IT45 : Rapport de Développement

Problèmes d'affectation, de planification et de routage des tournées des employés

Garbriel SCHWAB, Cyrille STROESSER

Printemps 2022

Contents

1 Introduction

L'objectif de ce projet d'IT45 vise à résoudre un problème de recherche opérationnelle concret. Celui-ci à pour but d'affecter une liste de missions commandée à un SESSAD à ses employés de façon à optimiser plusieurs critères tout en respectant différentes contraintes. Les trois critères principaux d'optimisation son :

1. L'équité et l'égalité entre les employés en harmonisant la charge de travail et la distance parcourue.
2. Les préferences des apprenants en minimisant le nombre des affectations dont la spécialité est insatisfaite.
3. L'objectif du centre SESSAD en minimisant le nombre d'heures supplementaires et perdues et la distance totale parcourue.

Pour résoudre ce problème, nous avons implémenté un algorithme génétique optimisé en casacade. Pour ce faire, il faut d'abord générer une population initiale avec des solutions valides et hétérogènes. Puis, il faut générer de nouvelles populations avec des croisements et des mutations pour en sélectionner les meilleurs individus selon le premier critère d'optimisation. Après un certain nombre d'itération, on sélectionne les meilleurs individus selon le deuxième critère d'optimisation et finalement on choisit le meilleur individu selon le derniers critère. Ensuite, il s'agit de trouver les meilleurs paramètres (taux de croisements, de mutations, méthode de séléction, population, itération,...) pour trouver la meilleure solution possible.

2 Modélisation

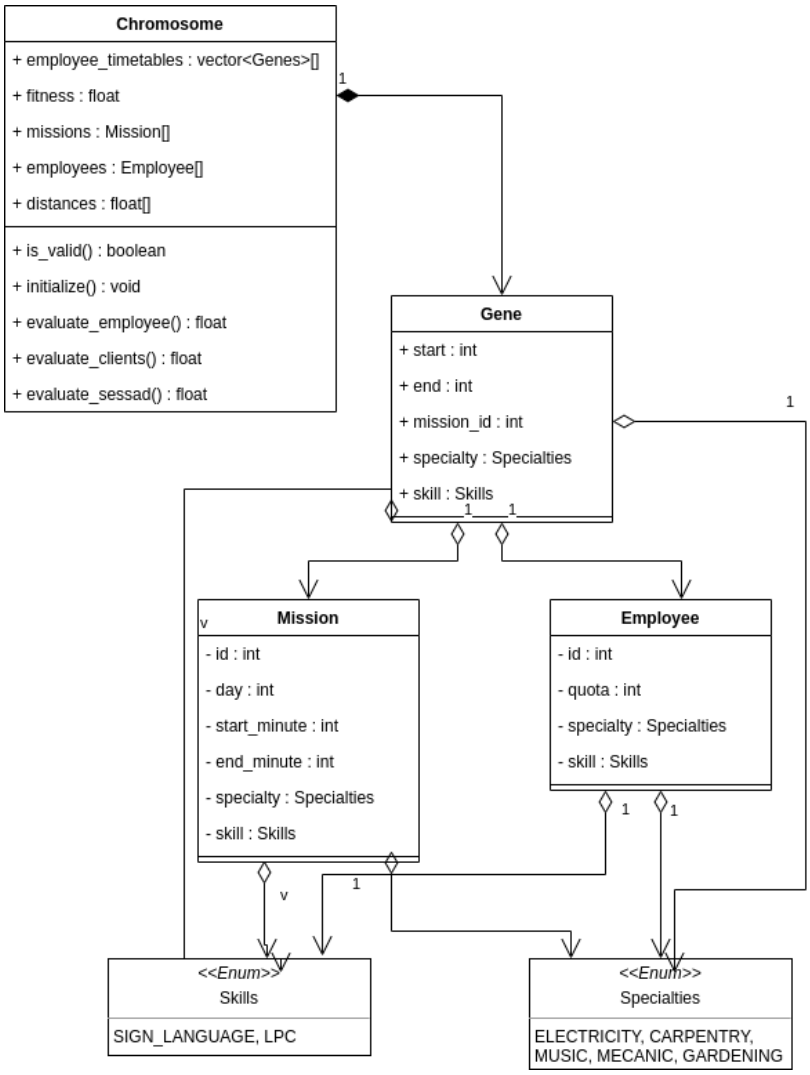

Figure 1: Digramme de classe

3 Implémentation

Nous avons choisi d'implémenter notre solution en C++ par souci de performance. En effet, ce langage permet une gestion de la mémoire bas niveau. Cela nous a permis d'optimiser au mieux la mémoire utilisée. En outre, nous maîtrisions déjà ce langage.

3.1 Structure de données

Afin d'implémenter la modélisation détaillée ci-dessus, nous avons choisi différentes structures de données :

- ***Missions et Employés*** : après avoir extrait les instances des fichiers *csv*, les missions et intervenants (employees) sont stockées dans les structures suivantes :

```
typedef struct
{
    int id;
    int day;
    int start_minute;
    int end_minute;
    Specialties specialty;
    Skills skill;
} Mission;
typedef struct
{
    int id;
    int quota;
    Specialties specialty;
    Skills skill;
} Employee;
```

- ***Matrice des distances***: Quant à la matrice des distances, bien que ce soit une matrice à deux dimensions, il est possible de la stocker dans un tableau à une dimension. Pour accéder à l'élément de la ligne i et de la colonne j, on utilise le code suivant :

```
// j = matrix size
distances[i*n_location + j];
```

  Nous avons fait ce choix par souci d'optimisation, car parcourir un tableau à une dimension est nettement plus rapide pour un ordinateur que parcourir un tableau à deux dimensions.

- ***Gène***: Un gène correspond à une affectation d'une mission sur un emploi du temps d'un employé. Il comprend toutes les informations nécessaires à l'optimisation de la solution (horaires, spécialité et compétence).

```
typedef struct
{
    int start;
    int end;
    int mission_id;
    Specialties specialty;
    Skills skill;
} Gene;
```

- ***Chromosome*** : les solutions, aussi appelées chromosomes, sont stockées dans une classe car leur manipulation par l'algorithme génétique rend l'utilisation de méthodes pertinente. Un chromosome est composé d'un objectif, d'un pointeur sur le tableau d'employés, d'un pointeur sur le tableau de missions et d'un pointeur sur la matrice des distances. Il est aussi composé d'un tableau de listes, *std* :: *vector*, stockant les emplois du temps de tous les employés.

```
class Chromosome
{
public:
    float fitness;
    const Mission *missions;
    const Employee *employees;
```

```
    const float *distances;
    std::vector<Gene> *employee_timetables;
}
```

De la même manière que la matrice des distances, *employee_timetables* est un tableau à deux dimensions, (jour de la semaine **x** nombre d'employés) ramené à une dimension. Pour accéder au jour j de l'employé k, on utilise le code suivant :

```
// N_WEEK_DAY = nombre de jours de le semaine
employee_timetables[k * N_WEEK_DAY + j];
```

On accède alors à une liste de gène correspondants aux différents missions affectées à l'employées ce jour-ci. Cette liste est toujours triée dans l'ordre croissant pour former un emploi du temps.

3.2 Population initiale

Notre algorithme de génération de la solution initiale ne nécessite pas d'assouplissement de contrainte car il génére des solutions respectant toutes les contraintes strictement. Pour ce faire nous avons implémenter l'algorithme **first-fit**. Pour chaque mission, il suffit de l'affecter au premier jour du premier employé qui respecte toutes les contraintes. La solution va alors dépendre de l'ordre dans lequel est donné les missions ou de l'ordre dans lequel est donné les employés. Ainsi, pour pouvoir générer pleins de solutions initiales hétérogènes, il nous suffit de mélanger le tableau des missions et des employés avant de les passer dans l'algorithme de génération. Pour ce faire, on utilise la fonction suivante suivante :

```
void initialize_population(Chromosome *population, const Mission missions_p[], const
    Employee employees_p[], const float distances_p[])
{
    population[0] = Chromosome(missions_p, employees_p, distances_p);
    population[0].initialize();

    for (int i = 1; i < population_size; ++i)
    {
        Mission missions[n_mission];
        Employee employees[n_employee];
        std::copy(missions_p, missions_p + n_mission, missions);
        std::copy(employees_p, employees_p + n_employee, employees);
        std::shuffle(missions, missions + n_mission, std::default_random_engine(i));
        std::shuffle(employees, employees + n_employee, std::default_random_engine(i))
            ;
        population[i] = Chromosome(missions, employees, distances_p);
        population[i].initialize();
    }
}
```

L'objectif principal est d'avoir une population initiale composée de différentes solutions, diversifiées et de qualités. Il est important d'avoir un bon équilibre entre ces deux critères. Si nous avons une population trop diversifiée et très peu optimale, l'algorithme risque de ne pas être très efficace et de prendre trop de temps à converger. A l'inverse, si la population initiale est très optimale mais peu diversifiée, le risque est de tomber dans des optimums locaux

3.3 Algorithme génétique

3.3.1 Fitness

Nous avons repris les fonction d'évaluation proposées dans le sujet. Nous allons opter pour une résolution en cascade, c'est à dire que nous allons faire tourner notre algorithme avec la première fitness, puis filtrer les solutions obtenues avec les deux autres fonctions. Voici les fonctions d'évaluation que nous allons utiliser :

$$f_{employees} = \frac{\zeta \cdot \sigma_{WH}(s) + \gamma \cdot \sigma_{OH}(s) + \kappa \cdot \sigma_D(s)}{3}$$

$$f_{clients} = \alpha \cdot penalties(s)$$

$$f_{SESSAD} = \frac{\beta \cdot sumWOH(s) + \kappa \cdot moyD(s) + \kappa \cdot maxD(s)}{3}$$

avec :
– $\sigma_{WH}(s)$ = écart type des heures non-travaillées des employés pour s
– $\sigma_{OH}(s)$ = écart type des heures supplémentaires des employés pour s
- $\sigma_D(s)$ = écart type des distances des employés pour s
- $penalties(s)$ = nombre d'affectation dont la spécialité est insatisfaite
- $\sigma_{WH}(s)$ = somme des heures non-travaillées et supplémentaires de tous les employés pour s
- $moyD(s)$ = distance moyenne parcourue par les employés pour s
- $maxD(s)$ = distance maximale parcourue par les employés pour s
- $\alpha, \beta, \gamma, \zeta$, sont des facteurs de corrélation.

3.3.2 Sélection

Pour sélectionner les meilleurs chromosomes destinés à être se reproduire entre eux, nous avons choisi d'utiliser une méthode de sélection proportionnelle, la sélection par roulette. Elle choisi les meilleurs chromosome ayant les meilleurs fitness. Ce choix nécessite d'avoir une population de départ très diversifiée pour ne pas rester bloqué dans des optimums locaux. Comme nous sommes dans le cas d'une minimisation d'objectif, la probabilité d'être séléctionner est donnée par la formule suivante : $P = \frac{\frac{1}{F_p}}{\sum_i \frac{1}{F_i}}$, F_p la fitness de l'individu
Nous l'avons implémenté de la manière suivante :

```
Chromosome *roulette_selection(Chromosome population[], std::default_random_engine &
    generator)
{
    float proba_sum = 0;
    float fitness_sum = 0;
    float proba_array[population_size] = {0};
    std::uniform_real_distribution<float> uniform_dist(0, 1);

    for (int i = 0; i < population_size; ++i){
        fitness_sum += 1 / population[i].evaluate_employees();
    }
    for (int i = 0; i < population_size; ++i){
        proba_array[i] = proba_sum + (1 / population[i].fitness) / fitness_sum;
        proba_sum += (1 / population[i].fitness) / fitness_sum;
    }

    size_t index = 0;
    float random = uniform_dist(generator);
    while (proba_array[index] < random)
        index++;

    return &population[index];
}
```

3.3.3 Croisement

Dans un premier temps, afin d'obtenir les meilleurs résultats et performances il faudra faire varier le taux de croisement, correspondant à la probabilité qu'un croisement ait lieu entre deux chromsomes au sein d'une même population. Ce dernier ce trouve dans l'intervalle $[0.2, 0.8]$.

Nous avons testé plusieurs opérateurs de croisement mais nous en avons gardé seulement 2 :

- ***Day 1X*** : cet opérateur sépare le cromosome en un point et l'enfant reçoit la première partie du premier parent et la deuxième partie du deuxième parent. Pour notre problème, la séparation en 2 se fait au milieu de la semaine. C'est à dire que lorsqu'on résonne en terme d'emploi du temps, l'enfant va recevoir toutes les affectations de missions du Lundi et du Mardi du parent 1 par exemple, et toutes les affectations de missions du Mercredi, Jeudi et Vendredi du parent 2.
- ***Days NX*** : de la même manière que le *Day 1X*, cet opérateur procède à un échange de journées entre deux chromosomes, mais au lieu de scinder le chromosomem en deux, on le scinde en autant de parties qu'il y a de jours de travail. Ainsi, un enfant peu recevoir toutes les affectations de mission du Lundi du parent 1, toutes les affectations de missions du Mardi du parent 2, toutes les affectations des missions du Mercredi du parent 1, etc ...

Nous avons garder uniquement ces deux opérateurs de croisement, car ceux-ci nous assuraient d'obtenir une solution valide après chaque croisement. Le problème à résoudre étant assez complexe et les contraintes assez dures à respecter totalement, il n'y avait pas beaucoup d'opérateurs de croisement permettant d'obtenir des solutions valides à chaque fois. Voici notre l'implémentation de l'opérateur de croisement ***Day 1X*** :

```
void crossover_1X(Chromosome *parent1, Chromosome *parent2, Chromosome *child1,
    Chromosome *child2)
{
    for (int i = 0; i < n_employee; ++i)
    {
        child1->employee_timetables[i * N_WEEK_DAY + MONDAY] = parent1->
            employee_timetables[i * N_WEEK_DAY + MONDAY];
        child1->employee_timetables[i * N_WEEK_DAY + TUESDAY] = parent1->
            employee_timetables[i * N_WEEK_DAY + TUESDAY];
        child1->employee_timetables[i * N_WEEK_DAY + WEDNESDAY] = parent1->
            employee_timetables[i * N_WEEK_DAY + WEDNESDAY];
        child1->employee_timetables[i * N_WEEK_DAY + THURSDAY] = parent2->
            employee_timetables[i * N_WEEK_DAY + THURSDAY];
        child1->employee_timetables[i * N_WEEK_DAY + FRIDAY] = parent2->
            employee_timetables[i * N_WEEK_DAY + FRIDAY];

        child2->employee_timetables[i * N_WEEK_DAY + MONDAY] = parent2->
            employee_timetables[i * N_WEEK_DAY + MONDAY];
        child2->employee_timetables[i * N_WEEK_DAY + TUESDAY] = parent2->
            employee_timetables[i * N_WEEK_DAY + TUESDAY];
        child2->employee_timetables[i * N_WEEK_DAY + WEDNESDAY] = parent2->
            employee_timetables[i * N_WEEK_DAY + WEDNESDAY];
        child2->employee_timetables[i * N_WEEK_DAY + THURSDAY] = parent1->
            employee_timetables[i * N_WEEK_DAY + THURSDAY];
        child2->employee_timetables[i * N_WEEK_DAY + FRIDAY] = parent1->
            employee_timetables[i * N_WEEK_DAY + FRIDAY];
    }
}
```

3.3.4 Mutation

Pour ce qui est des mutations, elles s'effectuent sur une partie de la population. Une mutation apparaît sur certains individus avec une propabilité définie par le taux de mutation. Pour conserver au maximum la validité de notre population, les opérateurs de mutation permuttent des missions d'employés ayant la même compétence.

- ***Full swap*** : cet opérateur de mutation permutte une journée entière de deux employés.
- ***Rand swap*** : cet opérateur agit aussi sur deux employés et permutte toutes les missions commençant par une heure choisie aléatoirement.

Voici l'implémentation du ***Full swap***:

```
        void mutate_full_swap(Chromosome *chromosome, const Employee employees[], std
            ::default_random_engine &generator)
{
    int day;
    int employee_index2;
    int employee_index1;
    std::uniform_int_distribution<int> uniform_dist_emp(0, n_employee - 1);
    std::uniform_int_distribution<int> uniform_dist_day(0, N_WEEK_DAY - 1);

    day = uniform_dist_day(generator);
    employee_index2 = uniform_dist_emp(generator);
    employee_index1 = uniform_dist_emp(generator);
    while (employees[employee_index1].skill != employees[employee_index2].skill ||
        employee_index1 == employee_index2)
        employee_index2 = uniform_dist_emp(generator);

    Chromosome temp_chromosome = *chromosome;

    temp_chromosome.employee_timetables[employee_index2 * N_WEEK_DAY + day] =
        chromosome->employee_timetables[employee_index1 * N_WEEK_DAY + day];
    temp_chromosome.employee_timetables[employee_index1 * N_WEEK_DAY + day] =
        chromosome->employee_timetables[employee_index2 * N_WEEK_DAY + day];

    if(temp_chromosome.is_valid()){
        chromosome->employee_timetables[employee_index1*N_WEEK_DAY + day] =
            temp_chromosome.employee_timetables[employee_index1 * N_WEEK_DAY + day];
        chromosome->employee_timetables[employee_index2*N_WEEK_DAY + day] =
            temp_chromosome.employee_timetables[employee_index2 * N_WEEK_DAY + day];
    }
}
```

3.4 Optimisations diverses

Par soucis de performances, nous avons effectué un certain nombre d'optimisations dans notre implémentation. En effet, le C++ nous offre une grande marge de manoeuvre quant à la gestion des ressources que nous avons exploité dans les optimisations suivantes :

- $i++$ est un opérateur à proscrire pour simplement incrémenter i car il stocke la valeur de i avant son incrémentation et la retourne. Pour éviter ce gachis de ressource nous utilisons l'opérateur $++i$ pour incrémenter nos variables.
- Nous utilisons au maximum le passage de variables par adresse ou par référence quand celles-ci sont en lecture-seul. Cela évite d'utiliser de la mémoire pour des éléments destinés uniquement à être lus.
- Nous avons utilisé des tableaux à une dimension pour représenter des tableaux à deux dimensions afin d'avoir une mémoire continu, cela rend l'accès aux données plus rapide.
- Afin de ne pas laisser des itérations tourner inutilement, nous utilisons les opérateurs *break*; et *continue*; au maximum pour stoper les boucles lorsque les itérations suivantes ne sont plus nécessaires.
- Nous utilisons peu de variables globales, et nous passons tous nos tableaux en paramètres car l'accès à une variable globale demande plus de ressources que l'accès aux paramètres de la fonction.
- Nous avons utilisé les flags de compilation *-O3* et *-flto* pour que le compilateur réalise des optimisations de notre code au moment de la compilation.

4 Benchmark

Afin de trouver les meilleurs paramètres pour générer la meilleure solution possible nous avons implémenté une fonction *benchmark*. Cette fonction résoud plusieurs fois le problème avec des paramètres différents et stocke le résultat de chaque résolution dans un fichier benchmark.txt. Cela permet de constater l'influence des paramètres sur les résultats. Pour réaliser les benchmarks sur chaque instance nous avons choisis les paramètres suivants :

- itération = 50 000
- population = 50
- taux de mutation = {0.2,0.3,0.4,0.5}
- taux de croisement = {0.2,0.4,0.6,0.8}

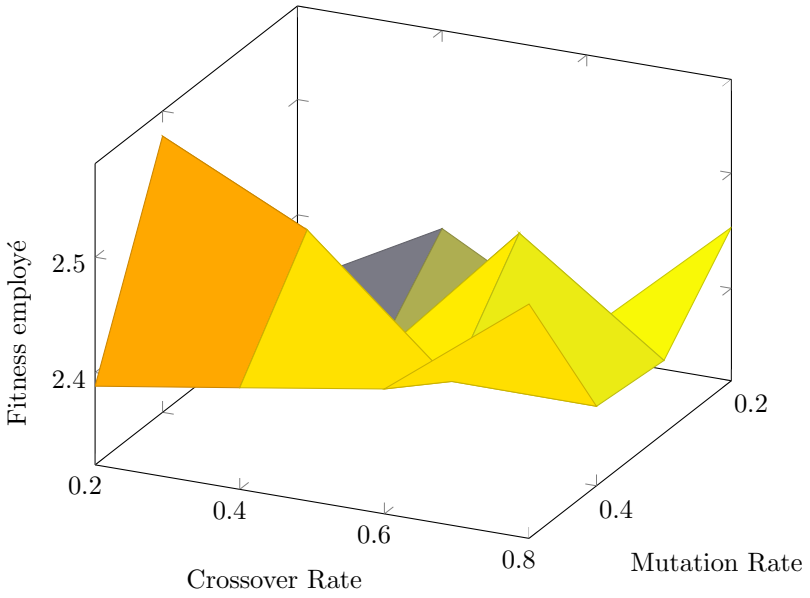

Figure 2: Instance 45

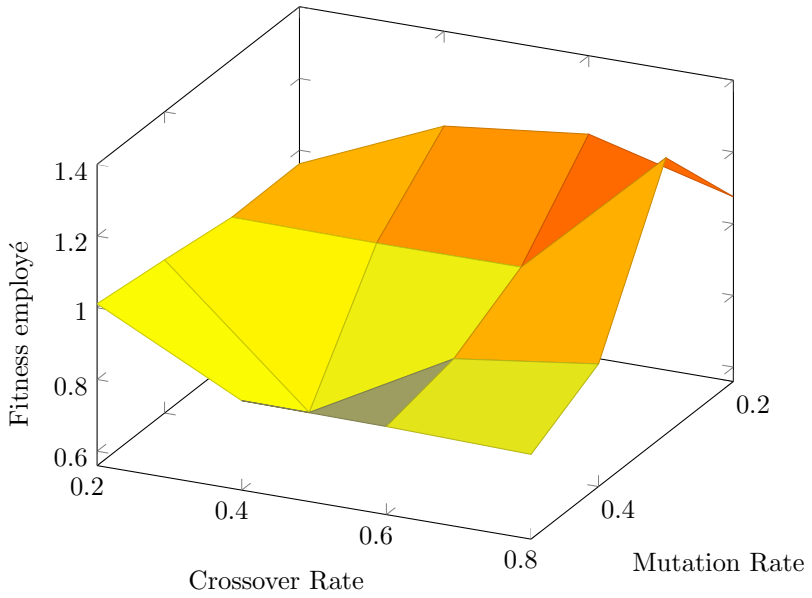

Figure 3: Instance 96

Figure 4: figure caption

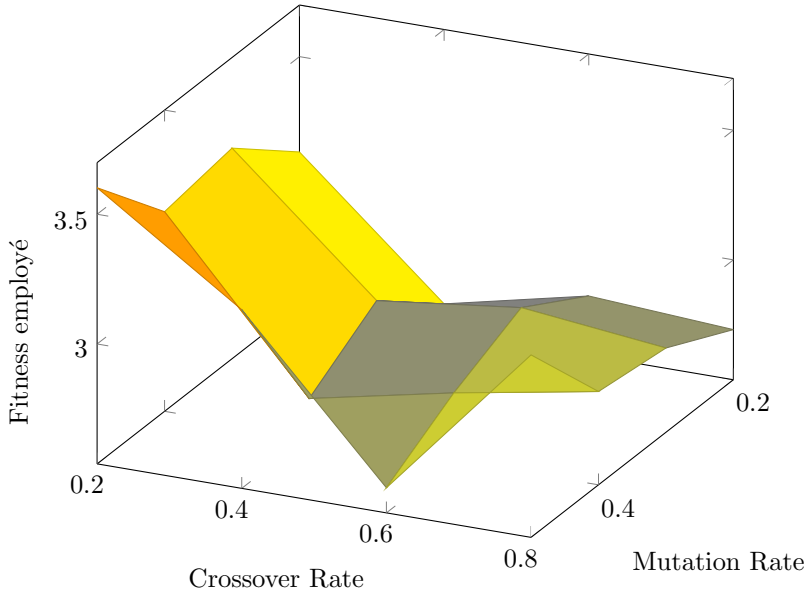

Figure 5: Instance 100

Afin de mieux compredndre ces résultats nous les avons affiché sous forme de graphique (voir figure ci-dessus). En abscisse on retrouve le taux de mutation, en ordonnée, le taux de croisement et en côte la première fitness. Sur la figure 2, on constate l'importance des mutations et des croisements car plus le taux de ces deux paramètres est élevé, plus la fitness est basse. La figure 3 nous confirme l'importance des croisements car plus le taux de croisement est élevé plus la fitness est basse. Cependant ce n'est pas une science exacte car la figure 1 est inexploitable.