M1 informatique UL – cours métaheuristique

vthomas@loria.fr

Sujet 3 – Optimisation discrete – recherche locale

L'objectif de ce sujet est de construire des algorithmes de recherche locale (approche par réparation) pour résoudre plusieurs types de problemes

- le probleme des 4 couleurs
- le probleme du voyageur de commerce

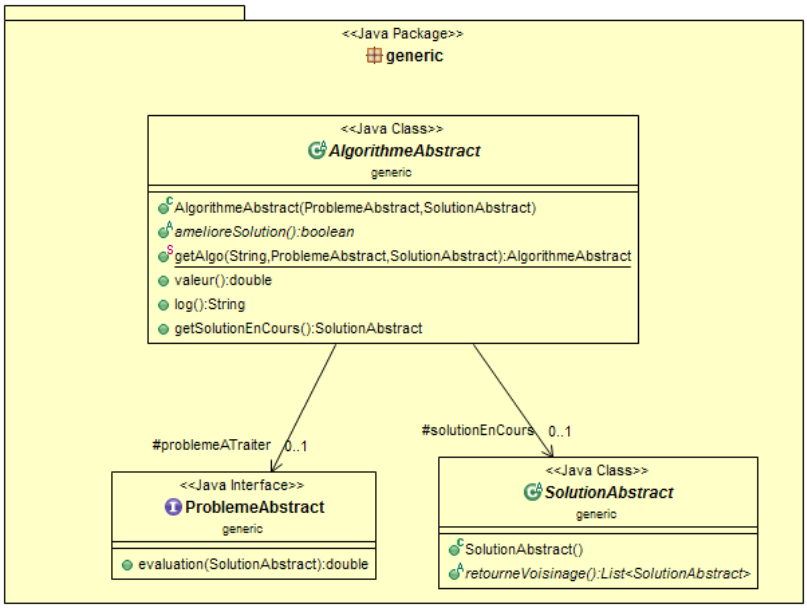
Vous serez amenés à écrire 3 algorithmes différents

- recherche locale greedy
- recherche recuit simulé
- recherche taboue

Package générique

Le package generic contient les classes permettant de définir les problemes, les solutions et les algorithmes

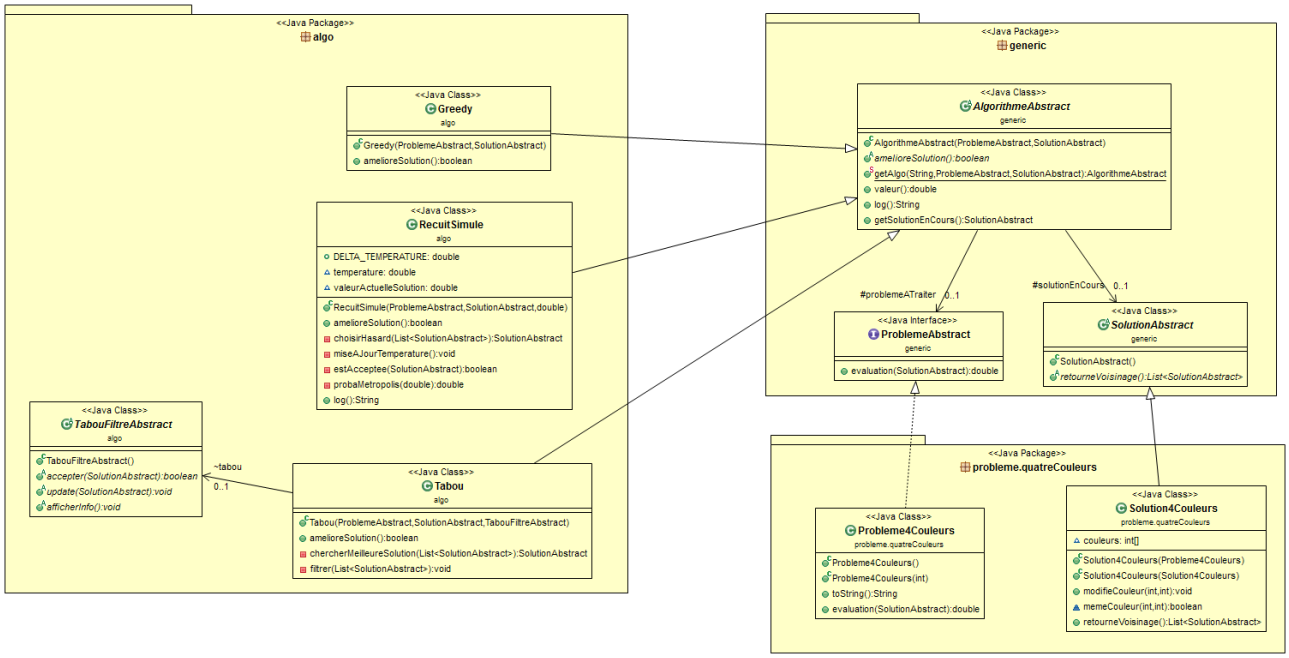
- un probleme est définie par la classe **`ProblemeAbstract`**. Cette classe est simplement caractérisée par une méthode evaluation(SolutionAbstract s) qui retourne l'évaluation d'une solution
- une solution est définie par la classe **`SolutionAbstract`**. Cette classe est caractérisée par une méthode `retourneVoisinage` qui retourne la liste des solutions voisines. Les solutions voisines caracterisent la manière dont on va explorer l'espace de recherche
- un algorithme est défini par la classe **`AlgorithmeAbstract`**. Construire un algorithme consiste à partir d'un probleme et d'une solution initiale et à appeler la méthode ameliorerSolution pour amelioerer la solution initiale de proche en proche



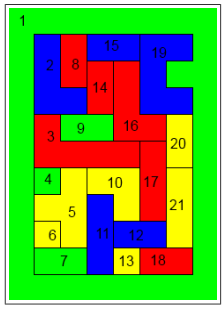
Probleme des 4 couleurs

Le probleme des 4 couleurs vous est fourni. Il est défini dans le package **`probleme.4Couleurs`**.

- La classe **`Probleme4Couleur`** défini un probleme 4 couleurs.
  - Il est caractérisé par des zones à colorier (dans un tableau 2d) et une matrice d'adjacence qui permet de savoir si deux zones sont adjacentes.
  - Un constructeur par défaut construit un probleme particulier, mais il est possible de créer des problemes aléatoires de taille n (qui passe par une construction complexe)
  - l'évaluation d'une solution consiste à compter le nombre de contraintes non respectées (à savoir le nombre de fois qu'une zone a la meme couleur qu'une zone qui lui est adjacente)
- une solution est définie dans la classe **`Solution4Couleurs`**.
  - Une solution associe à chaque id de zone une valeur de couleur (entre 0 et 3)
  - la méthode retourne voisinage retourne toutes les solutions pour lesquelles on a changé une zone dans une nouvelle couleur (le nombre de solutions voisines est donc de 3 fois le nombre de zones)
- la classe **`Afficher4Couleur`** permet d'avoir un rendu graphique du probleme et d'appeler les algorithmes développés (la class **`TestProbleme4Couleurs`** permet de lancer l'application)

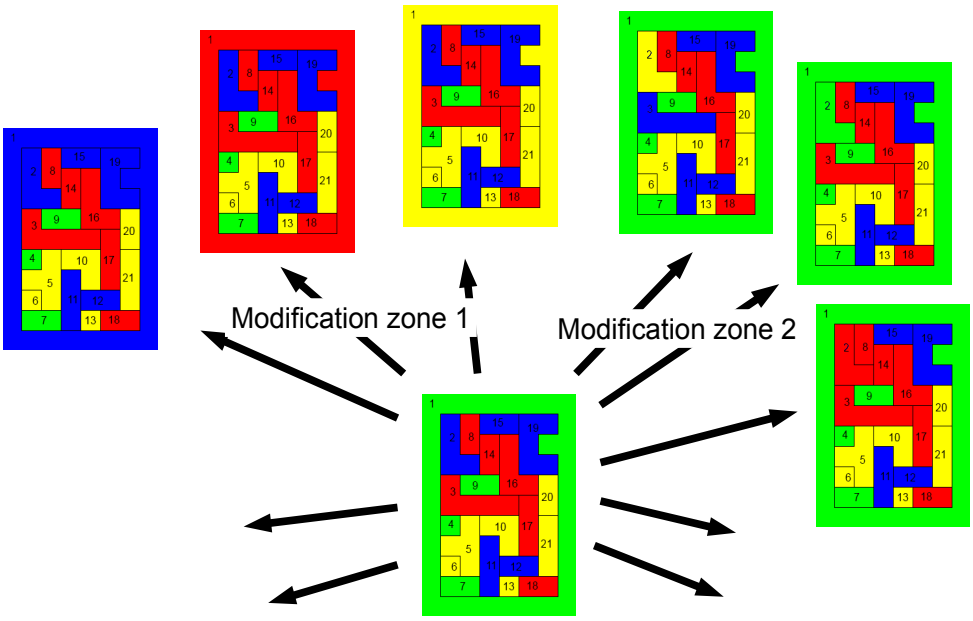


L'exemple ci contre présente la manière dont le probleme est représenté (probleme du constructeur par défaut):



- chaque zone a un identifiant unique (cf numeros ajoutés)
- une solution consiste à attribuer une couleur à chaque zone
- Il est possible de changer directement la couleur d'une zone en cliquant dessus.

Les opérateurs de voisinages consistent à modifier la couleur de chaque zone et sont présentés ci dessous



Algorithme Greedy

Compléter l'algorithme **`Greedy`** qui améliore une solution de proche en proche en choisissant à chaque fois la meilleure solution du voisinage.

Lorsque cette solution est appliquée sur l'exemple fourni, elle converge vers une solution avec 3 contraintes brisées (cf classe **`TestTous`**).

Algorithme RecuitSimulé

Compléter l'algorithme **`RecuitSimule`** qui améliore la solution de proche en proche en acceptant des dégradations (selon une loi de métropolis).

- La méthode **`amelioreSolution`** prend au hasard une solution du voisinage et vérifie si elle l'accepte ou non (ou limitera à un nombre de tirages fixés pour éviter de rester bloqué)
- la méthode **`choisirHasard`** retourne une solution au hasard
- la méthode **`estAcceptee`** retourne un booléan qui vaut vrai si la solution est acceptée. Elle utilise la loi de métropolis.
- La méthode **`probaMetropolis`** retourne la probabilite en fonction de la différence de valeur et la temperature.
- La méthode **`miseAJourTemperature`** est fournie et baisse la temperature par un facteur multiplicatif donné. Elle permet de passer progressivement d'une exploration totale à une exploitation totale.

Contrairement au greedy, l'algorithme recuitSimulé parvient quasiment tout le temps à converger vers la solution optimale (sans contrainte) y compris pour de grands problemes (cf classe **`TestProbleme4Couleurs`** et l'interface graphique)

Algorithme Tabou

Complétez l'algorithme **`Tabou`**. Cet algorithme utilise un objet **`tabou`** qui implémente la classe **`TabouFiltreAbstract`**.

La classe **`TabouFiltreAbstract`** permet de rejeter des solutions au fur et à mesure de

l'exploration. En faisant cela, elle force l'exploration des voisins qui n'ont pas déja été visités. Elle se base sur deux méthodes

- **`accepter`** qui retourne un boolean qui vaut vrai si et seulement si la solution n'est pas rejetée.
- **`update`** qui met à jour le filtre en fonction de la dernière solution rencontrée.
- **`AfficherInfo`** permet simplement d'afficher les informations sur le filtre

On vous donne la classe **`TabouFiltreEtat`** dans le package 4couleurs. Cette classe implemente un filtre en stockant et en interdisant les états deja rencontrés.

Completer la classe **`Tabou`**. Cela consiste à

- compléter la méthode **`amelioreSolution`** en filtrant les solutions du voisinage en fonction du filtre **`TabouFiltreAbstract`** et en selectionnant la meilleure solution restante. Cette méthode utilise les méthodes **`chercherMeilleureSolution`** et **`filtrer`**
- compléter la méthode **`filtrer`** qui retire du voisinage les solutions filtrées par le filtre
- compléter la méthode **`chercherMeilleureSolution`**

Filtre par attribut

Ecrire le filtre **`TabouFiltreAttribut`** pour filtrer les solutions pas par état mais par attribut: on va desormais interdire une solution dont on vient de modifier une zone qu'on a retenue comme ayant deja été modifiée.

- L'attribut **`zonesRefusees`** stocke la liste des zones sont on refuse une modification
- L'attribut **`derniereAcceptee`** permet de stocker la derniere solution acceptée (pour pouvoir calculer quelle est la zone modifiée lorsqu'on verifie si une solution est acceptée)
- la méthode **`chercherDifferente`** fournie permet de calculer la zone ayant une couleur différente par rapport à la derniere solution
- il faut donc
  - compléter la méthode **`accepter`** pour décider quand accepter une solution
  - compléter la méthode **`update`** pour mettre à jour la liste des zones dont on souhaite interdire la modification.

Probleme du TSP

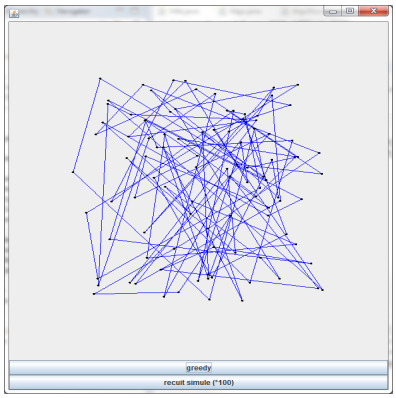
On vous demande enfin de représenter le probleme du TSP pour pouvoir utiliser les memes algorithmes (greedy et ercuit simulé) développés initialement pour le probleme 4 Couleurs.

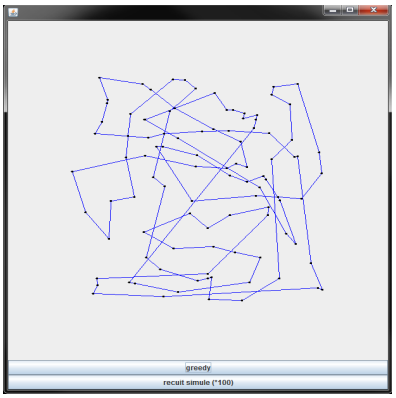
- La classe **`AfficheTSP`** fournie permet d'afficher le probleme du TSP et de lancer les algorithmes (la classe **`TestProblemeTSP`** permet de lancer l'interface graphique)
- la classe **`SolutionTSP`** présente une solution, mais il faut completer la méthode **`retourneVoisinage`**. Cette méthode retourne la liste des solutions avec deux villes échangées.

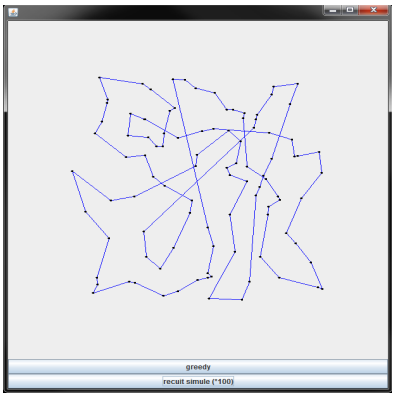
- Cette méthode suffit à définir un graphe d'exploration dans l'espace de recherche et donc à appliquer les techniques précédentes

Vous pouvez aussi développer un filtre associé au probleme TSP pour une recherche taboue (cf cours: on interdit d'echanger des villes deja échangées dans le passé).

Ci dessous, voici le résultat d'une execution







la premiere image présente le probleme initial (valeur de `20452`), la seconde le résultat avec une approche greedy (valeur de `5917`), la troisieme la solution obtenue avec une approche recuitSimulé (valeur de `4277`) dont la temperateur initiale est de 10000 et un facteur d'attenuation de 0,999

Il est à noter que la température et son facteur de décroissance dependent fortement de l'ordre de grandeur des évaluations des solutions. Augmenter le nombre de villes augmente les variations des évaluations et necessite de plus grandes temperatures et des décroissances plus faibles.