Rapport de projet

IA54 – semestre UTBM A2017

Application des optimisations par colonies de fourmis aux problèmes du voyageur de commerce

Étudiants : BOUCHEREAU Thomas – PROST Guillaume

Suiveur de projet : GAUD Nicolas

Sommaire

Table des matières

Index des illustrations

Présentation

Dans ce rapport, nous allons présenter le projet qu'il nous a été donné de faire à l'Université de Technologie de Belfort-Montbéliard, dans l'Unité de Valeur IA54, au cours du semestre Automne 2017.

Description du sujet

Ce projet portait sur la création d'un programme permettant de résoudre des instances du problème du TSP (comprendre *Travelling Salesman Problem*, ou en français *Problème du voyageur de commerce*), qui est un type de problème d'optimisation très répandu, du fait de la simplicité de son énoncé et de l'importance de son optimisation.

Ce programme devait être réalisé en utilisant le langage de programmation orienté agent SARL.

Il devait être doté d'une interface graphique permettant de visualiser à la fois le graph correspondant aux données actuellement en cours d'optimisation, et à la fois l'évolution de la valeur objectif. Elle devait également permettre l'interaction avec la partie optimisation du projet. L'interface pouvait être réalisée avec l'API de notre choix parmi Java FX et Swing (deux API graphiques en Java).

L'utilisation de librairies externes était autorisée et sans limitation.

Pistes d'amélioration

Divers aspects du programme pouvaient être approfondis dans le but de l'améliorer.

Les 3 aspects clé qui ont été retenus sont :

1. Variantes du TSP

   Cet aspect touche à la méthode de résolution du TSP. Une variante du TSP est une modification de sa méthode d'optimisation et/ou de ses données, pour obtenir d'autres résultats ou pour chercher à optimiser d'autres types de problèmes.

   Le programme pourrait potentiellement permettre la résolution de plusieurs variantes différentes, le rendant plus versatile.

2. Dimensions des problèmes

   Cet aspect touche à la taille des données. Les optimisations du TSP sont connues pour rapidement devenir très complexes et laborieuses, aussi il serait intéressant de chercher à rendre le programme performant pour les problèmes ayant beaucoup de données. La limite fixée avec l'enseignant est à 500 nœuds pour être valide, et chercher à aller au-delà est du domaine de l'amélioration du projet.

3. Dynamicité de l'optimisation

   Il serait intéressant de pouvoir modifier les données du TSP sans interrompre l'optimisation, pour pouvoir observer le comportement du solveur quand son environnement est modifié.

Cette dynamicité est donc très intéressante, mais représente un challenge au niveau de l'implémentation.

Réalisation

Principe de fonctionnement

Dans cette partie, nous allons définir les différents acteurs théoriques qui composent la version finale de notre projet, et nous allons présenter leur fonctionnement global, c'est-à-dire ce qu'on peut en attendre et les éventuelles équations qu'ils vont mettre en œuvre.

Description des différents éléments

Ces quatre types d'agents sont les suivants : agent environnement, agent séquenceur, agent afficheur et agent fourmi. Les trois premiers types ne seront instanciés qu'une seule fois, et le dernier type aura un nombre d'agent correspondant variable.

L'agent environnement conserve en interne toutes les données liées aux solutions actuelles, comme le nombre d'itérations, la meilleure solution trouvée, la matrice des distances ou encore le niveau de phéromones.

À chaque itération, l'environnement va collecter les influences des fourmis, c'est-à-dire les chemins empruntés, et va calculer les nouveaux niveaux de phéromones sur les trajets. Ces nouveaux niveaux sont calculés avec l'équation suivante :

$$pheromone_{(ij,n+1)} = Ev_{pheromones} * \left(\left(\frac{F_{distance} * fourmis_{ij}}{distance_{ij}} \right) + pheromone_{(ij,n)} \right)$$

avec :

- pheromone(ij, n) le niveau de phéromones sur le trajet ij à l'itération n
- Ev(pheromones) le taux d'évaporation des phéromones
- F(distance) un facteur d'importance de la distance
- distance(ij) la distance entre les nœuds i et j
- fourmis(ij) le nombre de fourmis ayant pris le trajet ij lors de l'itération

Il va ensuite notifier le changement de données pour que l'affichage soit mis à jour. Il va également fournir les données de l'environnement aux fourmis par l'intermédiaire de l'agent séquenceur, de manière à ce que l'environnement soit complètement observable par les fourmis (distance, phéromones, etc.).

L'agent séquenceur va s'occuper de gérer le cycle des itérations en s'assurant que toutes les fourmis ont terminé leurs calculs de solutions avant de demander à l'agent environnement d'appliquer les influences. Il va ensuite attendre la fin du traitement de ces influences avant de relancer les fourmis sur une nouvelle itération. Cet agent doit également s'occuper de recréer les fourmis lorsque le paramétrage de ces dernières change, ou lorsque les données du problème TSP changent.

L'agent afficheur sert de passerelle entre le code java gérant l'affichage et le reste des agents. Il doit lancer l'interface graphique lorsqu'il s'instancie, il doit recevoir des événements de la part de l'agent environnement pour mettre à jour les données des solutions, et doit émettre des événements

lorsque des contrôles sont activés sur l'interface graphique. Ces événements sont au nombre de trois :

1. Import d'un nouveau fichier contenant un problème de TSP
2. Modification du paramétrage de l'application
3. Fermeture de l'interface par l'utilisateur

Ces événements sont émis à tous les agents, et les agents peuvent ou non les écouter, et le cas échéant les prendre en compte. Ce choix a été fait car cela simplifiait grandement les procédures de *handshake* à l'initialisation de l'agent affichage et car l'agent affichage n'a pas à savoir à qui sont destiné les informations de certains des événements qu'il doit émettre : en effet, dans le cas des événements liés aux changements de paramètres, seuls les destinataires peuvent déterminer si le paramètre modifié les concerne ou non.

Enfin, les agents fourmis sont chargés, à chaque itération, de calculer chacune un chemin en prenant en compte les données de l'environnement pour l'itération en cours, et de transmettre ce chemin à l'agent environnement. Elles calculent ces chemins en choisissant nœud par nœud le trajet le composant, ce choix étant fait au hasard, et la probabilité d'élection de chaque choix dépend de la distance du trajet correspondant, ainsi que du niveau de phéromones sur ce trajet, en suivant l'équation suivante :

$$probabilité_{ij} = \frac{\left(\frac{1}{distance_{ij}}\right)^{\alpha} + \left(pheromone_{ij}\right)^{\gamma}}{\sum_{k=1}^{n}\left(\frac{1}{distance_{ik}}\right)^{\alpha} + \left(pheromone_{ik}\right)^{\gamma}}$$

avec :

- α le facteur d'importance de la distance
- γ le facteur d'importance du niveau de phéromones
- n le nombre de nœuds
- distance(ij) la distance entre les nœuds i et j
- pheromone(ij) le niveau de phéromone sur le trajet entre les nœuds i et j
- probabilité(ij) la probabilité d'élire j comme destination si l'on se place sur le nœud i

Les fourmis doivent calculer des chemins complets, et doivent recommencer sans notifier l'agent environnement si le trajet ne visite pas tous les nœuds. Les fourmis ne doivent pas changer d'algorithme lorsque la variante du TSP est changée, puisqu'elles seront changées lorsque cela arrivera. Elles doivent cependant être paramétrables à l'initialisation pour s'adapter à l'état actuel du programme.

Cycle de résolution

Les composants ci-dessus ont été définis avec un objectif précis en tête : un cycle de résolution. Ce cycle est un enchaînement d'événements qui constitue une itération de l'optimisation du problème du TSP. Il se déroule comme il suit :

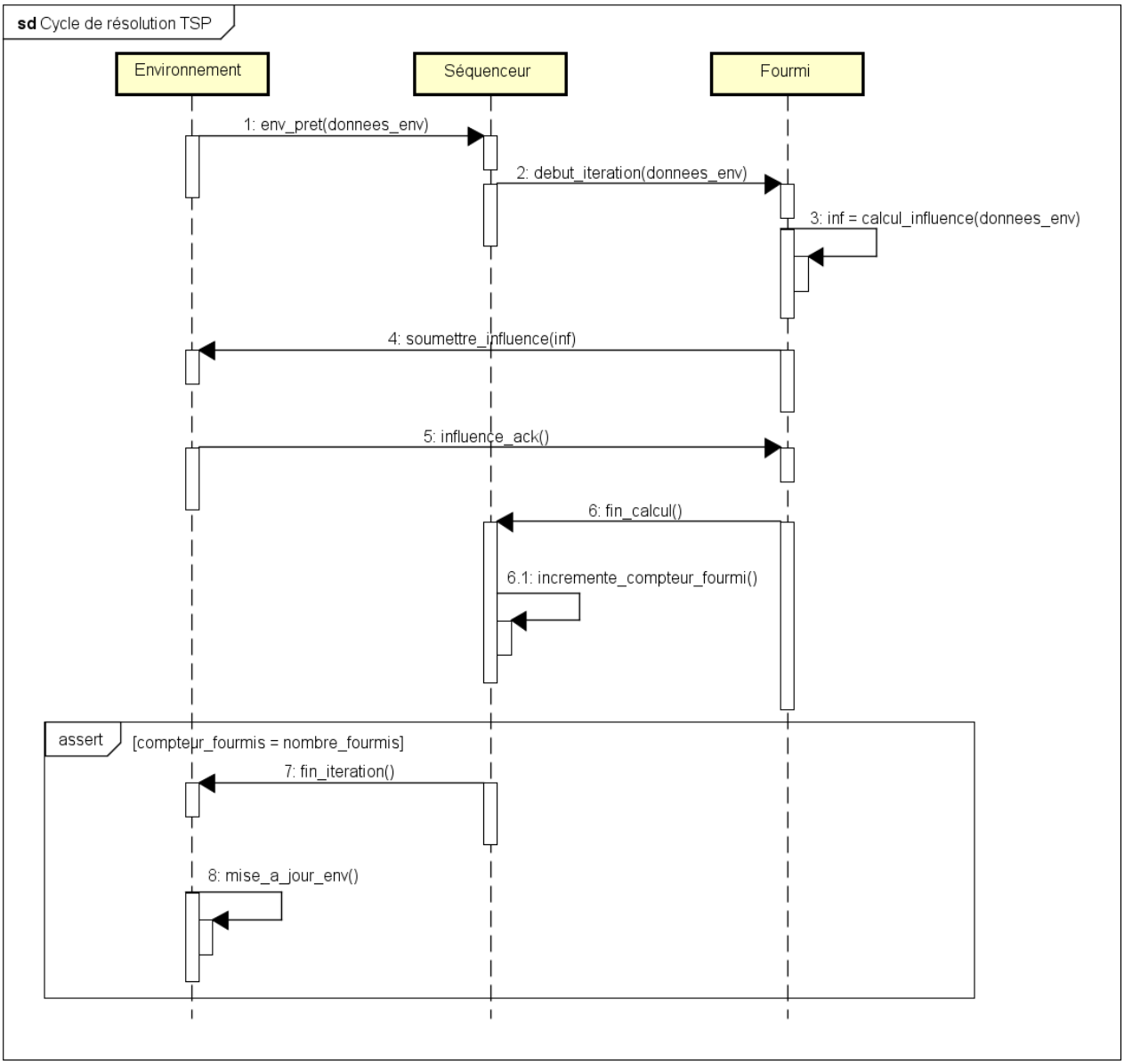

Illustration 1: Diagramme de séquence décrivant le cycle de résolution du programme

Étant donné que plusieurs fourmis calculent et transmettent leurs influences en parallèle, le séquenceur doit comptabiliser le nombre de fourmis ayant terminé leurs calculs, et ne déclenche la suite du cycle que quand toutes les fourmis ont terminé.

Il est à noter que la mise à jour de l'affichage ne fait pas partie de ce cycle, car ce dernier ne représente que la séquence d'étapes que le code métier va suivre pour calculer ce que nous voulons. Dans la pratique, cette mise à jour se situera à la fin de l'étape '*request update*', une fois que l'environnement à appliqué les influences.

Variantes envisagées

Nous pensons pouvoir appliquer ce cycle à plusieurs variantes de TSP. Initialement, des valeurs de TSP classique seront utilisées pour prouver le fonctionnement du cycle et de l'interface. Nous considérons que le TSP 'classique' possède les caractéristiques suivantes :

- le graph définissant les nœuds est complet, bidirectionnel
- la distance du trajet entre deux nœuds n1 et n2 est la même peu importe le sens (n1 → n2 ou n2 → n1)
- une solution est définie par une séquence de nœuds où chaque nœud n'est présent qu'une seule fois. Cette séquence ne contient donc aucun doublon, et ne revient pas au point de départ une fois le dernier nœud visité.
- l'objectif est de minimiser la distance totale parcourue

Par la suite, nous avons décidé d'implémenter, si le temps nous le permet, deux autres variantes de TSP : les TSP '*bottleneck*' et les TSP avec plages horaires

Les TSP dit *'bottleneck'* sont très similaires aux TSP classiques, puisque la seule chose qui diffère étant l'objectif : en effet, cette variante cherche à minimiser la taille maximale des trajets, au lieu de la distance totale. En d'autres termes, on cherche ici à minimiser la taille du plus long trajet entre deux nœuds, et non pas la somme des longueurs des trajets.

L'implémentation de cette variante demandera peu de modifications au niveau de l'algorithme, et est intéressante pour nous car elle nous permet de mettre en place rapidement un second comportement pour les fourmis, ce qui nous permettra de travailler sur cet aspect.

Enfin, les TSP avec plages horaires sont un peu plus complexes. On conserve les caractéristiques de la variante *'classique'*, mais on ajoute quelques éléments :

- l'environnement contient désormais une liste de plages horaires, qui associent désormais un nœud à une collection d'intervalles temporels, durant lesquels le nœud est accessible depuis n'importe quel autre nœud.
- l'environnement contient également l'heure de départ, sur laquelle les fourmis vont se baser pour commencer leurs optimisations.
- les fourmis sont définies comme allant à une vitesse de 1, ce qui nous permet de considérer que notre matrice de distance de trajet est *de facto* une matrice de temps de trajet. Cela nous permet de grandement simplifier la structure interne des fourmis, et nous permet d'établir une égalité entre le temps de trajet et la distance parcourue. Cette égalité est importante car elle nous permet de conserver le même objectif, qui est la minimisation de la distance totale parcourue, sans autres modifications.

Ce sont les trois variantes que nous avons envisagé de mettre en place. À côté de cela, nous avons décidé d'implémenter trois éléments dans l'interface graphique .

Le premier de ces éléments est la visualisation des données de l'optimisation en cours sur deux écrans différents, l'un affichant le graph avec ses nœuds et les solutions (meilleure solution de l'itération et meilleure solution parmi toutes les itérations), l'autre affichant une courbe représentant l'évolution de la valeur objectif, celle que nous cherchons à minimiser, en fonction des itérations.

Le deuxième de ces éléments est une barre d'outil permettant de changer divers paramètres lié à l'optimisation des solutions :

Nom du paramètre	Description
Nombre de fourmis	Entier, nombre de fourmis prenant part à l'optimisation. (vaut 5 par défaut)
Facteur d'évaporation des phéromones	Décimal, influe proportionnellement sur l'évaporation des phéromones entre les itérations. (vaut 0,5 par défaut)
Facteur de distance	Décimal, influe proportionnellement sur la prise en compte de la distance lors du dépôt des phéromones sur un trajet. (vaut 1 par défaut)
Facteur d'importance des phéromones	Décimal, influe exponentiellement sur l'importance du niveau de phéromone lors du calcul des trajets par les fourmis. (vaut 1 par défaut)
Facteur d'importance de la distance	Décimal, influe exponentiellement sur l'importance de la distance des trajets lors du calcul des trajets par les fourmis. (vaut 1 par défaut)
Numéro du premier noeud	Entier, permet d'indiquer un numéro de nœud, qui sera alors pris comme point de départ. La valeur -1 signifie que les fourmis prendront leurs points de départ au hasard. (vaut -1 par défaut)
Numéro du dernier noeud	Entier, permet d'indiquer un numéro de nœud, qui sera automatiquement le dernier nœud visité par les solutions. Une valeur de -1 désactive cette fonctionnalité. (vaut -1 par défaut)

Conception et implémentation des différents éléments

Passons maintenant à la description de la conception et de l'implémentation des éléments décrit dans la partie précédente. Nous allons voir ici comment ceux-ci ont été codés et comment le programme a été organisé d'un point de vue pratique.

Structure globale

Le programme est composé de 3 projets séparés. Deux d'entre eux sont des projets Java classiques, compilé en archive JAR, et ajouté en dépendance du 3ème projet, ce dernier étant un projet SARL. Les deux projets Java sont des projets annexes et seront décrits plus loin dans cette partie.

Le projet SARL, quant à lui, est le projet principal de notre programme. Il contient l'intégralité des agents, des événements, des behaviors et des skills que nous avons défini, et il utilise les deux autres projets annexe dans son code. Nous allons maintenant détailler les 4 parties de ce projet SARL, qui sont l'environnement, le séquenceur, les fourmis et la gestion de l'interface.

Note : dans les sections suivantes, les paramètres des fonctions ont volontairement été retirés des diagrammes de classes, pour des raisons de lisibilité.

Environnement

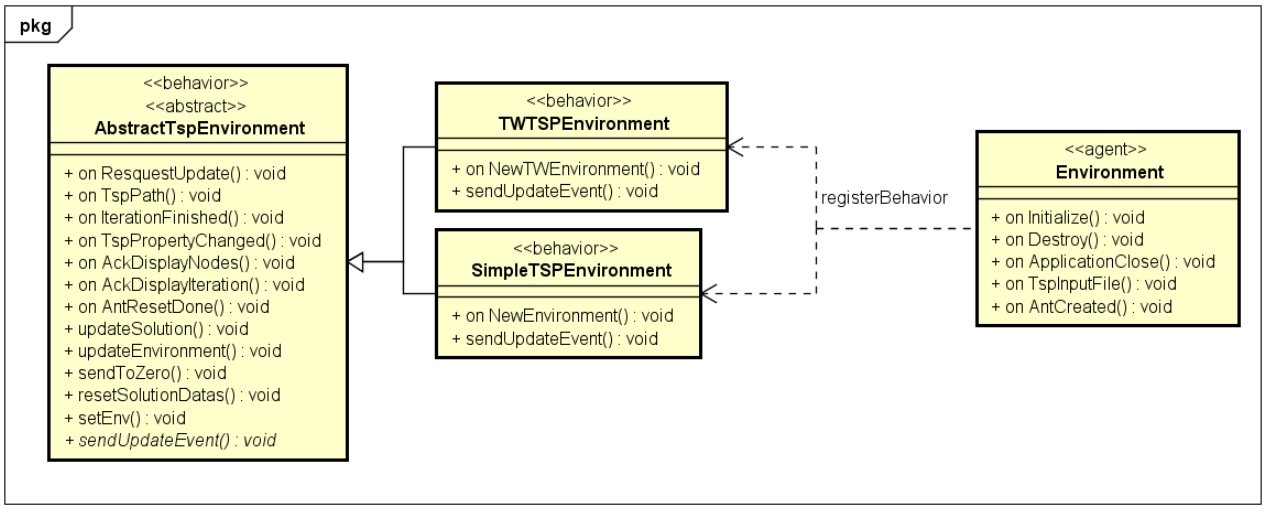

Illustration 2: Diagramme de classe représentant l'implémentation de la notion d'environnement

L'environnement a été implémenté comme un agent dans notre programme. Cependant, comme cet agent ne doit pas gérer les mêmes données suivant les variantes du TSP, nous avons créé des behaviors pour contenir le code métier, behavior que l'agent va pouvoir utiliser ou non, et sera en mesure d'en changer dynamiquement. Les variantes *classique* et *bottleneck* utilisent le même behavior, puisque les données utilisées sont identiques.

Comme une grande majorité du code est commun à toutes les variantes du TSP, nous avons créé un super-behavior pour centraliser tout le code métier commun. Ce behavior est défini comme abstrait et possède une méthode abstraite pure, *sendUpdateEvent()*, qui permet d'envoyer au séquenceur la notification de fin de mise à jour, et qui transmet également les nouvelles données de l'environnement. Le code centralisé concerne la mise à jour de l'environnement entre les itérations, la réception des influences des fourmis, le handshake avec ces dernières et le traitement des changements de paramètres qui sont émis lorsque l'utilisateur modifie l'un d'eux grâce à l'interface.

La seule chose qui n'est pas commune entre les variantes, c'est l'initialisation de l'environnement, qui se fait dans notre programme par des événements séparés suivant le behavior (via *NewTWEnvironment* ou *NewEnvironment*).

L'agent *Environment* ne contient lui-même que très peu de chose. Il s'occupe uniquement de réceptionner les événements *TspInputFile*, qui sont levés lorsque l'utilisateur décide d'importer un nouveau fichier de TSP. L'agent va alors parser le fichier, et suivant les données lues, va changer de behavior (ou non) et va émettre un événement *NewEnvironment* ou *NewTWEnvironment* (suivant son behavior actif) pour réinitialiser les données. Dans ce cas de figure, la première notification émise au séquenceur pour fournir les données d'environnement va lui signifier qu'il doit réinitialiser le groupe de fourmis.

Il est à noter que l'agent *Environment* reçoit également un autre événement : *ApplicationClose*. Ce dernier est levé lorsque l'interface graphique a été fermée, et signifie à tous les agents que le programme doit s'arrêter.

Les données de l'environnement ont été conservées sous al forme d'une matrice de décimaux à 3 dimensions. Les deux premières dimensions ont une taille égale au nombre de nœuds, et la 3e dimension à une taille de 2. Cela nous permet de stocker à la fois la distance et le niveau de phéromones.

Ex : env[i][j][0] contient la distance entre les nœuds i et j, et env[i][j][1] contient le niveau de phéromone sur le trajet $i \rightarrow j$.

Une solution, ou influence, est stockée sous la forme d'un tableau d'entiers, contenant les identifiants des nœuds visités dans l'ordre de visite. Une solution est composée également d'une valeur décimale, qui correspond à la valeur objectif.

Séquenceur

Le séquenceur a été implémenté comme un unique agent, sans *behavior* ni *capacity* spécifiques. En effet, la simplicité des interactions qui le concerne et la simplicité de son code interne limite fortement l'utilité de l'abstraction ou de la centralisation de code.

Cependant, il doit quand même être capable de gérer toutes les variantes de TSP implémentées. Il a donc directement des entrées pour les événements de chaque variante (ici, les événements *Updated* et *TWUpdated*).

Le reste des événements concerne la gestion du cycle des fourmis, et cette gestion ne varie pas suivant les variantes.

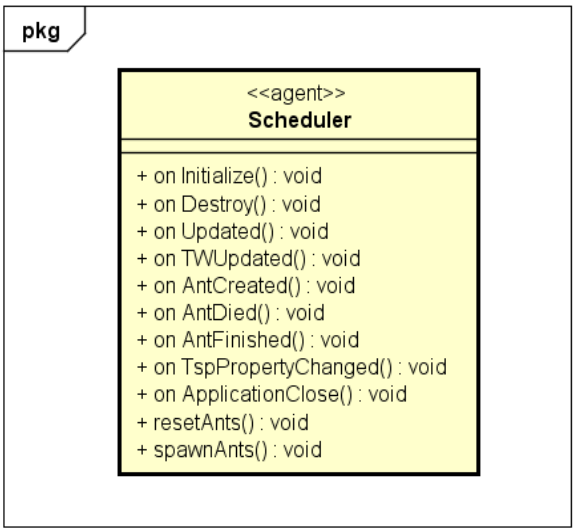

Illustration 3: Diagramme de classe représentant l'implémentation du séquenceur

À son initialisation, l'agent *Scheduler* lance l'initialisation des fourmis, et attend la fin de leurs initialisations, signifiée par un événement *AntCreated*. Lorsque le bon nombre de fourmis est atteint, l'agent demande le lancement de l'itération à l'environnement. Il transmet ensuite les données de l'environnement aux fourmis via des événements *SimpleIterationStart* (ou *TWIterationStart*, mais nous le verrons dans la section suivante). Les fourmis vont alors lancer leurs calculs, vont notifier l'environnement, et lorsqu'elles reçoivent la réponse de l'agent *Environment*, vont émettre un événement *AntFinished* à destination du séquenceur. Ici aussi, le séquenceur va attendre que toutes les fourmis aient envoyé l'événement avant de lancer la suite du cycle, en notifiant l'environnement. Lorsque l'environnement répond, un nouveau cycle commence.

L'agent *Scheduler* reçoit également des événements de la part de l'interface : *TspPropertyChanged* et *ApplicationClose*. Ils ont presque la même signification pour cet agent que pour l'agent *Environment* décrit précédemment. La seule différence avec ce dernier est que les paramètres que notre agent séquenceur gère, et qui sont potentiellement modifiés avec *TspPropertyChanged*, sont différents de ceux de l'environnement.

Notre agent séquenceur gère également les paramètres relatifs aux fourmis, et lorsqu'un de ces paramètres est modifié, il recréé toutes les fourmis avec les nouvelles valeurs, en abandonnant les

précédentes. Ce renouvellement des fourmis ne se fait normalement qu'entre les itérations, pour des raisons de logique. En effet, c'est également à ce moment-là que le séquenceur reçoit la notification de l'environnement qui contient potentiellement des données d'un nouveau *benchmark*, et où il doit également renouveler le groupe de fourmis. Cela permet d'effectuer un unique test pour toutes les conditions, et de renouveler (ou non) une unique fois le groupe de fourmis.

Le renouvellement se termine par un événement destiné à l'agent *Environment*, qui lui demande de ré-envoyer les données de l'itération.

Fourmis

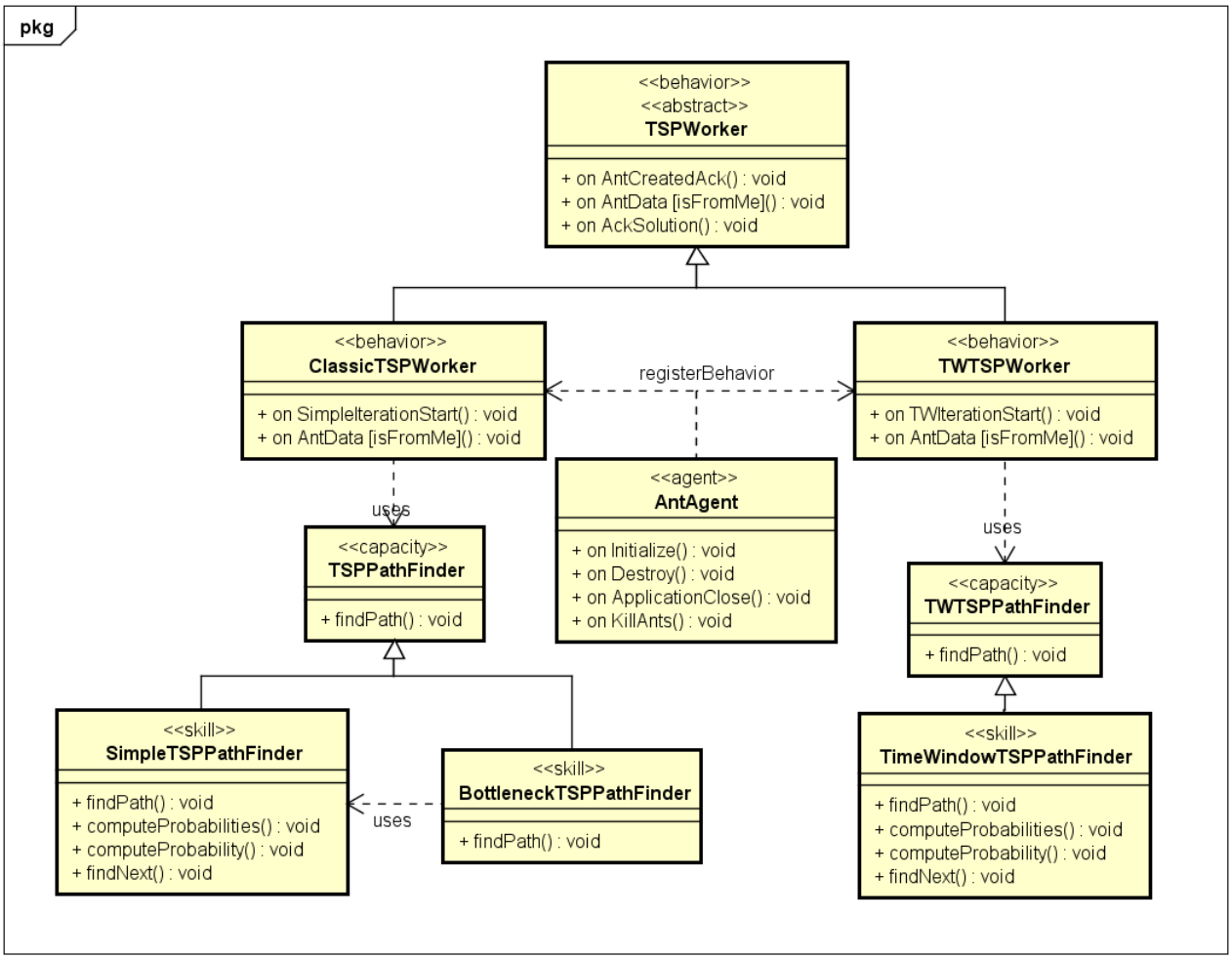

Illustration 4: Diagramme de classe représentant l'implémentation des fourmis et des algorithmes de résolution

L'implémentation des fourmis est l'implémentation la plus complexe du programme. Comme pour l'agent *Environment*, les fourmis utilisent des behaviors différents suivant la variante du TSP actuellement en cours d'optimisation. Puisque le séquenceur renouvelle les fourmis à chaque changement de variante, ces dernières n'ont pas à changer dynamiquement de behavior, et la variante peut être définie à l'initialisation. Ici aussi le code commun aux deux behaviors a été centralisé dans un super-behavior abstrait.

Celui-ci contient le code correspondant aux événements de confirmation (*AntCreatedAck* et *AckSolution*), et centralise également divers attributs, comme l'UUID de l'agent environnement, ou

encore les paramètres d'importance de la distance et des phéromones, dont nous avons parlé dans la section sur les fonctionnalités.

Les behaviors 'fils' n'implémentent que les réceptions d'événements liés aux lancements d'itérations, chaque événement contenant toutes les données nécessaires à la génération d'une influence (donc d'un chemin pour le TSP) pour la variante en cours. Ici aussi, les variantes *classique* et *bottleneck* utilisent le même behavior, puisque l'environnement est composé des mêmes données.

Concernant les algorithmes de résolutions, ils sont rassemblés dans des *capacity* et des *skill*. On peut voir qu'il existe deux capacity, chacune étant destinée à un behavior 'fils' en particulier, puisque la méthode *findPath* prend en paramètre les données de l'environnement.

On peut également voir que ces *skill* sont l'unique endroit du programme où les variantes *classique* et *bottleneck* ne sont pas rassemblées. En effet, c'est ici que la différence entre les variantes utilisant les mêmes données se fait. Dans le cas de ces deux variantes, la grande majorité de l'algorithme est commun, il a donc été écrit pour un seul *skill*, et l'autre utilise le premier. Le code pour la variante *avec plages horaires* a été réécrit complètement, en raison des différences de données avec les autres variantes, mais il reste commun dans la structure et dans l'ordre des étapes. Un *design pattern* ***méthode*** aurait pu ici être utilisé si le programme avait été destiné à être utilisé et amélioré par la suite.

```
Fonction trouverSolution(nb_noeuds, env, id_noeud_depart, id_dernier_noeud){
    Paramètres
        * nb_noeuds : entier
        * env :  matrice 3D de décimaux de taille (nb_noeuds * nb_noeuds * 2)
        * id_noeud_depart : entier
        * id_dernier_noeud : entier
        * facteur_importance_distance : décimal
        * facteur_importance_pheromone : décimal

    variables
        * noeuds_visités : tableau de binaires de taille (nb_noeuds)
        * noeuds_solution : tableau d'entiers de taille (nb_noeuds])
        * proba_election_destination : tableau de décimaux de taille (nb_noeuds)
        * numero_etape : entier
        * distance_Atale : décimal
        * id_prochain_noeud : entier

    begin
        noeuds_visités <- {false, false, ..., false}
        noeuds_solution <- {-1, -1, ..., -1}
        noeuds_solution[0] <- id_noeud_depart
        numero_etape <- 0
        distance_Atale <- 0

        POUR numero_etape de 1 à nb_noeuds-1 FAIRE{
            id_prochain_noeud <- -1

            FAIRE{
                SI(numero_etape = nb_noeuds-1 ET id_dernier_noeud <> -1) FAIRE{
                    id_prochain_noeud <- id_dernier_noeud
                }SINON{
                    proba_election_destination <- calculerProbabilitéDestinations(...)

                    id_prochain_noeud <- trouverProchainNoeud(nb_noeuds, proba_election_destination)
                }
            }TANTQUE(id_prochain_noeud = -1)

            distance_Atale <- distance_Atale + env[noeuds_solution[numero_etape-1]][id_prochain_noeud][0]
            noeuds_solution[numero_etape] <- id_prochain_noeud
            noeuds_visités[id_prochain_noeud] <- true

        }

        Renvoyer Résultat(noeuds_solution, distance_Atale)
    Fin
}
```

Illustration 5: Algorithme de la fonction findPath(...)

L'algorithme de calcul de chemin des fourmis est décliné en 3 fonctions distinctes. Nous allons maintenant présenter l'algorithme de la variante *classique*.

La première, que l'on peut voir sur la figure 5, correspond à la méthode *findPath(…)* présente sur le diagramme de classe. C'est donc le point d'entrée principal de l'algorithme. Le fonctionnement est relativement simple : un premier nœud est déterminé comme point de départ en amont de l'appel de la fonction. Ensuite, pour chaque étape du trajet global de la fourmi, l'algorithme va calculer les probabilités pour chaque destination d'être élue grâce à *calculerProbabilitéDestinations(…)*, comme on peut le voir à la ligne 32 de l'algorithme. Elle va ensuite choisir l'un de ces nœuds grâce à *trouverProchainNoeud(…)*. Cette étape est répétée tant qu'aucun nœud n'a été choisi.

Note : contrairement à l'algorithme ci-contre, l'implémentation finale de l'algorithme intègre un nombre maximum de boucles, qui force la fourmi à recommencer du début la génération du chemin lorsqu'il est atteint. Cela permet d'éviter les points de blocage.

Une fois un nœud choisi, on met à jour les données de l'algorithme et on recommence jusqu'à former un chemin passant par tous les nœuds. À la fin, on transmet comme résultat une structure de donnée contenant à la fois le chemin et la valeur objectif.

```
1  Fonction calculerProba{
2      Paramètres
3          * nb_noeuds : entier
4          * distance_destination : matrice 2D de décimaux de taille (nb_noeuds * 2)
5          * noeuds_visités : tableau de binaires de taille (nb_noeuds)
6          * id_dernier_noeud : entier
7          * facteur_importance_distance : décimal
8          * facteur_importance_pheromone : décimal
9
10     Variables
11         * nbNoeudsEligibles : entier
12         * denominateur : décimal
13         * iter : entier
14         * proba_election_destination : tableau de décimaux de taille (nb_nodes)
15
16     Début
17         nbNoeudsEligibles <- 0
18         AtalPheromone <- 0
19
20         POUR iter DE 0 A nb_noeuds-1 FAIRE{
21             SI(noeuds_visités[iter] = false ET iter <> id_dernier_noeud) FAIRE{
22                 nbNoeudsEligibles = nbNoeudsEligibles + 1
23                 denominateur = denominateur + F(...)
24             }
25         }
26
27         POUR iter DE 0 A nb_noeuds-1 FAIRE{
28             SI(noeuds_visités[iter] = false ET iter <> id_dernier_noeud ET iter = 0) FAIRE{
29                 proba_election_destination[iter] <- F(...)
30             }SINON SI(noeuds_visités[iter] = false ET iter <> id_dernier_noeud) FAIRE{
31                 proba_election_destination[iter] <- proba_election_destination[iter-1] + F(...)
32             }SINON SI(iter = 0) FAIRE{
33                 proba_election_destination[iter] <- 0
34             }SINON{
35                 proba_election_destination[iter] <- proba_election_destination[iter-1]
36             }
37         }
38
39         Renvoyer proba_election_destination
40     Fin
41 }
```

Illustration 6: Algorithme de la fonction computeProbabilities(...)

Ensuite, la deuxième fonction de l'algorithme est la fonction *calculerProbabilitéDestinations(…)*, qui correspond à la méthode *computeProbabilities(...)* sur le diagramme de classe. Cette fonction

prend en paramètre la matrice environnement correspondant aux destinations depuis le nœud actuel, un tableau de binaires répertoriant pour chaque nœud s'il a été déjà visité ou non, l'identifiant du dernier nœud à visiter et les facteurs d'importance, et renvoie un tableau de probabilité.

Ces probabilités sont cumulatives, c'est-à-dire qu'une case i du tableau contient la somme des probabilités pour les nœuds de 0 à i, et que proba[i]-proba[i-1] donne la probabilité d'élection du nœud i en tant que destination. Ce tableau a été construit de cette manière pour facilité l'élection du nouveau nœud, qui se fait dans la 3e fonction. En pratique, les lignes 27 à 37 sont celle qui inscrivent les probabilités dans le tableau. Si un nœud est éligible, sa probabilité est calculée normalement, et sa case est remplie avec la somme entre celle-ci et la case précédente du tableau. Si le nœud n'est pas éligible, comme c'est par exemple le cas avec les nœuds déjà visité, alors la case est remplie avec la valeur de la case précédente. Avec ce système, la dernière case contient une valeur égale à 1 (ou proche).

La fonction de calcul des probabilités, représenté dans l'algorithme ci-contre par la fonction *F(…)*, est l'implémentation exacte de l'équation décrite dans la partie *Principe de fonctionnement* de ce rapport.

```
1  Fonction trouverProchainNoeud(nb_noeuds, proba_election_destination){
2      Paramètres
3          * nb_noeuds : entier
4          * proba_election_destination : tableau de décimaux de taille (nb_noeuds)
5
6      Variables
7          * rand : décimal
8          * temp : entier
9          * id_noeud_elu : entier
10
11     Début
12         rand <- nbAléaAire(0,1)
13         id_noeud_elu <- -1
14         temp <- 0
15
16         TANTQUE (rand >= proba_election_destination[temp] ET temp<nb_noeuds) FAIRE{
17             temp = temp + 1
18         }
19
20         SI(rand >= proba_election_destination[temp]) FAIRE{
21             id_noeud_elu <- -1
22         }SINON{
23             id_noeud_elu <- temp
24         }
25
26         Renvoyer id_noeud_elu
27     Fin
28 }
```

Illustration 7: Algorithme de la fonction findNext(...)

La troisième fonction de l'algorithme est la fonction *trouverProchainNoeud(…)*, appelée *findNext(…)* sur le diagramme de classe. Elle prend en paramètre le nombre de nœuds et le tableau de probabilités cumulatives généré par la fonction précédente, et renvoie un entier correspondant au numéro du nœud qui a été élu comme destination.

La détermination du nœud se fait via la génération d'un unique nombre décimal aléatoire entre 0 et 1. Il est ensuite comparé aux probabilités du tableau, en commençant par l'indice 0. Tant que la valeur du tableau est inférieure à la valeur aléatoire, le parcours se poursuit. Une fois qu'on atteint une valeur dans le tableau qui est supérieure à la valeur aléatoire, on récupère l'indice de la case, et cet indice devient le numéro de la prochaine destination. Il est renvoyé comme résultat de la fonction.

Il est parfois possible que ce nombre ne soit pas déterminé, auquel cas la fonction renvoie -1.

Dans ce cas de figure, la détermination du prochain nœud recommence (Voir fonction *findPath(...)*).

Pour la variante *bottleneck* du problème du voyageur de commerce, une infime portion de l'algorithme a été modifié. En effet, seule la valeur objectif est traitée différemment : au lieu de cumuler des distances de trajets, l'algorithme va uniquement garder en mémoire la distance la plus grande.

Le cas de la variante *avec plages horaires* est différent : il a fallu ajouter une portion de code à la fonction calculant les probabilités d'élection de chaque destination. Cette portion de code a été ajoutée aux endroits où on détermine si un nœud est éligible, et permet de prendre en compte les plages horaires : si l'heure actuelle est comprise entre les valeurs de début et de fin d'au moins une plage horaire associée à un nœud, il est considéré comme accessible. Si un nœud n'a pas de plages horaires, alors il est d'office considéré comme accessible. Un nœud non-accessible à cause de ses plages horaires est donc associé à une probabilité de 0.

Cette méthode d'intégration est donc une méthode « idiote » (comprendre de faible niveau de cognition) et a le défaut d'augmenter les chances d'obtenir un chemin incomplet. Cependant, nous voulions garder nos fourmis suffisamment simples et augmenter leur niveau de cognition allait à l'encontre de notre vision de la chose.

Note importante

Au final, en raisons de problèmes de compilations incompréhensibles et à cause d'erreurs levées par l'IDE Eclipse pour SARL, la variante du TSP *avec plages horaires* n'est pas fonctionnelle. L'intégralité du code associé est présent dans le projet, mais le programme est incapable de récupérer le résultat du parsing des plages horaires, pour une raison incompréhensible.

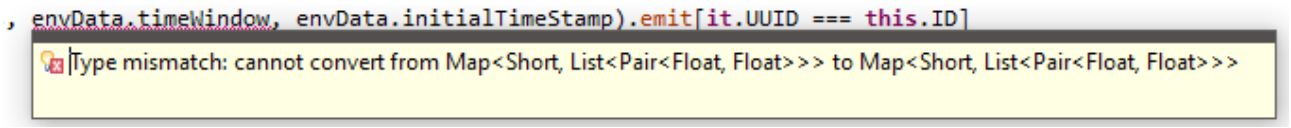

Illustration 8: Erreur de compilation empêchant l'implémentation de la variante du TSP avec plages horaires

La figure ci-contre est une capture de l'erreur de compilation qui est générée lorsqu'on essaye d'utiliser le résultat du parsing d'un fichier contenant des plages horaires. La ligne ici représentée est présente dans le code de l'agent *Environment* (Fichier *Environment.sarl*), dans le récepteur de l'événement *TspInputFile*.

Interface utilisateur

Comme décrit précédemment, l'interface utilisateur est composé de deux écrans différents, l'un présentant un graph avec les nœuds, l'autre présentant une courbe d'évolution des valeurs objectifs en fonction des itérations. La navigation entre ces deux écrans se fait via un menu *Ecrans* en haut de la fenêtre.

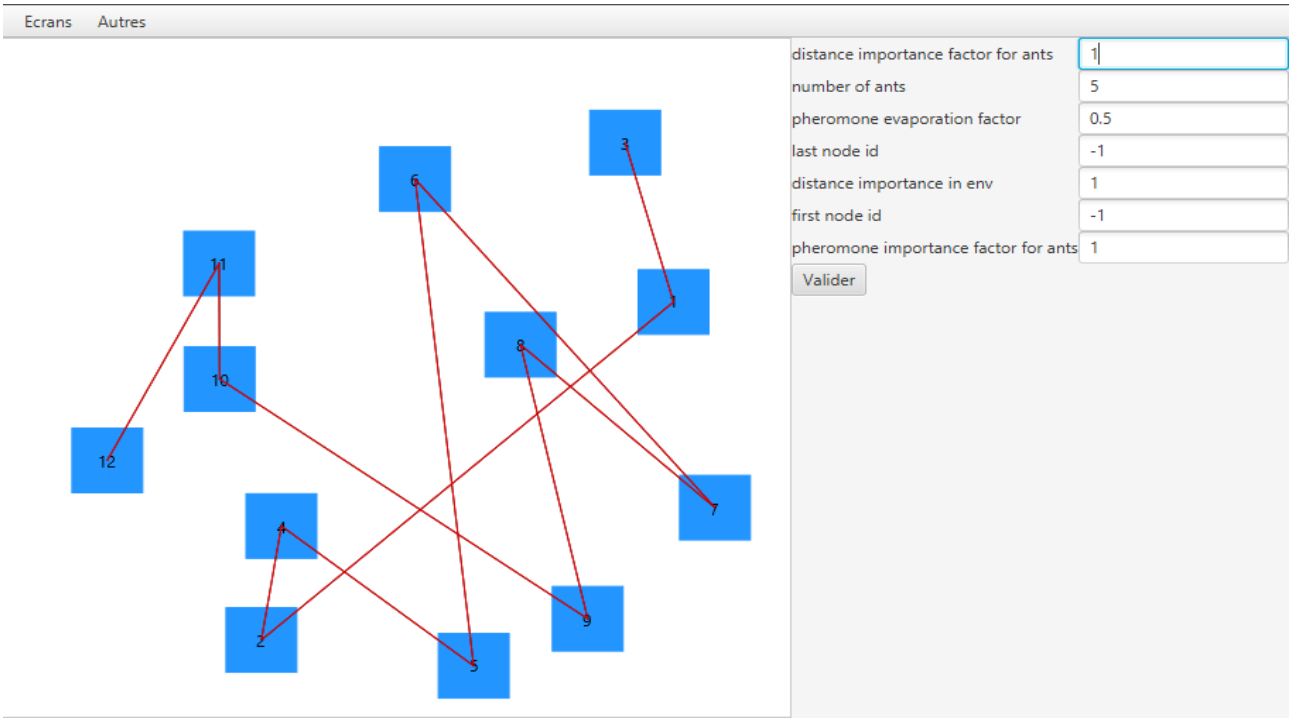

Illustration 9: Ecran de l'interface montrant les noeuds de la solution

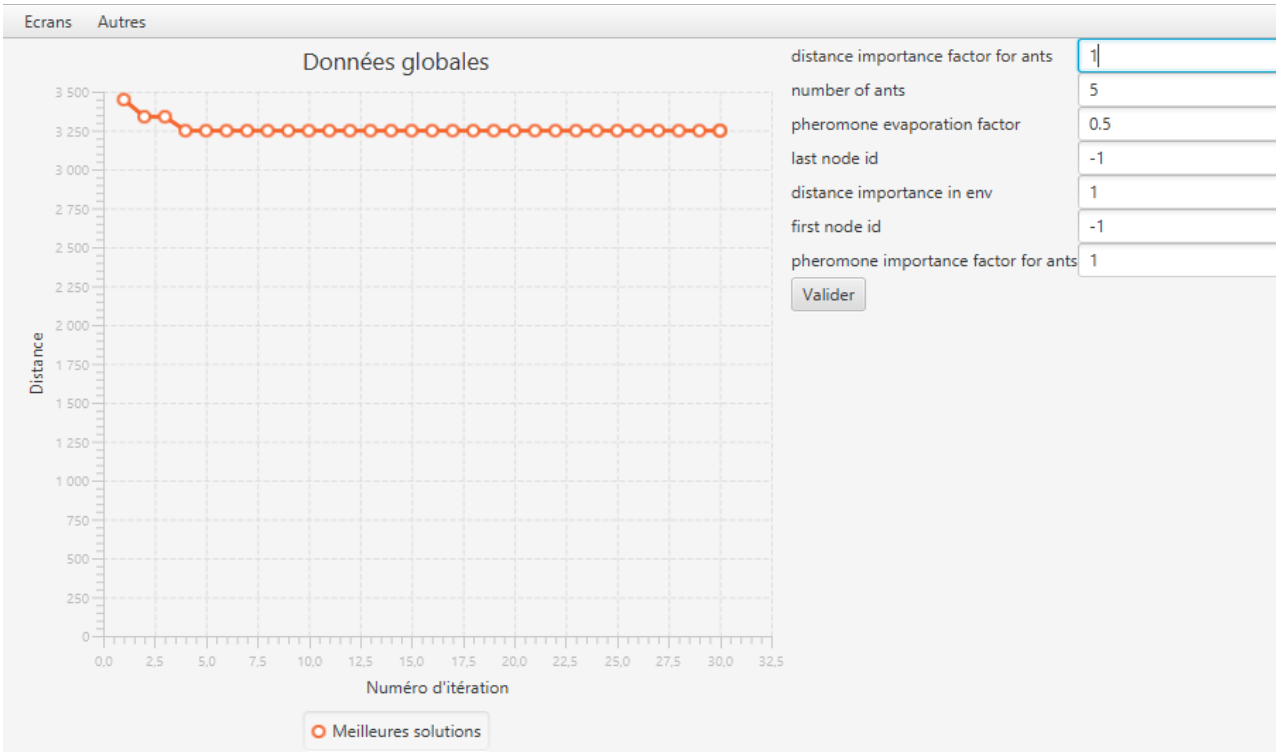

Illustration 10: Eccran de l'interface montrant l'évolution de la valeur objectif

Sur le bord droit de l'interface se trouve un onglet contenant la liste des paramètres modifiables, et un bouton de validation permettant d'appliquer les modifications. Aucune vérification de valeur n'est effectuée, aussi écrire une lettre, une valeur négative ou une valeur positive trop grande fera planter le programme. Une piste d'amélioration serait de régler ce problème en ajoutant des tests lors du parsing des valeurs, dans le code des agents.

Cet onglet occupe une large partie de l'écran, mais peut être affiché et caché à volonté via une option du menu supérieur.

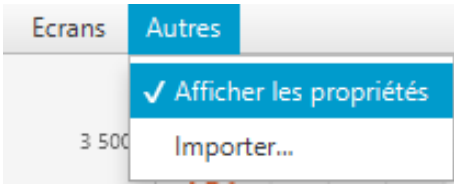

Illustration 11: Option du menu supérieur de l'interface permettant de masquer l'onglet latéral

L'import des fichiers de TSP se fait via une option du menu supérieur, qui permet d'afficher un sélecteur de fichiers.

La fermeture de l'interface lève un événement interne qui demande aux agents de s'arrêter eux aussi. Les agents s'arrêtent en général à la fin d'un cycle, aussi le programme peut mettre du temps à s'arrêter si une optimisation d'un TSP avec beaucoup de nœuds est en cours.

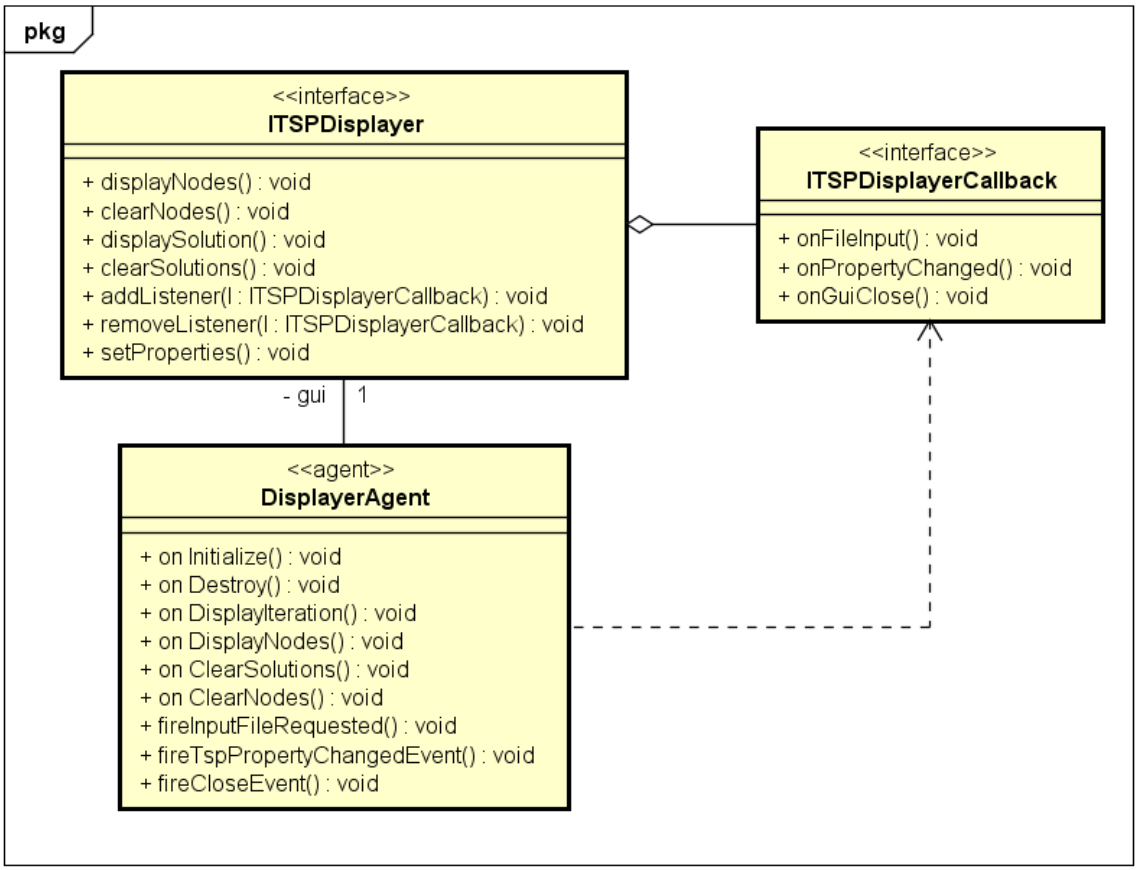

Illustration 12: Diagramme de classe représentant l'implémentation de la gestion de l'interface graphique

Du point de vue du code, la communication avec l'interface se fait par l'intermédiaire de deux interfaces java distinctes : l'une décrit un contrôleur de l'interface graphique qui permet l'affichage de nœuds et de solutions, ainsi que leur effaçage. L'autre décrit un *callback* pouvant être enregistré auprès de la première interface, et qui permet de recevoir des notifications lors d'événements spécifiques : import d'un nouveau fichier, modification d'un paramètre ou fermeture du programme.

L’agent afficheur est chargé de faire la passerelle entre l’interface graphique et le reste des agents du programme. Il reçoit les événements entrant de l’agent *Environment*, et émet les événements de l’interface graphique en broadcast, à destination de tous les agents du programme.

Librairies tierces

Comme écrit précédemment, le programme comporte deux projets annexes, écrit en Java. Le premier de ces projets contient le code l’interface graphique. Cette dernière a été réalisée avec l’API JavaFX. Étant donné que celle-ci ne fait pas partie de la notation du projet en IA54, nous ne détaillerons pas son contenu. Les deux interfaces java permettant d’interagir avec elle ont été décrites dans la section précédente.

Le deuxième projet est un parseur de fichier que nous avons écrit pour pouvoir lire des fichiers de *benchmark*, c’est-à-dire des fichiers contenant toutes les informations relatives à un problème de TSP :

- Nombre de nœuds
- Matrice des distances entre les nœuds
- Numéro de la variante du problème de TSP
  - 0 équivaut à *classique*
  - 1 équivaut à *bottleneck*
  - 2 équivaut à *TSP avec plages horaires*
- Heure de début du TSP (pour les variantes *avec plages horaires*)
- Détail des plages horaires (pour les variantes *avec plages horaires*)

```
1  # exemple de commentaire
2
3  NB_NODES=3
4  TSP_VERSION=2
5
6  NODES_DIST_SECTION
7  0 1 2
8  1 0 2
9  1 2 0
10 END_NODES_DIST_SECTION
11
12 NODE_TIME_SECTION
13 0-2 4-6
14 NONE
15 NONE
16 END_NODE_TIME_SECTION
17
18 TSP_INITIAL_TIME=1
```

Illustration 13: Exemple de contenu pour un fichier de benchmark

Ces fichiers se présentent sous la forme d’un fichier texte :

Chacun des champs ne contenant qu’une seule valeur sont écrit sur une seule ligne, avec le nom du champ suivi d’un caractère ‘=’ puis de la valeur. Sur la figure, ci-contre, on voit un fichier de TSP comprenant 3 nœuds et étant de la variante *avec plages horaires*.

Pour la matrice de distance, chaque ligne correspond à la distance entre un nœud (point de départ) et toutes les destinations envisageables (lui-même inclus, bien qu’il ne soit pas possible de passer 2 fois apr le même nœud). Par exemple, sur la figure ci-contre, la ligne 7, donc la première ligne de la matrice de distances, répertorie les distances des trajets partant du nœud 0 ($0 \rightarrow 0$, $0 \rightarrow 1$, $0 \rightarrow 2$), tous séparé par des espaces. Il en va de même pour les lignes suivantes.

Une autre section du fichier permet de spécifier les plages horaires. Cette section est délimitée par les balises *NODE_TIME_SECTION* et *END_NODE_TIME_SECTION*, et chaque ligne située entre

ces deux balises sont considérées comme des plages horaires (la première ligne pour le premier nœud, etc.).

Chaque plage horaire est définie par une heure de départ et une heure d'arrivée séparées par des tirets. Si un nœud possède plusieurs périodes d'accessibilité, elles sont écrites dans un ordre quelconque les unes à la suite des autres, sur la même ligne et séparées par des espaces. Si au contraire un nœud n'a pas de périodes particulières d'accessibilité, alors le mot-clé *NONE* permet de signifier que le nœud est accessible tout le temps.

Cette syntaxe de fichier a été créée car elle est simple à parser, simple à lire à l'oeil nu, et pour avoir une syntaxe commune pour tous les fichiers de benchmark, peu importe la variante et peu importe la source : les différents sites Internet où nous sommes allé chercher ces fichiers avaient tous des syntaxes différentes pour leurs données. Il nous a également fallu calculer les matrices de distance pour une grande partie des fichiers de benchmark, puisque seules les coordonnées des nœuds sont données sur la majorité des sites.

Performances

Nous allons maintenant présenter divers résultats que nous avons obtenu avec notre programme lors de l'optimisation de quelques TSP.

TSP classique

Nous avons utilisé 3 fichiers avec cette variante, et les paramètres ont été laissés à leurs valeurs par défaut.

Le premier fichier, *uk12* est un problème à 12 nœuds inspiré du Royaume-Uni, d'où son nom. La distance optimale de ce problème n'était pas spécifiée par la source où nous avons trouvé les données. La valeur optimale trouvée par notre programme est de 3 249, et cette valeur est trouvée quelques secondes par le programme, au bout d'environ 3 itérations.

Le deuxième fichier, *djibouti38* est un problème de 38 nœuds, et dont la distance optimale est de 6 656. Sur ce problème-ci, notre programme ne descend jamais en dessous de 32 878 pour la valeur optimale. Ce résultat aberrant est obtenu très rapidement lui aussi, mais n'est jamais battu. Nous avons essayé de relancer l'optimisation une trentaine de fois, en laissant tourner le programme une centaine d'itérations à chaque fois.

Le dernier fichier, *luxembourg980*, défini un problème à 980 nœuds, et dont la valeur optimale est de 11 340. Nous avons choisi de tenter d'optimiser ce problème en raison de son grand nombre de nœuds (le plus gros parmi les benchmark que nous avons trouvé), largement supérieur aux 500 que nous avions fixé comme seuil à atteindre. Les itérations sur ce problème durent moins d'une seconde avec les paramètres par défaut, ce qui indique que le programme semble bien se comporter même pour des problèmes de cette taille. Seulement, comme pour le problème précédent, la valeur optimale trouvée par le programme reste aberrante, puisque la meilleure valeur obtenue est 286 338.

TSP Bottleneck

Pour la variante *bottleneck*, nous avons testé qu'un seul fichier, *djibouti38_bottleneck*, créé en modifiant le numéro de variante du fichier *djibouti38* de 0 à 1. Bien évidemment, nous ne connaissons pas la valeur optimale de ce TSP. La valeur optimale trouvée par notre programme est de 10 040 et elle est trouvée en quelques itérations elle aussi.

TSP avec plages horaires

Cette variante n'étant pas fonctionnelle, aucun test d'optimisation n'a pu être réalisé.

Bugs, éléments non-implémentés et difficultées

Dans cette section, nous allons parler de divers éléments qu'il nous semble important de mettre en avant, que ça soit une fonctionnalité annoncée et non-fonctionnelle, ou une difficulté que nous avons rencontré et qui nous a posé problème.

Variante TSP avec plages horaires

Comme annoncé plus tôt dans la section sur les fourmis et sur l'algorithme que nous utilisons pour résoudre les problèmes de TSP (ici), la variante avec plages horaires n'est actuellement pas utilisable.

Limitation de l'interface graphique

L'interface graphique comprend comme nous l'avons dit précédemment un écran permettant de visualiser les nœuds sur un graph, mais du fait de la simplicité et de la rudimentarité de celle-ci, un trop grand nombre de nœuds rend cet écran rapidement illisible. Nous avons tenté de limiter cela en diminuant la taille des nœuds et en limitant le nombre de solutions affichées en même temps à l'écran (2 au maximum), mais celle-ci reste très rapidement surchargée.

Problèmes avec l'IDE SARL

Une grande partie de notre temps passé sur le projet a été passée à se battre avec l'IDE fourni pour coder en SARL (IDE Eclipse).

L'un des bugs les plus récurant est un bug déjà connu : le contenu d'un fichier SARL est parfois intégralement ou partiellement souligné en rouge, et est compté comme une erreur de compilation. La solution à ce problème nous était connue, mais cela n'en reste pas moins frustrant et chronophage. De plus, ce problème semble arriver systématiquement lorsqu'on utilise la fonction « clean project » en vue de recompiler l'intégralité du projet, ce qui arrive fréquemment lorsqu'on utilise des projets annexes écrit en Java et ajouté en dépendances du projet SARL puisque chaque modification de ces projets annexes demande une recompilation complète des binaires, sans quoi l'IDE ne prend pas en compte les modifications faites à ces projets annexes.

En plus de cela, de forts ralentissements ont été observés lors de l'utilisation de cet IDE, ralentissements nous empêchant presque d'utiliser nos ordinateurs portables pour travailler sur le projet. En effet, nos ordinateurs ont souvent été dépassés par la consomation en ressources de l'IDE,

des erreurs critiques étant levées au bout de quelques dizaines de secondes après avoir chargé le projet. Les ordinateurs les plus puissants tenaient le coup, mais les ralentissements se sont toujours fait sentir, et ce même pour les opérations les plus simples.

Un dernier problème notable est survenu maintes fois durant ce projet : des fichiers sont parfois indiqués dans l'arborescence du projet comme contenant des erreurs de compilation, alors que le fichier en lui-même ne contient pas d'erreurs. La solution à ce problème est, là encore, de recompiler le projet, mais recompiler le projet entraîne parfois ce problème sur d'autres fichiers. Au final, il nous est parfois arrivé de passer 5 minutes à passer de fichiers en fichiers pour faire disparaître ces erreurs « fantômes », tout ça pour finalement lancer le programme pour 5 secondes, vérifier le comportement du logiciel, et recommencer à modifier le code, ce qui lève d'autres erreurs « fantômes », etc.

Valeurs optimales aberrantes

Dernier point important que nous souhaitons mettre en valeur : durant toutes nos expérimentations avec la variante *classique*, les valeurs objectif calculées par le programme semblaient anormalement élevée, et, pour les benchmark dont nous possédions déjà la valeur objectif optimale, largement supérieure à ce qu'elle devrait être.

La source du problème ne semble pas résider dans l'algorithme, puisque ce dernier effectue bien les calculs voulus, sans modifier par erreur les valeurs.

Les données du benchmark elles-mêmes peuvent être la cause du problème. En effet, la matrice de distance a été calculée par nos soins à partir de la liste des coordonnées des points, en utilisant un calcul de distance euclidienne. Il est possible que ce calcul soit inadapté aux données que l'on a récupéré, rendant la valeur optimale, fournie par nos sources, inutile.

Bilan

Au final, ce projet a été un travail assez laborieux, la programmation dans un nouveau langage n'étant pas chose aisé et le nombre de problèmes qui sont survenus durant la réalisation de ce projet n'aidant pas.

Le programme est tout de même fonctionnel et possède les fonctionnalités de base que nous avions imaginé. Il reste bien sur de la place à l'amélioration pour ce programme, puisque tous les bugs n'ont pas été réglés et que toutes les pistes d'améliorations n'ont pas été explorées.