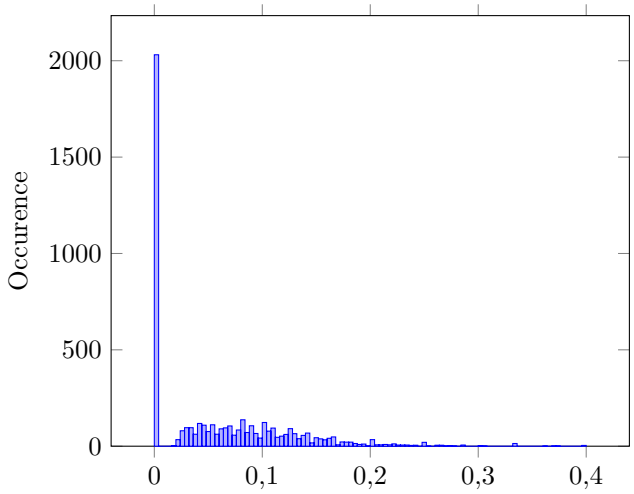

FIGURE 5 – Différence relative entre chemin optimal et naïf pour le problème de répartition optimale d'un temps de travail

Stratégie gloutonne	Moyenne	Médiane	Variance	Écart type
Par maximisation de différence	0.057978	0.043478	0.0041908	0.064737

TABLE 3 – Statistiques pour le problème de la répartition optimale d'un temps de travail

ATTENTION pour ce problème, nous avons identifié ce qui nous semble être un problème avec la stratégie optimale, en effet, la récupération du gain maximum `M[E][S]` semble retourner la somme des gains maximums de chaque entrepôt, expliquant un histogramme aussi défavorable pour la stratégie gloutonne.

4 Problème de la répartition optimale d'un stock

Détails

Un grossiste dispose d'un stock S a répartir sur n entrepôts. Pour tout entrepôt i, $i \in [0 : n]$, et tout stock s, $s \in [0 : S + 1]$, ce grossiste connaît le gain $g(i, s)$ obtenu en livrant le stock s à l'entrepôt i. On demande de calculer le gain $m(n, S)$ d'une répartition optimale du stock S sur les n entrepôts.

Résolution

Optimisation globale La stratégie optimale a été étudiée en cours et n'est pas détaillé dans ce rapport. Cependant, les commentaires du code sont disponibles en annexe de ce rapport pour la compréhension de la résolution du problème par programmation dynamique.

Optimisation locale Pour l'optimisation locale, sur le modèle de l'algorithme d'optimisation locale pour le problème de la répartition optimale d'un temps de travail, on cherche à maximiser la différence de grain entre un stock s et $s+1$ d'un entrepôt i, soit le calcul suivant

$$g(i, s+1) - g(i, s)$$

Analyse statistique

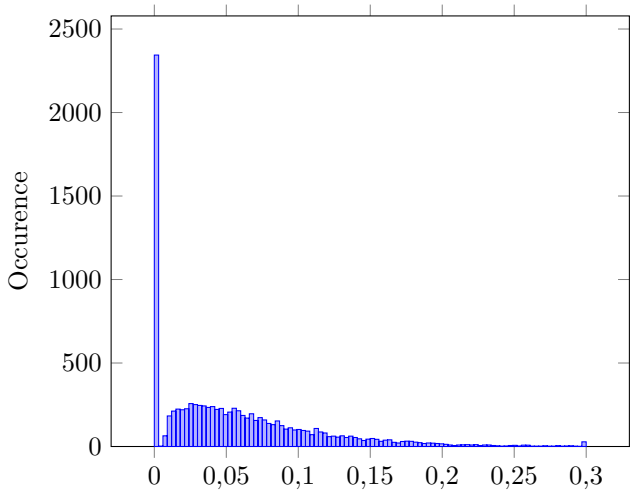

FIGURE 6 – Différence relative entre chemin optimal et naïf pour le problème de répartition optimale d'un stock

Stratégie gloutonne	Moyenne	Médiane	Variance	Écart type
Par maximisation de différence	0.055558	0.042553	0.0031038	0.055712

TABLE 4 – Statistiques pour le problème de répartition optimale d'un stock

La dispersion des valeurs est, pour la majorité d'entre elles, inférieure à 0.2 dans le graphique ci-dessus, indiquant une certaine efficacité de l'algorithme naïf.

5 Problème du chemin minimum dans un triangle

Détails

Cet exercice est le même que celui des exercices 18 et 67 du Projet Euler.

En commençant par le haut du triangle ci-dessous et en se déplaçant vers les chiffres adjacents de la rangée inférieure, trouver le chemin maximum rejoignant le rangée inférieure du triangle.

75
95 64
17 47 82
18 35 87 10
20 04 82 47 65
19 01 23 75 03 34
88 02 77 73 07 63 67
99 65 04 28 06 16 70 92
41 41 26 56 83 40 80 70 33
41 48 72 33 47 32 37 16 94 29
53 71 44 65 25 43 91 52 97 51 14
70 11 33 28 77 73 17 78 39 68 17 57
91 71 52 38 17 14 91 43 58 50 27 29 48
63 66 04 68 89 53 67 30 73 16 69 87 40 31
04 62 98 27 23 09 70 98 73 93 38 53 60 04 23

FIGURE 7 – Triangle présenté dans l'exercice 18 du Projet Euler

Résolution

Mise en place

Posons le problème : considérons que le triangle à traiter est des m niveaux que l'on numérotera de 0 (le haut du triangle) à $m-1$ (le bas du triangle). On peut remarquer qu'au niveau 0 on à 1 valeurs, au niveau 1, 2 valeurs, au niveau 2, 3 valeurs etc... Ainsi, le nombre n de valeurs dans un triangle à m niveaux est la somme des m premiers entiers, soit

$$n = \sum_{i=0}^{m} i = \frac{m(m+1)}{2} \tag{1}$$

On stockera les données du triangle dans un tableau $T = [0 : n]$. Une telle structure de donnée pose une question cruciale pour la résolution du problème : supposons que nous avons la valeur associé à l'index i. Comment savoir l'indice du descendant de gauche et droit de la valeur associé à l'indice i ?

Pour ça il faudra procéder comme suit :

— Trouver le niveau l dans le triangle auquel appartient i.

— Déduire le niveau p de l'indice i dans le niveau l.

— Ayant le niveau l et la position p de l'indice i dans son niveau, retourner l'indice $g(i)$ de son descendant gauche

Calcul du niveau l Supposons que l'on a une valeur d'indice i se trouvant à un niveau l inconnu. Il faut d'abord remarquer que l'équation (1) donnera la position dans le tableau de la dernière valeur du triangle. Si on note l'indice de la dernière valeur i_{max}, la formule (1) nous donnera donc, pour un triangle à m niveaux, l'indice $i_{max} + 1$, l'index des tableaux commençant à 0 en Java. Ainsi on peut poser

$$\frac{l(l+1)}{2} - 1 = i_{max}$$

A partir de là, il est très simple de trouver le niveau l correspondant à un indice i d'une valeur séquentiellement. Pour cela, on augmentera le niveau l cherché jusqu'à ce que $i_{max} \leq i$ (on utilisera donc $i_{max} < i$ comme condition d'arrêt).

FIGURE 8 – Script python pour déterminer le niveau l d'une valeur d'indice i

```
while True:
    indice = int(input("Indice : "))
    level = 1

    while 0.5 * level * (level + 1) - 1 < indice:
        level += 1

    print(level - 1)
```

Calcul de la position p En suivant la remarque du paragraphe précédent, pour déterminer la position p d'une valeur d'indice i, il suffit de récupérer i_max, qui sera la m-ème valeur de la ligne, et de faire la différence avec i. Une fois cette différence trouvé (notée Δi), on sait que sur un niveau m, il y a $m + 1$ valeurs, alors la position p de la valeur i sera donnée par $p = m - \Delta i$

FIGURE 9 – Algorithme pour déterminer le niveau l et la position p d'une valeur d'indice i

```
while True:
    i = int(input("Indice : "))

    l = 1
    while 0.5 * l * (l + 1) - 1 < i:
        l += 1

    i_max = 0.5 * l * (l + 1) - 1

    print(f"- l'indice {i} est sur le niveau l = {l - 1}")
    print(f"- le niveau {l - 1} admet i_max = {i_max}")

    diff = i_max - i
    print(f"- l'indice {i} est en position {l - 1 - diff}")
```

Calcul de $g(i)$ et $d(i)$ Nous savons désormais la position p et le niveau l de la valeur à 'indice i ainsi que l'indice i_max de la dernière valeur du niveau l. Il est évident que $i_max + 1$ est la première valeur du niveau $l + 1$. De plus, on sait que le descendant sera de position p dans le niveau $l + 1$, ce qui nous fait écrire que

$$g(i) = i_{max} + 1 + p$$

Évidemment, il suit que le descendant à droite $d(i)$ sera calculé de la façon suivante

$$d(i) = g(i) + 1$$

Stratégie

Optimisation locale Pour l'optimisation locale, en partant du haut du triangle de niveau m, il suffit de choisir le minimum entre le descendant à droite et le descendant à gauche et ce à chaque étape jusqu'à arriver au niveau $m - 1$.

Optimisation globale La programmation dynamique s'applique particulièrement bien dans ce problème. On suppose le problème résolu, on est alors au niveau $m - 1$. Soit T le tableau contenant les données du triangle.

Si on est sur une feuille de l'abre on aura

$$m(0) = T[0]$$

Sinon, on aura alors

$$m(0) = \max(m(g(0)), m(d(0))) + T[0]$$

En généralisant, on aura pour tout ce qui n'est pas une feuille de l'arbre l'équation de récurrence suivante

$$m(i) = max(m(g(i)), m(d(i))) + T[i]$$

On implémente alors tout cela dans le code.

Analyse statistique

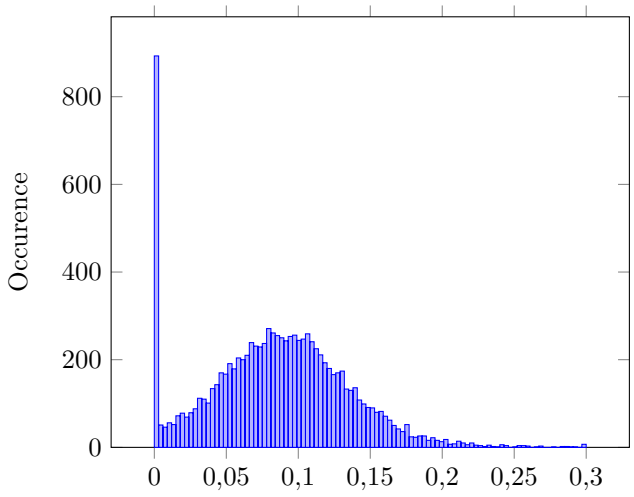

FIGURE 10 – Différence relative entre chemin optimal et naïf pour le problème du chemin maximum dans un triangle

Stratégie gloutonne	Moyenne	Médiane	Variance	Écart type
Par choix du noeud maximum	0.084741	0.085569	0.0024415	0.049411

TABLE 5 – Statistiques pour le problème de répartition optimale d'un stock

6 Conclusion

Dans ce rapport nous avons pu constater l'étendu de l'écartement des stratégies d'optimisation locale par rapport aux stratégies d'optimisation globale. Avoir une analyse statistique de ces problèmes permet de faire des choix cohérents avec le but cherché : savoir le degré de précision que l'on veut atteindre permet de choisir en connaissance de cause l'approche qu'on utilisera. Certaines différences sont énormes, d'autres beaucoup plus proche, montrant que toutes les stratégies d'optimisation locales ne se valent pas.

Table des figures

Liste des tableaux

A Codes

Ici sont présentés les codes utilisés dans le cadre de ce projet

A.1 Utilitaires

Utils.java

```java
/**
 * Code written in the framework of the IGI-2102 unit by Corentin Poupry
↪  (corentin.poupry@edu.esiee.fr) and Neo Jonas
 * (neo.jonas@edu.esiee.fr). All rights reserved.
 *
 * Created with Java 18
 **/
import java.io.FileWriter;

public class Utils {
    private static final String SEPARATOR = "\n";

    /**
     * Export data under the requested format in order to generate graph
     * @param name name of the file to create
     * @param data data to be used in the file
     */
    public static void export_data(String name, double[] data) {
        try {
            FileWriter file = new FileWriter("./data/%s.csv".formatted(name));

            for (double datum : data) {
                file.append(Double.toString(datum));
                file.append(SEPARATOR);
            }

            file.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    /**
     * Print the results of the strategies
     * @param g result of the greedy strategy
     * @param v result of the optimised strategy
     */
    public static void print_result(int g, int v) {
        System.out.println();
        System.out.println("RESULT");
        System.out.printf("naive : %d", g);
        System.out.println();
```

```
        System.out.printf("optimised : %d", v);
        System.out.println();
    }
}
```

stats.m

```
PATH = "./data";

args = argv();
FILE = args{1};

data = csvread(fullfile(PATH, FILE));

format short g

disp("File"), disp(FILE)
disp("Moyenne "), disp(mean(data))
disp("Médiante "), disp(median(data))
disp("Variance "), disp(var(data))
disp("Ecart type"), disp(std(data))
```

A.2 Code pour le chemin minimum du robot

MinimumPathRobot.java

```
/**
 * Problem of the robot minimum path
 *
 * Derived from the basic code provided by René Natowicz (rene.natowicz@esiee.fr)
 * Code written in the framework of the IGI-2102 unit by Corentin Poupry
↪  (corentin.poupry@edu.esiee.fr) and Neo Jonas
 * (neo.jonas@edu.esiee.fr). All rights reserved.
 *
 * Created with Java 18
 **/

import java.util.Random;
import java.util.concurrent.ThreadLocalRandom;
import java.util.concurrent.TimeUnit;

public class MinimumPathRobot {
    /**
     * Maximum width of the grid
     */
    public static final int LMAX = 1000;

    /**
```

```java
     * Maximum height of the grid
     */
    public static final int CMAX = 1000;

    /**
     * Maximum value for a grid element
     */
    public static final int VMAX = 200;

    public static void main(String[] args) {
        var data = launch(10_000);
        Utils.export_data("minimum_path_robot", data);
    }

    /**
     * Launch the different strategies and review the results data
     * @param run_limit number of runs to make
     * @return the relative distances
     */
    static double[] launch(int run_limit) {
        // we use a thread local PRNG to ensure that there will be no unnecessary
↪   instantiation & allocations
        Random rand = ThreadLocalRandom.current();
        double[] data = new double[run_limit];

        long start_time = System.nanoTime();

        for (int run = 0; run < run_limit; run++) {
            System.out.printf("--- Run number #%d ---", run + 1);
            System.out.println();

            // let's generate the dimensions of the problem grid at random
            // one is added so that the grid cannot be zero sized
            int L = rand.nextInt(LMAX) + 1;
            int C = rand.nextInt(CMAX) + 1;

            System.out.printf("Grid dimension: %d x %d", L, C);
            System.out.println();

            // we generate our moving cost matrices (or grid)
            int[][] N = generateGrid(L, C);
            int[][] E = generateGrid(L, C);
            int[][] NE = generateGrid(L, C);

            // apply the naive way
            int g = glouton(N, E, NE);

            // apply optimised strategy
            int[][] M = calculerM(N, E, NE);
            int v = M[L - 1][C - 1];
```

```java
            Utils.print_result(g, v);

            data[run] = v == 0 ? 0 : (double) (g - v) / (double) v;
        }

        long elapsed_time = System.nanoTime() - start_time;

        System.out.println();
        System.out.printf("Elapsed time: %fms", (double)
↪   TimeUnit.NANOSECONDS.toMillis(elapsed_time));

        return data;
    }


    /**
     * Apply the naive strategy
     *
     * @param north_grid     northbound movement grid
     * @param east_grid      eastward movement grid
     * @param northeast_grid north-eastward movement grid
     * @return the total cost of getting from (0, 0) to (l - 1, c - 1)
     */
    public static int glouton(int[][] north_grid, int[][] east_grid, int[][]
↪   northeast_grid) {
        int total = 0;

        int l = 0, c = 0;
        int L = north_grid.length, C = north_grid[0].length;

        while (l < L - 1 || c < C - 1) {
            // We calculate in which direction it is less expensive to go
            int n_cost = N(l, c, L, C, north_grid);
            int e_cost = E(l, c, L, C, east_grid);
            int ne_cost = NE(l, c, L, C, northeast_grid);

            int min_cost = min(n_cost, e_cost, ne_cost);

            if (min_cost == n_cost) {
                l++; // go to North
            } else if (min_cost == e_cost) {
                c++; // go to East
            } else {
                l++; // go to North...
                c++; // ...and East
            }

            // add the cost of moving
            total += min_cost;
        }
```

```
        return total;
    }

    /**
     * Apply the optimised strategy
     *
     * @param north_grid     northbound movement grid
     * @param east_grid      eastward movement grid
     * @param northeast_grid north-eastward movement grid
     * @return the problem-solving matrix
     */
    public static int[][] calculerM(int[][] north_grid, int[][] east_grid,
↪   int[][] northeast_grid) {
        int L = north_grid.length;
        int C = north_grid[0].length;

        int[][] M = new int[L][C]; // de terme général M[l][c] = m(l,c)
        // base
        M[0][0] = 0;
        for (int c = 1; c < C; c++) M[0][c] = M[0][c - 1] + E(0, c - 1, L, C,
↪   east_grid);
        for (int l = 1; l < L; l++) M[l][0] = M[l - 1][0] + N(l - 1, 0, L, C,
↪   north_grid);

        // cas général
        for (int c = 1; c < C; c++)
            for (int l = 1; l < L; l++)
                M[l][c] = min(
                        M[l][c - 1] + E(l, c - 1, L, C, east_grid),
                        M[l - 1][c] + N(l - 1, c, L, C, north_grid),
                        M[l - 1][c - 1] + NE(l - 1, c - 1, L, C, northeast_grid)
                );

        return M;
    }

    /**
     * Generate a L-C grid as a matrix L x C with random values
     *
     * @param L how many L cells the matrix should have
     * @param C how many C cells the matrix should have
     * @return the grid created
     */
    public static int[][] generateGrid(int L, int C) {
        // we use a thread local PRNG to ensure that there will be no unnecessary
↪   instantiation & allocations
        Random rand = ThreadLocalRandom.current();

        // Matrix M(L, C) as our grid
        int[][] grid = new int[L][C];
```

```
        for (int i = 0; i < L; i++) {
            for (int j = 0; j < C; j++) {
                grid[i][j] = rand.nextInt(VMAX);
            }
        }

        return grid;
    }

    /**
     * Minimum between three value
     *
     * @param a first value
     * @param b second value
     * @param c third value
     * @return the minimum
     */
    public static int min(int a, int b, int c) {
        return Math.min(Math.min(a, b), c);
    }

    /**
     * Cost of moving the robot to the East of its position
     *
     * @param l current height coordinate
     * @param c current width coordinate
     * @param L dimension of the height of the grid
     * @param C dimension of the width of the grid
     * @param S the cost grid
     * @return the cost of moving
     */
    private static int E(int l, int c, int L, int C, int[][] S) {
        // prevent out of bounds
        if (c + 1 >= C) return Integer.MAX_VALUE;
        return S[l][c + 1];
    }

    /**
     * Cost of moving the robot to the North of its position
     *
     * @param l current height coordinate
     * @param c current width coordinate
     * @param L dimension of the height of the grid
     * @param C dimension of the width of the grid
     * @param S the cost grid
     * @return the cost of moving
     */
    private static int N(int l, int c, int L, int C, int[][] S) {
        // prevent out of bounds
        if (l + 1 >= L) return Integer.MAX_VALUE;
        return S[l + 1][c];
```

```
    }

    /**
     * Cost of moving the robot to the North East of its position
     *
     * @param l current height coordinate
     * @param c current width coordinate
     * @param L dimension of the height of the grid
     * @param C dimension of the width of the grid
     * @param S the cost grid
     * @return the cost of moving
     */
    private static int NE(int l, int c, int L, int C, int[][] S) {
        // prevent out of bounds
        if (l + 1 >= L || c + 1 >= C) return Integer.MAX_VALUE;
        return S[l + 1][c + 1];
    }
}
```

A.3 Code pour le sac de valeur maximale

MaximumValueBag.java

```
/**
 * Problem of the maximum value of a bag
 *
 * Derived from the basic code provided by René Natowicz (rene.natowicz@esiee.fr)
 * Code written in the framework of the IGI-2102 unit by Corentin Poupry
↪  (corentin.poupry@edu.esiee.fr) and Neo Jonas
 * (neo.jonas@edu.esiee.fr). All rights reserved.
 *
 * Created with Java 18
 **/

import java.util.Arrays;
import java.util.Comparator;
import java.util.Random;
import java.util.concurrent.ThreadLocalRandom;
import java.util.concurrent.TimeUnit;

public class MaximumValueBag {
    /**
     * maximum number of objects
     */
    public static final int NMAX = 100;

    /**
     * maximum capacity of the bag
```

```
     */
    public static final int CMAX = 1000;

    /**
     * maximum value of one object
     */
    public static final int VMAX = 100;

    /**
     * maximum size of one object
     */
    public static final int SMAX = 50;

    public static void main(String[] args) {
        var data_ratio = launch(5000, GloutonStrategy.BY_RATIO);
        Utils.export_data("maximum_value_bag_ratio", data_ratio);

        var data_value = launch(5000, GloutonStrategy.BY_VALUE);
        Utils.export_data("maximum_value_bag_value", data_value);

        var data_size = launch(5000, GloutonStrategy.BY_SIZE);
        Utils.export_data("maximum_value_bag_size", data_size);
    }

    /**
     * Launch the different strategies and review the results data
     * @param run_limit number of runs to make
     * @return the relative distances
     */
    static double[] launch(int run_limit, GloutonStrategy strategy) {
        // we use a thread local PRNG to ensure that there will be no unnecessary
↪   instantiation & allocations
        Random rand = ThreadLocalRandom.current();
        double[] data = new double[run_limit];

        long start_time = System.nanoTime();

        for (int run = 0; run < run_limit; run++) {
            System.out.printf("--- Run number #%d ---", run + 1);
            System.out.println();

            int c = rand.nextInt(CMAX) + 1;
            System.out.printf("Capacity of the bag: %d", c);
            System.out.println();

            // 20 <= Number of objects <= 100
            int n = 20 + rand.nextInt(NMAX - 20) + 1;
            System.out.printf("Number of objects: %d", n);
            System.out.println();

            // create our list of objects
```

```
            BagObject[] objects = new BagObject[n];

            // fill it
            for (int i = 0; i < n; i++) {
                objects[i] = BagObject.CreateRandomObject();
            }

            Comparator<BagObject> comparator = null;

            if (strategy == GloutonStrategy.BY_RATIO) comparator =
↪   Comparator.comparing(BagObject::ratio).reversed();
            if (strategy == GloutonStrategy.BY_VALUE) comparator =
↪   Comparator.comparing(BagObject::value).reversed();
            if (strategy == GloutonStrategy.BY_SIZE) comparator =
↪   Comparator.comparing(BagObject::size).reversed();

            // applies the naive strategy to sort objects
            int g = glouton(objects, c, comparator);

            // apply optimised strategy
            int[][] M = calculerM(objects, c);
            int v = M[n][c];

            Utils.print_result(g, v);

            data[run] = v == 0 ? 0 : (double) (v - g) / (double) v;
        }

        long elapsed_time = System.nanoTime() - start_time;

        System.out.println();
        System.out.printf("Elapsed time: %fms", (double)
↪   TimeUnit.NANOSECONDS.toMillis(elapsed_time));

        return data;
    }

    /**
     * Apply the optimised strategy
     * @param objects objects that can be chosen
     * @param C max capacity of the bag
     * @return the problem-solving matrix
     */
    static int[][] calculerM(BagObject[] objects, int C) {
        int n = objects.length;
        // Retourne M[0:n+1][0:C+1], de terme général M[k][c] = m(k,c)
        int[][] M = new int[n + 1][C + 1];

        // Base : m(0,c) = 0 pour toute contenance c, 0 <= c < C+1
        for (int c = 0; c < C + 1; c++) M[0][c] = 0;
```

```
        // Cas général, pour tous k et c, 1 <= k < n+1, 0 <= c < C+1,
        // m(k,c) = max(M[k-1][c], V[k-1] + M[k-1][c-T[k-1]])
        for (int k = 1; k < n + 1; k++) {
            for (int c = 0; c < C + 1; c++) {
                // calcul et mémorisation de m(k,c)
                if (c - objects[k - 1].size < 0) // le k-ème objet est trop gros
↪   pour entrer dans le sac
                    M[k][c] = M[k - 1][c];
                else
                    M[k][c] = Math.max(M[k - 1][c], objects[k - 1].value + M[k -
↪   1][c - objects[k - 1].size]);
            }
        }

        return M;
    }

    /**
     * Apply the greedy strategy
     * @param objects  list of possible objects
     * @param capacity bag capacity limit
     * @return the sum of the values of the items in the bag
     */
    static int glouton(BagObject[] objects, int capacity, Comparator<BagObject>
↪   comparator) {
        // sort the objects in descending order of ratio (from the most to the
↪   least interesting object to take)
        Arrays.sort(objects, comparator);

        int sum = 0;

        for (BagObject object : objects) {
            // the object is too big for the actual capacity
            if (capacity < object.size) continue;

            capacity -= object.size;
            sum += object.value;
        }

        return sum;
    }

    /**
     * Existing naive strategies
     */
    enum GloutonStrategy {
        /**
         * Compare objects by descending ratio
         */
        BY_RATIO,
```

```
        /**
         * Compare objects in descending order of value
         */
        BY_VALUE,

        /**
         * Compare objects in descending order of size
         */
        BY_SIZE
    }

    /**
     * Represents an object used in the problem
     *
     * @param size  Size of the object
     * @param value Value of the object
     */
    public record BagObject(int size, int value) {
        /**
         * Creates an object whose size and value are random
         *
         * @return the randomly created object
         */
        static BagObject CreateRandomObject() {
            Random rand = ThreadLocalRandom.current();

            int size = rand.nextInt(SMAX) + 1;
            int value = rand.nextInt(VMAX) + 1;

            return new BagObject(size, value);
        }

        /**
         * Gives the ratio of the object, calculated by dividing the value by the
↪   size
         *
         * @return the ratio of the object
         */
        public float ratio() {
            return (float) this.value / (float) this.size;
        }

        @Override
        public String toString() {
            return "BagObject{size=%d, value=%d, ratio=%f}".formatted(size,
↪   value, this.ratio());
        }
    }
}
```

A.4 Code pour la répartition optimale d'un stock

OptimalWarehouse.java

```java
/**
 * Problem of the optimal distribution of a stock on warehouses
 *
 * Derived from the basic code provided by René Natowicz (rene.natowicz@esiee.fr)
 * Code written in the framework of the IGI-2102 unit by Corentin Poupry
↪  (corentin.poupry@edu.esiee.fr) and Neo Jonas
 * (neo.jonas@edu.esiee.fr). All rights reserved.
 *
 * Created with Java 18
 **/

import java.util.Random;
import java.util.concurrent.ThreadLocalRandom;
import java.util.concurrent.TimeUnit;

public class OptimalWarehouse {
    /**
     * Maximum value of warehouses
     */
    public static final int WMAX = 50;

    /**
     * Maximum value of the stock
     */
    public static final int SMAX = 50;

    public static void main(String[] args) {
        var data = launch(10_000);
        Utils.export_data("optimal_warehouse", data);
    }

    /**
     * Launch the different strategies and review the results data
     * @param run_limit number of runs to make
     * @return the relative distances
     */
    static double[] launch(int run_limit) {
        // we use a thread local PRNG to ensure that there will be no unnecessary
↪  instantiation & allocations
        Random rand = ThreadLocalRandom.current();
        double[] data = new double[run_limit];

        long start_time = System.nanoTime();

        for (int run = 0; run < run_limit; run++) {
            System.out.printf("--- Run number #%d ---", run + 1);
            System.out.println();
```

```
            int stock = rand.nextInt(SMAX) + 1;
            System.out.printf("Stock: %d", stock);
            System.out.println();

            int warehouses = rand.nextInt(WMAX) + 1;
            System.out.printf("Number of warehouses: %d", warehouses);
            System.out.println();

            int[][] gains = generateGain(warehouses, stock);

            int[][] M = calculerMA(gains);
            var v = M[warehouses][stock];

            int g = glouton(gains, stock);

            Utils.print_result(g, v);

            data[run] = v == 0 ? 0 : (double) (v - g) / (double) v;
        }

        long elapsed_time = System.nanoTime() - start_time;

        System.out.println();
        System.out.printf("Elapsed time: %fms", (double)
↪   TimeUnit.NANOSECONDS.toMillis(elapsed_time));

        return data;
    }

    static int[][] calculerMA(int[][] G) {
        // G[0:n][0:S+1] de terme général
        // G[i][s] = gain d'une livraison d'un stock s à l'entrepôt i.

        // Calcule : M[0:n+1][0:S+1] de tg M[k][s] = m(k,s) et A = arg M.
        int n = G.length;
        int S = G[0].length - 1;
        int[][] M = new int[n + 1][S + 1];

        // base: m(0, s) = 0
        for (int s = 0; s < S + 1; s++)
            M[0][s] = 0;

        // cas général
        for (int k = 1; k < n + 1; k++) {
            for (int s = 0; s < S + 1; s++) {
                for (int sk = 0; sk < s + 1; sk++) {
                    int mks = G[k - 1][sk] + M[k - 1][s - sk];

                    if (mks > M[k][s]) M[k][s] = mks;
                }
```

```
            }
        }

        return M;
    }

    /**
     * Apply the greedy strategy
     * @param gains values of the gains for the warehouses
     * @param stock max stock usable
     * @return the resolution matrix
     */
    private static int glouton(int[][] gains, int stock) {
        int warehouses = gains.length;
        int[] stock_allocation = new int[warehouses];

        while (stock > 0) {
            int max = 0;
            int index = 0;

            // we find the most advantageous gain to work with by looking at the
            // difference between the stock with s+1 stock and s stock
            for (int i = 0; i < warehouses; i++) {
                int allocated = stock_allocation[i];
                int diff = gains[i][allocated + 1] - gains[i][allocated];

                if (diff > max) {
                    max = diff;
                    index = i;
                }
            }

            // a new stock is allocated to the most interesting unit
            stock_allocation[index] += 1;
            // our quota of stock is reduced by one
            stock -= 1;
        }

        int sum = 0;
        for (int i = 0; i < warehouses; i++) {
            int s = stock_allocation[i];
            sum += gains[i][s];
        }

        return sum;
    }

    /**
     * Generate random gain on warehouses according to the stock they have
     * This procedure is practically the same as for the optimal planning problem
     * @param warehouses number of warehouses
```

```
     * @param stock_max maximum number of stock
     * @return a 2D table representing the warehouses and the nested table, the
↪  gain that can
     * be expected with a stock corresponding to the index
     */
    static public int[][] generateGain(int warehouses, int stock_max) {
        Random rand = ThreadLocalRandom.current();

        // G[i][h] = g(i,h). Les estimations sont aléatoires, croissantes selon
↪  h.
        int[][] G = new int[warehouses][stock_max + 1];

        for (int i = 0; i < warehouses; i++) G[i][0] = 0;
        for (int i = 0; i < warehouses; i++)
            for (int h = 1; h <= stock_max; h++)
                G[i][h] = G[i][h - 1] + rand.nextInt(6);

        return G;
    }
}
```

A.5 Code pour le chemin maximum dans un triangle

MaximumPathTriangle.java

```
/**
 * Problem of the optimal distribution of a stock on warehouses
 *
 * Code written in the framework of the IGI-2102 unit by Corentin Poupry
↪  (corentin.poupry@edu.esiee.fr) and Neo Jonas
 * (neo.jonas@edu.esiee.fr). All rights reserved.
 *
 * Created with Java 18
 **/

import java.util.Random;
import java.util.concurrent.ThreadLocalRandom;
import java.util.concurrent.TimeUnit;


public class MaximumPathTriangle {
    /**
     * Maximum number of levels
     */
    public static final int LMAX = 100;

    /**
     * Maximum value for data
```

```java
     */
    public static final int VMAX = 50;

    public static void main(String[] args) {
        double[] data = launch(5_000 * 2);
        Utils.export_data("maximum_path_triangle", data);
    }

    /**
     * Launch the different strategies and review the results data
     * @param run_limit number of runs to make
     * @return the relative distances
     */
    public static double[] launch(int run_limit) {
        // we use a thread local PRNG to ensure that there will be no unnecessary
↪   instantiation & allocations
        Random rand = ThreadLocalRandom.current();
        double[] data = new double[run_limit];

        long start_time = System.nanoTime();

        for (int run = 0; run < run_limit; run++) {
            System.out.printf("--- Run number #%d ---", run + 1);
            System.out.println();

            int l = rand.nextInt(LMAX) + 1;
            System.out.printf("Levels: %d", l);
            System.out.println();

            // calculation of the number of values to fill all the levels
            int values = (int)(0.5 * l * (l + 1));
            System.out.printf("Number of values: %d", values);

            // triangle's data
            int[] T = new int[values];

            for (int i = 0; i < values; i++) {
                T[i] = rand.nextInt(VMAX);
            }

            // applies the naive strategy to sort objects
            int g = glouton(T, l);

            // apply optimised strategy
            int[] M = calculerM(T);
            int v = M[0];

            Utils.print_result(g, v);

            data[run] = v == 0 ? 0 : (double) (v - g) / (double) v;
        }
```

```java
        long elapsed_time = System.nanoTime() - start_time;

        System.out.println();
        System.out.printf("Elapsed time: %fms", (double)
↪   TimeUnit.NANOSECONDS.toMillis(elapsed_time));

        return data;
    }

    /**
     * Apply the greedy strategy
     * @param data values of the triangle
     * @param max_level levels that the triangle has
     * @return sum of the maximum path taken
     */
    public static int glouton(int[] data, int max_level) {
        int i = 0;
        // the first node is always part of the path
        int sum = data[0];

        // we will have a choice to make at each level before hitting the bottom
        for (int j = 0; j < max_level - 1; j++) {
            int i_left = g(i);
            int i_right = d(i);

            // we choose the minimum between the two descendants
            int max = Math.max(data[i_left], data[i_right]);

            // we give the index its new value
            if (max == data[i_left]) i = i_left;
            if (max == data[i_right]) i = i_right;

            // we add the weight of the node
            sum += max;
        }

        return sum;
    }

    /**
     * Apply the optimal strategy
     * @param T triangle's data
     * @return the resolution array
     */
    public static int[] calculerM(int[] T){
        int[] M = new int[T.length];

        // On commence par le bas du triangle pour ensuite remonter
        for(int i = T.length - 1; i >= 0; i--){
            // On regarde si on sort du triangle.
```

```
            // Si oui, on est alors sur une feuille
            if(g(i) >= T.length){
                M[i] = T[i];
            } else {
                // Sinon on est sur un nœud qui n'est pas une feuille
                // il faut alors prendre en compte le sous-problème de la somme
↪   du triangle de
                // gauche et de celle du triangle de droite auquel on
↪   additionnera le poids du nœud actuel.
                // On sélectionnera le plus grand sous-problème
                // m(i) = max(m(g(i)), m(d(i))) + T[i]
                M[i] = Math.max(M[g(i)], M[d(i)]) + T[i];
            }
        }

        return M;
    }

    /**
     * Return the index of the left descendant of the parent index
     * @param i parent index
     * @return left descendant index
     */
    public static int g(int i) {
        // we determine the level at which the value of index i is
        int l = 1;
        while (0.5 * l * (l + 1) - 1 < i) l++;

        // then we determine the index i_max of the last value of the level l
        int i_max = (int)(0.5 * l * (l + 1) - 1);

        // we calculate the relative distance between i_max and our index i
        // this will give us the relative position p at the end of the level l
        int diff = i_max - i;

        // we know that on the line l there are l + 1 values, and we know the
↪   relative position to the last value.
        // We calculate the positioning p index
        int p = l - 1 - diff;

        // i_max + 1 is the index of the first element of the level l + 1 and p
↪   its position in the line,
        // we add them together to obtain the new index
        return i_max + 1 + p;
    }

    /**
     * Return the index of the right descendant of the parent index
     * @param i parent index
     * @return right descendant index
     */
```

```
    public static int d(int i) {
        return g(i) + 1;
    }
}
```