TP : Algorithmes Bio-Inspirés

Objectif du TP

L'objectif de cette séance est d'utiliser un Algorithme Génétique (AG ou GA – Genetic Algorithm) pour contrôler Ms. Pacman en java en utilisant la librairie ECJ (https://cs.gmu.edu/~eclab/projects/ecj/).

EXERCICES SUR MACHINES !

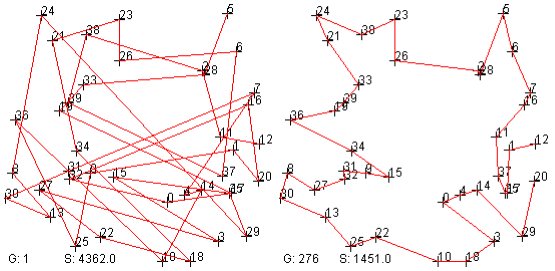
Avant d'utiliser un algorithme bio-inspiré pour pacman, je vous propose d'abord de résoudre le problème du voyageur de commerce pour apprendre à utiliser cette librairie.

Exercice 1 Installation de la librairie *Evolutionary Computation in Java* (ECJ)

Il existe beaucoup de librairies qui permettent d'utiliser et de construire ses propres algorithmes évolutionnaires pour tous les langages de programmation (Java, C#, Python, Scala, R, C++, Matlab…). Nous avons choisi ECJ car elle est accompagnée de plusieurs tutoriels d'utilisation et d'un manuel de 300 pages en plus d'une javadoc. Elle est très complète et illustre parfaitement les paradigmes de la programmation orientée objets en Java (héritage, introspection, Design Pattern…). De plus, les auteurs de cette librairie, des universitaires, sont très actifs et répondent même aux questions que vous pourriez leur poser par mail.

1. Ouvrir Eclipse et créer un nouveau projet (TSP par exemple).
2. Ajouter `ecj2020.jar` fourni dans le `Java Build Path` des propriétés de votre projet eclipse.
3. Télécharger également les librairies annexes utilisées par ECJ et ajouter les également au `Java Build Path` en tant que jar externes.

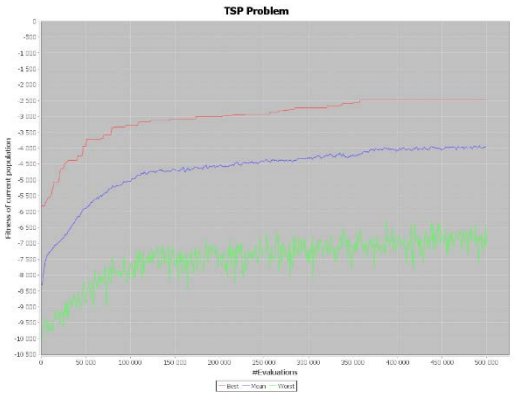
Exercice 2 Problème du voyageur de commerce (PVC ou Traveling Salesman Problem)



Le problème du voyageur de commerce est un problème d'optimisation combinatoire très classique et très connu : étant donné un ensemble de villes placées sur une carte (40 ci-dessus), l'objectif du commercial consiste à visiter une fois et une seule chaque ville (représentée par un numéro sur la figure) tout en minimisant la distance parcourue et revenir à son point de départ. Pour simplifier, on supposera qu'il n'y a qu'un seul chemin entre deux villes et qu'il est en ligne droite. C'est donc un problème d'optimisation qui peut être résolu par un algorithme génétique pour lequel :

- Une solution au problème = un ensemble d'entiers compris entre 1 et le nombre de villes à visiter, représentant le sens de déplacement du commercial, en sachant que chaque ville apparait qu'une seule fois dans le tableau (20→29→14→… dans l'exemple ci-dessus).
- L'évaluation d'une solution = distance parcourue par le voyageur (4362.0 pour la première figure)

1. Ajouter la classe TSP`main.java` à votre projet et lisez les commentaires.
2. Ouvrez le fichier TSP`Problem.params` avec un éditeur de texte (Notepad++ par exemple). Si vous utilisez Notepad++, vous pouvez modifier la coloration syntaxique en changeant le langage en `Makefile` ou `Properties`.



3. Comme vous pouvez le constater, la classe TSP`Main` charge le fichier texte que vous venez d'ouvrir et le transforme en une sorte de base de paramètres utilisables dans tout le programme. Ajouter ce fichier de configuration à la racine du projet eclipse.
4. ECJ distingue la notion d'espèce (`Species`) et d'individu (`Individual`) correspondant à une solution au problème que l'on souhaite optimiser. Une espèce est une représentation générique : par exemple un vecteur de bits, d'entiers ou de réels, un arbre, un réseau de neurones… Un individu appartient à une seule espèce. Il faut donc préciser dans le fichier de configuration quelle est l'espèce (et ses propriétés) et le type des individus manipulés par l'Algorithme Evolutionnaire (AE). ECJ propose de base plusieurs espèces et individus. Toutefois, le type d'individu `IntegerVectorIndividual` proposé par défaut qui pourrait nous intéresser ne permet pas les opérations de croisement, de mutation ou d'initialisation spécifique à un problème d'optimisation combinatoire (la ville ne doit apparaître qu'une seule fois). C'est pourquoi, nous avons créé la classe `IntegerCombinatorialIndividual` qui contient les opérations génétiques spécifiques à notre problème. Ajoutez cette classe à votre projet eclipse et dites à ECJ, dans le fichier de configuration que vous utilisez cette classe d'individus. Indice : il faut tenir compte de l'emplacement de la classe (package) par rapport au projet eclipse. Jetez un œil avisé à cette nouvelle classe.
5. Exécuter le projet eclipse. Vous devriez obtenir une erreur et c'est normal car ECJ tente d'instancier une classe correspondant au problème à optimiser. Ce problème correspond au problème du voyageur de commerce matérialisé par la classe TSP`Problem.java` que nous avons créé pour vous. Ce problème positionne un ensemble de villes (`Point`) aléatoirement sur une carte (méthode `setup` appelée automatiquement par ECJ lors de l'instanciation de la classe TSP`Problem`) et explique comment évaluer un individu (méthode `evaluate`). Ajouter cette classe au projet eclipse et modifiez votre fichier de configuration en conséquence.
6. Si vous exécuter le projet à nouveau, cela fonctionne mais rien ne semble se passer. Modifiez le paramètre `silent` de telle sorte que ECJ ne soit plus silencieux.
7. Difficile de comprendre ce qu'il se passe ! Pour cela, nous avons développé une classe qui permet de visualiser l'emplacement des villes et le chemin le plus court de la population courante trouvé par l'AE. Ajouter la classe TSP`Viewer.java` à votre projet et dé-commentez les lignes relatives à `viewer` dans la classe TSP`Problem`. Vous pouvez exécuter et obtenir les images précédentes.
8. Vous serez peut-être surpris par une valeur négative dans la valeur de fitness des individus. Enlevez le signe négatif dans la méthode `evaluate`, exécutez et constatez ! Une explication ? Nous avons utilisé l'équation $\min_x f(x) = \max_x -f(x)$ vu que, par défaut, ECJ maximise la valeur de la fitness de chaque individu.
9. Afin de mieux comprendre l'évolution de la qualité des solutions produites par l'AE, nous avons créé la classe `BestMeanWorstFitnessChartStatistics.java` capable de générer le graphique ci-dessous. En fait, pendant que l'AE s'exécute, il est possible, avec ECJ, de générer des fichiers de sorties à condition

de le préciser dans le fichier de configuration. Ces sorties sont appelées des « statistiques ». Actuellement, dans votre fichier de configuration, vous n'avez qu'une seule statistique (`stat.num-children=1`) référencée par `stat.child.0` et c'est la classe `SimpleShortStatistics` qui se trouve dans le package `ec.simple` qui s'occupera de générer les informations de sortie pour cette statistique. L'objectif est d'ajouter la création de l'image ci-contre. Pour cela, ajoutez une nouvelle statistique qui crée le graphique ci-contre en utilisant la classe `BestMeanWorstFitnessChartStatistics`. Comme cette dernière classe hérite de `ChartableStatistics`, cette statistique possède des propriétés comme le titre du graphique (`title`), le libellé de l'axe des abscisses (`x-axis-label`) et de l'axe des ordonnées (`y-axis-label`) que vous pouvez spécifier dans le fichier de configuration.

10. Amusez-vous (pas trop longtemps) à modifier le nombre de villes (`numcity`), la graine de génération de nombres aléatoires (`seed`), le nombre de générations de l'AE (`generations`), le nombre d'individus dans la population (`subpop.0.size`), la probabilité de mutation (`mutation-prob`), la taille du tournoi pour la sélection des parents (`tournament.size`)…

Exercice 3 Utilisation du TSP pour contrôler MS. Pacman

L'idée est d'utiliser ce que l'on vient de développer pour construire un contrôleur pour Pacman. Le principe consiste à identifier les endroits (les cases identifiées par des entiers) que Pacman doit visiter pour terminer le plateau. L'ensemble des « villes » que Pacman doit visiter correspond donc aux cases contenant des pastilles ou super-pastilles. Toutefois, il sera compliqué pour l'AG d'identifier un chemin optimal en si peu de temps pour beaucoup de pastilles (plus d'une centaine en début de partie). Par conséquent, on se propose de calculer un TSP sur un nombre réduit de pastilles : une heuristique parmi plusieurs possibles est d'identifier les `MaxNumberOfPills2Visit` pastilles les plus proches autour de pacman et d'appliquer un TSP sur ces pastilles. Toutefois, si un fantôme non comestible se trouvait entre deux villes que MS. Pacman devrait suivre, on se propose de pénaliser ce chemin en considérant que la distance entre ces deux villes est la distance maximale du plateau de jeu plutôt que la distance correspondant au plus court chemin.

En pratique, l'objectif consiste à programmer un nouveau contrôleur pour pacman qui, lorsqu'on lui demande de donner la direction à prendre (getMove), lancera un algorithme évolutionnaire pour trouver le plus court chemin entre un certain nombre de pastilles en tenant compte de la position des fantômes. Comme le temps disponible entre 2 appels à cette méthode est relativement court et insuffisant pour que l'algorithme évolutionnaire converge vers une solution optimale, on se propose de relancer complètement l'algorithme évolutionnaire (`eaState.run(EvolutionState.C_STARTED_FRESH)`) si pacman se trouve sur une case de jonction. A l'inverse, on se propose de continuer l'évolution là où elle s'était arrêtée lors du précédent appel (`eaState.run(EvolutionState.C_STARTED_FROM_CHECKPOINT)`).

1. Créer une nouvelle classe intitulée `EAPacman.java` héritant de `Controller<MOVE>`. Ce nouveau contrôleur pour pacman possède une variable globale de type `EvolutionState` et nommée `eaState` par exemple afin de garder une référence à l'AG entre deux appels de la méthode `getMove`.
2. Dans la méthode `getMove`, on doit vérifier si la variable `eaState` a déjà été initialisée ou non. Si ce n'est pas le cas, il faut la créer avec le code vu dans `TSPMain.java` (cf. Exercice 2 question 3).
3. Pour pouvoir évaluer une solution produite par l'AG, nous avons besoin d'avoir accès au jeu de pacman (`Game game`). Pour cela, il faudrait pouvoir instancier un AG (`EvolutionState`) avec un paramètre de type `Game`. Toutefois, cela n'est pas possible vu la manière dont l'AG s'instancie avec ECJ. Le seul moyen dont nous avons à notre disposition est d'utiliser la variable `Object[] eaState.job` qui est conservée pendant toute l'évolution. Dans cette variable qu'il faudra prendre soin d'initialiser, on peut donc y stocker la variable `game` passée en paramètres dans `getMove`. Faites-le juste après l'initialisation de `eaState`.

4. Maintenant, considérons que pacman se trouve sur une case de jonction. A l'aide de la méthode `getShortestPills` donnée sous forme de fichier, identifiez les cases que pacman doit visiter. L'ensemble de ces identifiants correspond aux valeurs possibles du problème d'optimisation combinatoire qui doit être préciser à l'AG grâce à la méthode `IntegerCombinatorialIndividual.setAllowedValues()`. Ensuite, relancez l'AG à partir de rien (C_STARTED_FRESH).
5. Si pacman ne se trouve pas sur une jonction, relancer l'AG là il où il avait été arrêté.
6. Arrivé à ce stade, l'AG s'est exécuté et nous pouvons récupérer le meilleur individu de la population qui contient le meilleur parcours entre toutes les pastilles précisées dans la question 4. Pour cela, inspirez-vous de la méthode `finishEvaluating` de la classe `TSPProblem` (cf. Exercice 2). Ensuite, vous pouvez récupérer l'identifiant de la case à laquelle pacman doit se rendre.
7. Nous avons défini comment et quand lancer l'AG dans la méthode `getMove`. Toutefois, il nous reste à définir comment l'AG évalue une solution qui représente un parcours entre plusieurs pastilles dans un labyrinthe. Pour cela, il faut créer une nouvelle classe, `TSPPacmanProblem.java` par exemple, qui contiendra la méthode `evaluate` (s'inspirer fortement de la même méthode de `TSPProblem.java`). Néanmoins, nous n'avons plus de `Point` mais un jeu de pacman matérialisé par le type `Game` accessible via la variable `job` (cf. question 3) de l'AG passé en paramètre de la méthode `evaluate` (souvent appelée `state` dans ECJ). Programmez l'évaluation de la solution produite par l'AG appelée `ind` en utilisant le jeu de pacman.
8. Modifiez le fichier de configuration des paramètres pour ce nouveau problème et tentez une exécution.
9. Vous obtenez un `nullPointerException` ! Normal, car la première case occupée par pacman n'est pas une case de jonction. Or, lorsque pacman est sur une case normale, nous avons programmé la relance de l'AG qui avait été lancé précédemment qui malheureusement n'a jamais été lancé encore. ECJ fait une différence entre initialisation de l'AG et exécution de l'AG au moins une fois : un certain nombre de variables ont été instanciées dans le deuxième cas. Je vous propose de tester la valeur à null de la variable population de l'AG pour faire la différence entre les deux cas. Solution :

POUR ALLER PLUS LOIN

Exercice 4 Programmation Génétique (évolution de programmes)

Comme nous l'avons vu en cours, les algorithmes évolutionnaires permettent de faire évoluer beaucoup de représentations informatiques et en particulier des arbres. Ces arbres peuvent représenter des programmes et/ou des règles de décisions (par exemple : si je suis loin de pacman et qu'il y a des pastilles proches alors manger les pastilles, etc …) et c'est ce qu'on appelle la programmation génétique . ECJ est particulièrement prévu pour développer de tels algorithmes évolutionnaires (cf. tutoriel 4). Daniel Tauritz explique comment adapter la programmation génétique au problème de pacman :
http://web.mst.edu/~tauritzd/courses/ec/cs348fs2010/cs348fs2010a2.pdf

Exercice 5 NeuroEvolution of Augmenting Topologies (NEAT)

Le principe consiste à faire évoluer des réseaux de neurones jouant à pacman.
https://www.youtube.com/watch?v=eoZLXCczs50

Sources pour ce TP :

A. Hussain et al., *Genetic Algorithm for Traveling Salesman Problem with Modified Cycle Crossover Operator*, 2017. https://doi.org/10.1155/2017/7430125