Application d'optimisation
Conception générale

J.C. Créput

Janvier 2015

Contents

Introduction

Ce document répond à une problématique d'optimisation type. La solution logicielle répond aux spécifications fournies dans le document :

- *cahier_des_charges.pdf*

Le problème d'optimisation traité est un problème NP-difficile fictif, traité par méta-heuristique. La très grande combinatoire du problème et l'étude des approches d'optimisation de la littérature en recherche opérationnelle amène à considérer son traitement par l'usage des méthodes métaheuristiques. Ce document présente une solution type pour résoudre le problème. La méthode est basée sur des métaheuristiques de type recherche locale et algorithme évolutionnaire et mémétique.

Les différentes sections sont organisées pour faire ressortir les principaux composants et étapes de la conception et modélisation de l'algorithme d'optimisation et son implantation en C++. Les chapitres présentent successivement les éléments suivants :

- *Définition du problème d'optimisation*
- *Principe de résolution*
- *Structures de données*
- *Opérateurs de base*
- *Algorithme métaheuristique*
- *Jeux de tests et évaluation*

Une conclusion générale termine le document.

Chapter 1

Définition du problème d'optimisation

Le problème est apparenté au problème de flôt optique (*optical flow*).

Ici, nous donnons la définition du problème d'optimisation, ce qui revient à préciser ses entrées, ses sorties, et les contraintes et objectifs du problème.

1.1 Entrée/sortie du programme d'optimisation

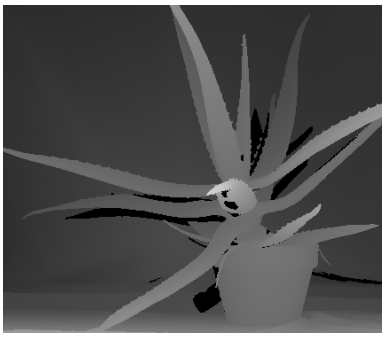

Figure 1.1: Entrée : exemple d'entrée

Le programme d'optimisation est une fonction d'optimisation qui transforme une donnée d'entrée en une solution de sortie, en principe optimale au sens des contraintes et objectifs du problème. La donnée d'entrée est présentée par la figure 1.1. Elle comporte différents éléments. Un exemple de solution est donné à la figure 1.2.

Figure 1.2: Sortie : exemple de solution

1.2 Fonction objectif

1.2.1 Fonction objectif globale agrégative

La fonction objectif globale à minimiser est une somme pondérée des objectifs définissant les critères et contraintes du problème :

$$objectif = f\_objectif\_1 * weightObjective1 + f\_objectif\_2 * weightObjective2 + f\_objectif\_3 * weightObjective3$$

La fonction objectif globale est réalisée par la fonction C++ :

- *double SolutionObject::computeObjectif(void)*

Les différents objectifs et critères sont précisés dans les sections suivantes.

1.2.2 Objectif 1

1.2.3 Objectif 2

1.2.4 Objectif 3

1.3 Définition du problème d'optimisation

Le problème d'optimisation est défini par une entrée, l'instance du problème, et une sortie, la solution du problème, dont l'évaluation doit correspondre à une valeur minimum de la fonction objectif agrégative.

Une instance du problème consiste en différents composants. La solution fournit des valeurs des variables du problème. Le but est de fournir une solution optimale au sens de la minimisation de la fonction objectif agrégative.

Une solution est dite admissible lorsque toutes les contraintes sont satisfaites. La satisfaction d'une contrainte revient à une mise à 0 de certains objectifs.

Chapter 2

Approche d'optimisation

Les problèmes de flôt optiques avec contraintes sont NP-difficiles. Nous n'avons pas trouvé, dans la littérature sur le sujet, de méthode de résolution exacte de ce type de problème répondant à nos besoins. Une instance avec très peu d'éléments comporte déjà une combinatoire de type factorielle qui rend très difficile la résolution exacte en temps raisonnable. Il paraît justifié de tenter de résoudre le problème par des méthodes heuristiques et métaheuristiques. Dans ce cas, nous ne garantissons pas l'obtention systématique d'un optimum global, ce qui prendrait un temps infiniment long, mais nous cherchons des solutions admissibles et de bonne qualité en temps raisonnable. Pour vérifier l'efficacité, les performances sont validées expérimentalement sur des jeux de tests représentatifs du problème concret. Nous abordons dans ce chapitre la présentation des principes de base des méthodes métaheuristiques proposées.

2.1 Stratégie d'optimisation

La démarche de résolution, appelée aussi stratégie de recherche ou d'optimisation, est une démarche heuristique, fondée sur le principe des recherches locales d'une part, et sur le principe des algorithmes à base de population de solutions, tels que les algorithmes évolutionnaires, d'autre part.

De manière imagée, il s'agit de faire évoluer pas à pas une solution de départ, très approximative et ne respectant pas nécessairement les contraintes, vers une solution admissible satisfaisant toutes les contraintes du problème, via des séquences de mouvements simples et opérations de *swap*, plus ou moins aléatoires, portant sur les composants de la solution. Ces mouvements sont réalisés conjointement aux réévaluations rapides des

valeurs des objectifs impactés. Ce sont ces évaluations qui déterminent et orientent la sélection des mouvements réalisés.

Chaque mouvement de base de la solution comporte l'évaluation des objectifs liés au composant considéré, le mouvement proprement dit, puis l'évaluation de l'impact du mouvement sur les valeurs des objectifs. Si l'impact est favorable, la nouvelle solution est retenue comme candidate à de nouvelles modifications, sinon elle n'est pas considérée et la solution précédente est à nouveau utilisée comme point de départ ou pivot de la recherche locale.

La recherche locale met en oeuvre une démarche d'optimisation par réitération de mouvements locaux via les opérateurs de base de la résolution. Un des inconvénients de la recherche locale réside dans le minimum local où elle peut se trouver lorsque les mouvements de faible amplitude ne sont pas suffisants à faire évoluer favorablement la solution. La réponse apportée ici réside, ou bien dans la réitération du processus d'optimisation à partir de solutions aléatoires initiales différentes, ou bien dans l'utilisation d'une démarche évolutionnaire mémétique.

Le principe informel de l'algorithme évolutionnaire mémétique consiste à appliquer des recherches locales sur une population de solutions, d'autres opérations éventuelles, et à procéder à des sélections et remplacements de solutions. Les solutions les moins satisfaisantes sont remplacées par les plus satisfaisantes, à chaque itération, appelée une génération. La meilleure solution produite est fournie en sortie.

Dans cette section, nous ne donnons que le principe général des méthodes sans entrer dans le détail des algorithmes. Les opérateurs de base ainsi que les pseudo-codes des méthodes de recherche sont présentés en détail dans les chapitres suivants.

2.2 Principe de la recherche locale

La recherche locale consiste en une modification locale d'une solution courante en considérant un voisinage plus ou moins large de cette solution. Cette solution courante, pas nécessairement admissible (satisfaisant toutes les contraintes), est obtenue au départ par une méthode de construction rapide. Les modifications locales déterminent une intensification de la recherche dans une zone plus ou moins restreinte de l'espace des solutions. La réitération de la recherche locale à partir de solutions initiales aléatoires permet de diversifier la recherche [1].

En se basant sur une solution de départ préalablement construite mais pas nécessairement admissible, trois versions de la stratégie de recherche locale sont considérées. La première version est une méthode de recherche aléatoire itérée, de type marche aléatoire, que nous appelons IRS, pour *Iterated Random searh*. La deuxième est une méthode de recherche locale gloutonne de type premier-meilleur que nous appelons ILS-FI, pour *Iterated Local Search First Improvement*. La troisième est une recherche locale en profondeur que nous appelons ILS BI pour *Iterated Local Search Best Improvement*.

La méthode IRS fait évoluer une solution courante en effectuant un nombre donné de mouvements successifs dans le voisinage, exécutés via un opérateur de voisinage, puis sélectionne la meilleure solution rencontrée. Chaque solution examinée est obtenue par modification de la solution précédente à l'aide de l'opérateur de voisinage. Une suite de modifications de la solution courante est exécutée, leur nombre étant préalablement fixé. La meilleure solution rencontrée est sélectionnée. Le procédé complet est réitéré un certain nombre de fois après permutation éventuelle des épures pour favoriser l'émergence d'un ordre valide de chargement. La méthode est très simple puisque très peu d'opérations de copie de données sont effectuées à chaque d'itération.

Dans les recherches locales ILS-FI et ILS-BI, une solution donnée constitue l'élément pivot autour duquel est effectuée la recherche dans le voisinage. Chaque nouvelle solution examinée est obtenue par modification de la solution pivot avec l'opérateur de voisinage. Les deux versions diffèrent par la règle de pivotage choisie. Dans ILS-FI, la première solution rencontrée de qualité supérieure à la solution pivot devient le nouveau pivot. Dans la version ILS-BI, la meilleure solution rencontrée à l'intérieur d'un échantillonnage du voisinage est sélectionnée comme le nouveau élément pivot. Seul un échantillon de taille limitée du voisinage est examiné.

Dans les deux versions ILS-FI et ILS-BI, la recherche locale est arrêtée lorsqu'aucune amélioration n'est trouvée, c'est-à-dire lorsqu'on atteint un minimum local. En pratique, seul un échantillon du voisinage est examiné car il s'agit ici d'un voisinage large avec un trop grand nombre de voisins et d'un opérateur stochastique. La taille du voisinage est déterminée par le nombre de choix possibles d'épures auxquelles s'appliquent les différents opérateurs élémentaires de mouvement suivant un grand nombre de choix possibles, à chaque pas d'exécution. La taille du voisinage ne doit pas être confondue avec la taille de l'échantillon examiné, c'est-à-dire le nombre maximum de solutions examinées obtenues à partir d'un élément pivot.

Ensuite, le procédé complet est réitéré un certain nombre de fois après permutation des épures ou réinitialisation de la solution via une nouvelle construction de départ.

2.3 Principe de l'algorithme mémétique

Nous proposons une autre manière d'explorer l'espace des solutions suivant le paradigme des algorithmes évolutionnaires et mémétiques [2, 3]. La recherche procède maintenant par une application simultanée des opérateurs de base, de voisinage et de construction, à un ensemble (une population) de solutions. Suivant la terminologie des algorithmes évolutionnaires, les solutions sont maintenant des « individus » qui doivent répondre aux exigences du problème.

Pour évaluer la qualité de la solution, une fonction appelée *fitness* associe une valeur scalaire à chaque individu de la population. Cette fonction de fitness est une adaptation directe de la fonction objectif globale agrégative. Etant donné que nous représentons le problème par une minimisation de la fonction objectif, et non pas une maximisation, la fitness est ici de valeur opposée. Elle est définie de façon à ce que les « meilleurs » individus ont une plus grande fitness, tandis que les « mauvais » individus une fitness plus petite.

Des opérateurs de sélection permettent le remplacement des « mauvaises » solutions de la population par les « meilleures » de la population. Un ou plusieurs opérateurs de mutation participent à la diversification et l'amélioration des solutions. Il sont mis en œuvre par les opérations de base de manipulation des imbrications.

Si nous utilisons une recherche locale au sein d'un l'algorithme évolutionnaire, nous obtenons un algorithme évolutionnaire de type mémétique [2]. La figure 2.1 nous permet de schématiser le principe de recherche de solution par l'algorithme mémétique. Sur la figure sont illustrées les recherches locales exécutées en parallèle sur les individus de la population et conduisant à un minimum local. La multiplication de ces recherches locales couplée aux opérateurs de sélection et de mutation détermine la dynamique de recherche. L'objectif est de réutiliser les avantages de la recherche locale tout en fournissant des mécanismes de diversifications pour éviter d'être piégé dans un minimum local.

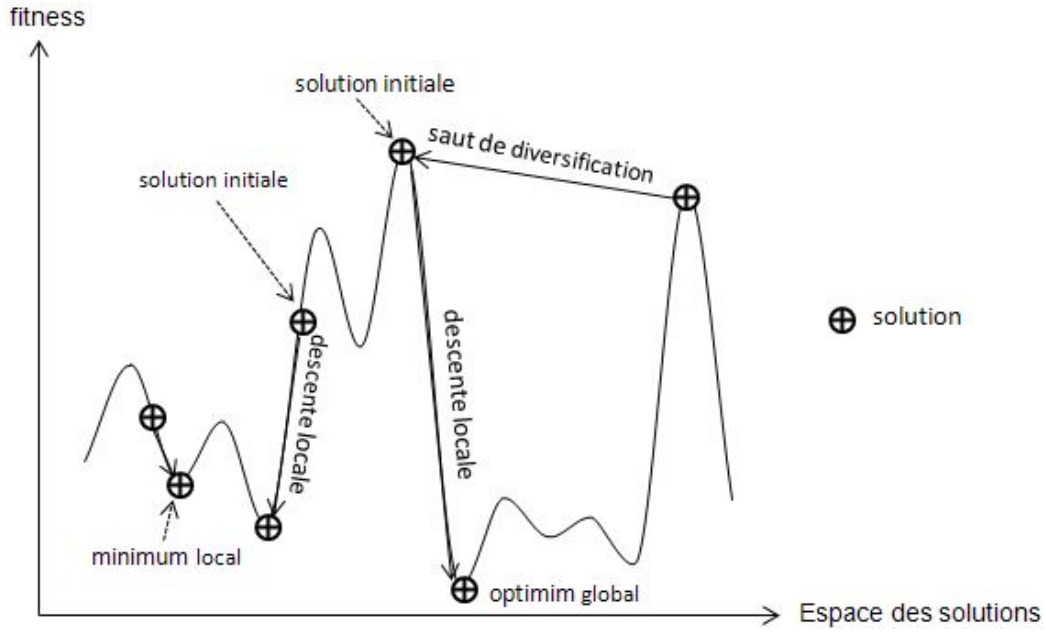

Figure 2.1: Principe de l'algorithme mémétique.

Chapter 3

Structures de données

L'utilisation de structures de données adaptées aux procédures d'optimisation est un élément clé du processus de conception. Un des principes clés réside dans l'utilisation de tableaux à accès direct indicés par identifiant.

Nous donnons ci-dessous, une description des éléments clés de la modélisation orientée objet sous forme de diagramme UML. Sont présentés, les composants de base, les tableaux de données, l'environnement commun d'optimisation.

3.1 Structure position

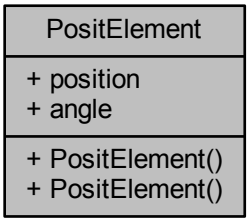

Figure 3.1: Position d'un élément/composant.

Tout composant géométrique est caractérisé par un point de référence

et un angle d'orientation. Sa position dans le plan est déterminée par la position du point de référence dans le plan et l'angle avec l'horizontale. Le diagramme de classe UML est donné par la figure 3.1.

3.2 Objet solution

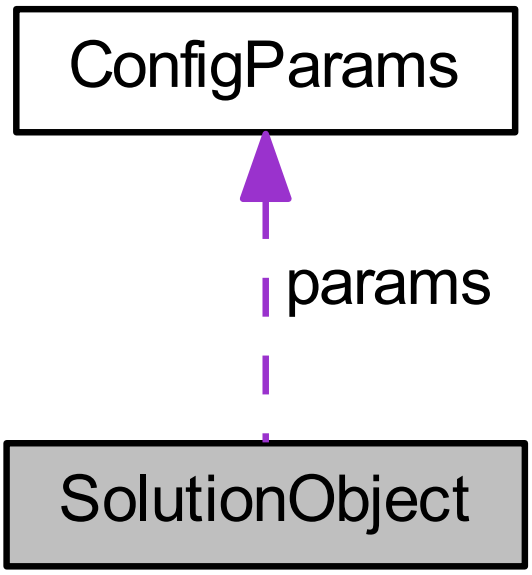

Figure 3.2: Classe principale solution.

La classe *SolutionObject* est la structure principale représentant une so-

lution du problème. Elle comporte toutes les variables de la solution, variables de positionnement, d'ordonnancement. Chaque solution manipulée et/ou construite par l'algorithme d'optimisation est une instance de la classe *SolutionObject*. Une solution peut être dupliquée et reproduite à volonté à l'aide de l'opérateur d'affectation standard (=). La solution est un objet manipulé par les algorithmes métaheuristiques de résolution. Le diagramme de classe UML est donné par la figure 3.2. Seule la classe de configuration *ConfigParams* apparait ici.

Chapter 4

Opérateurs de base

Deux types de méthodes d'optimisation sont utilisées. Ce sont respectivement la recherche locale itérée et un algorithme mémétique dont nous avons donné les grandes lignes précédemment et que nous décrivons plus en détail dans le chapitre suivant. Ces méthodes ont en commun un ensemble d'opérateurs de base opérant sur la structure d'une solution.

Dans les deux cas de figure, une ou plusieurs solutions sont modifiées de manière itérative selon un opérateur de voisinage qui transforme un ou plusieurs items (épures) d'une solution courante. Cet opérateur de voisinage est composite en ce qu'il combine différents opérateurs de base ayant différentes fonctions (mouvement, swap). L'opérateur est stochastique en ce que le choix de la modification locale est en partie aléatoire. Plusieurs types d'opérateurs sont proposés. Enfin, l'opérateur de voisinage composite effectue une combinaison probabiliste des opérations de base.

4.1 Opérateur de mouvement

L'opérateur de mouvement élémentaire est illustré par la figure 4.1. Il permet les opérations de mouvements. Cet opérateur de base est appelé par l'opérateur de voisinage composite avec une probabilité élevée (environ 0,9). Il constitue l'opération de base la plus active dans le processus d'optimisation.

Figure 4.1: Opération de mouvement.

Figure 4.2: Opération de swap.

4.2 Opérateur de swap

L'opérateur de swap est illustré par la figure 4.2. L'opération est appliquée par l'opérateur composite avec une probabilité assez faible (environ 0,01).

4.3 Opérateur de construction initiale

L'opérateur de construction initiale cherche à produire une solution aussi vite que possible. Le résultat n'est pas nécessairement une solution admissible et peut présenter des contraintes non satisfaites, ainsi qu'illustré à la figure 4.3. Le pseudo-code est donné dans l'Algorithme 1. Noter que l'opérateur de construction n'est appelé qu'une seule fois pour initialiser une solution. Les recherches locales, et les procédés d'amélioration métaheuristiques peuvent alors démarrer.

Figure 4.3: Construction avec bruit aléatoire.

Algorithme 1: Construction initiale (constructSolutionSeq)

Entrées : $S \in$ Solution
Sorties : S
début
  `// Les composants sont préalablement triés`
  tant que *tous les composants ne sont pas placés* **faire**
    pour *chaque composant* $v \in S$ **faire**
      Placer le composant selon une stratégie de construction;
  retourner S;

4.4 Opérateur de voisinage

L'opérateur de voisinage composite est utilisé au sein d'une stratégie de recherche locale ou mémétique. Il applique l'ensemble des opérateurs de base suivant une probabilité d'exécution spécifique à l'opérateur. Noter que l'opérateur de construction n'est pas concerné puisqu'il fournit seulement la solution de départ qui va être améliorée. L'opérateur de voisinage composite vise à transformer, au pas à pas, une solution non admissible de départ en une solution admissible avec un coût le plus réduit possible en temps d'exécution. Son exécution inclut la réévaluation incrémentale de la fonction objectif au sein de chaque opérateur élémentaire, ce qui est crucial pour le temps d'exécution. Le pseudo-code est résumé par l'Algorithme 2. La fonction de voisinage composite C++ opérant sur une solution correspond au profil *void SolutionObject::generateNeighbor()*.

Algorithme 2: Opérateur de voisinage composite (generateNeighbor)

Entrées : $S \in$ Solution
Sorties : S
début
    // Sélectionner un des opérateurs de base selon sa probabilité d'application
    // Appliquer l'opérateur i sélectionné à la solution S
    $S \leftarrow operator_i(S)$;
    retourner S;

Chapter 5

Algorithme métaheuristique

Le développement de méthodes heuristiques est a priori justifié par la difficulté du problème d'imbrication qui est NP-difficile. Les heuristiques considérées sont trois variantes de recherches locales itératives, opérant sur une solution unique préalablement construite mais non nécessairement admissible, un algorithme évolutionnaire opérant sur une population de solutions et un algorithme mémétique incorporant une recherche locale au sein d'un algorithme évolutionnaire. En définitive seuls la recherche locale *First Improvement* et l'algorithme Mémétique semblent vraiment performant a priori sur un problème quelconque. Nous détaillons cependant les différentes versions.

La recherche locale fonctionne par des mouvements de faible amplitude dans un voisinage de la solution courante. La fonction de voisinage détermine la solution suivante obtenue à partir d'une modification locale de la solution courante. Lorsqu'aucune amélioration de la solution ne peut être obtenue dans le voisinage, la solution en cours est un optimum local relatif à la fonction de voisinage. La réitération de la recherche locale à partir de conditions de départ aléatoires permet de diversifier les zones d'exploration.

L'opérateur de voisinage composite est à la base du principe de la recherche locale. Nous avons retenu trois versions de la recherche locale suivant le mode d'examen du voisinage retenu. La première version est une recherche de type marche aléatoire. Les deux autres versions sont une recherche gloutonne *first improvement search* et une recherche en profondeur *best improvement search*.

En combinant les mêmes opérateurs de base dans un schéma d'algorithme à base de population, avec des opérateurs de sélection sur les solu-

tions, nous spécifions un algorithme évolutionnaire. Puis, pour tenter de tirer parti des avantages de l'algorithme évolutionnaire et de la recherche locale conjointement, nous incluons celle-ci en tant qu'opérateur dans un algorithme évolutionnaire selon le principe d'un algorithme mémétique.

Dans la section 5.1, sont présentées les recherches locales itérées selon trois versions. L'algorithme évolutionnaire opérant sur une population de solutions, et l'algorithme mémétique, incluant la recherche locale, sont présentés dans la section 5.2.

5.1 Recherche locale itérée

Les recherches locales combinent les opérateurs de base selon différentes stratégies de recherche. Elles opèrent toutes à partir d'une solution unique préalablement construite et non nécessairement admissible. Elles diffèrent par le mode de parcours d'un voisinage autour de la solution courante. Elles sont réitérées avec des ordonnancements de véhicules modifiés.

5.1.1 Boucle externe

La recherche locale itérée [1] est l'une des méthodes les plus simples de recherche heuristique appliquées aux problèmes NP-difficiles. La boucle principale de la méthode consiste à réitérer l'exécution de recherches locales simples à partir de conditions initiales aléatoires. Le pseudo-code de la boucle principale est présenté dans l'Algorithme 3.

Les opérations sont reparties en deux phases: une phase de construction suivie d'une phase d'amélioration. Dans l'Algorithme 3, une tentative de construction suivie d'une tentative d'amélioration sont effectuées par les deux appels *constructSolution* et *improveSolution*. Les détails de ces deux procédures sont donnés respectivement dans les sections 5.1.2 et 5.1.3. La construction génère rapidement des solutions partielles non nécessairement admissibles. A partir de la solution obtenue, la procédure d'amélioration applique des modifications locales en vue d'améliorer la solution.

Dans tous les cas de figure, il convient d'évaluer et de comparer des solutions non nécessairement admissibles. C'est pourquoi, le classement des solutions s'effectue selon la valeur de la fonction objectif agrégative qui inclut le respect des contraintes. Le détail en est donné dans la section de

Algorithme 3: Algorithme de recherche locale itérée

```
Sorties : Best
début
    S ← initialize(); // initialisation des structures de
        données
    count ← 0;
    tant que count < maxCount faire
        count ← count + 1;
        S ← constructSolution(S); // construction d'une
            solution non nécessairement admissible
        S ← improveSolution(S); // amélioration de la solution
            par recherche locale
        Best ← selectBest(S, Best);
    retourner Best;
```

définition du problème. Le classement de deux solutions est effectué par la procédure *selectBest*.

5.1.2 Boucle de construction

La boucle principale de construction *constructSolution* est détaillée dans l'Algorithme 4. Son rôle est de générer aussi rapidement que possible de nouvelles solutions candidates, admissibles ou non, de plus ou moins bonne qualité. La procédure répète *maxConstruct* fois un procédé de construction de base. La procédure *selectBest* effectue un classement suivant l'objectif global du problème.

5.1.3 Boucle d'amélioration

La génération de nouvelles solutions de départ diversifiées à partir de conditions initiales aléatoires est effectuée par la boucle de construction. À partir de la solution fournie, l'intensification de la recherche est réalisée par la procédure *improveSolution* de l'Algorithme 3 précédent. Elle permet des mouvements de faible ampleur dans une petite région de l'espace des solutions et tente de transformer des solutions non admissibles en solutions admissibles. Le schéma de base d'une procédure d'amélioration consiste à intégrer un opérateur de voisinage dans une stratégie de recherche. En

Algorithme 4: constructSolution

Entrées : $S \in$ Solution, $maxConstruct$
Sorties : $Best$
début
  $count \leftarrow 0$;
  tant que $count < maxConstruct$ **faire**
    $count \leftarrow count + 1$;
    `// Appliquer une opération de perturbation`
    $S \leftarrow perturbationOp(S)$;
    `// Construction séquentielle`
    $S \leftarrow constructSolutionSeq(S)$;
    $Best \leftarrow selectBest(S, Best)$;
  retourner $Best$;

suivant ce schéma, trois versions de recherche locales sont proposées : une recherche aléatoire simple et deux recherches locales à pivotage.

5.1.3.1 Recherche aléatoire

La stratégie de recherche définie par l'Algorithme 5 s'inspire du principe de la marche aléatoire. Nous l'avons dénommée recherche aléatoire itérée ou IRS. Elle consiste à faire évoluer une solution courante en effectuant un nombre donné de mouvements successifs dans le voisinage puis à sélectionner la meilleure solution rencontrée lors de cette succession de mouvements. La méthode est très simple car très peu d'opérations de copie de données sont effectuées à chaque pas d'itération.

5.1.3.2 Recherche locale

Dans cette section, nous présentons l'algorithme de recherche locale suivant deux versions. La première est la recherche locale gloutonne, que nous appelons « recherche locale itérée premier meilleur » , ou encore ILS-FI. La seconde est la recherche locale en profondeur, que nous appelons « recherche locale itérée meilleure amélioration » , ou encore ILS-BI. L'Algorithme 6 donne le code commun aux deux versions. La différence entre les deux exécutions tient dans la condition d'évaluation de la boucle interne « tant que » . Le code indiqué correspond à la version ILS-FI, retenue en définitive comme solution performante. L'instruction qui teste la détection d'une amélioration, dans la condition, doit être retirée dans le cas ILS-BI. Par rapport à IRS, ILS-FI et ILS-BI comportent chacune deux boucles imbriquées, et non une seule, comme le montre l'Algorithme 6.

La boucle externe contrôle la profondeur de la recherche. L'algorithme s'arrête lorsqu'aucune amélioration n'a été trouvée, ce qui correspond à l'atteinte d'un minimum local. La solution courante constitue l'élément pivot autour duquel s'effectue la recherche dans le voisinage. C'est à partir de cet élément que des solutions voisines sont générées et examinées. La boucle interne met en œuvre la règle de pivotage. Elle détermine la meilleure solution voisine vers laquelle la recherche doit se déplacer. Cette solution devient le nouveau pivot à partir duquel des solutions voisines sont à nouveau générées.

Par rapport à la recherche aléatoire itérée IRS, nous pouvons noter l'ajout d'une variable supplémentaire S' utilisée pour tester la présence d'une amélioration. Dans les deux types de recherche locale ILS-FI et ILS-BI, une variable supplémentaire de mémorisation est donc requise pour

Algorithme 5: iteratedRandomSearch

Entrées : $S \in$ Solution, $maxImprove$
Sorties : $Best$
début
  $count \leftarrow 0$;
  tant que $count < maxImprove$ **faire**
    $count \leftarrow count + 1$;
    $S \leftarrow generateNeighbor(S)$;
    $Best \leftarrow selectBest(S, Best)$;
  retourner $Best$;

Algorithme 6: Recherche locale *First Improvement*

Entrées : $S \in$ Solution
Sorties : $Best$
début
  $Best \leftarrow S$;
  $improvementFound \leftarrow true$;
  tant que $improvementFound$ `/* définit la profondeur */`
  faire
    $count \leftarrow 0$;
    $improvementFound \leftarrow false$;
    tant que $count < neighborhoodSize$ **et** $\neg$ *improvementFound*
    faire
      $count \leftarrow count + 1$;
      $S' \leftarrow generateNeighbor(S)$ `/* examen du voisin */`;
      Si $isBest(S', Best)$ $Best \leftarrow S'$;
      $improvementFound \leftarrow true$;
    $S \leftarrow Best$;
  retourner $Best$;

tester la possibilité d'amélioration, cela entraîne davantage de copies de données que dans la première méthode de recherche aléatoire itérée IRS.

Dans ILS-FI, la première meilleure solution rencontrée devient le nouvel élément pivot. Dans ILS-BI, c'est la meilleure solution dans un échantillon aléatoire du voisinage qui devient le nouvel élément pivot. La taille de l'échantillon est définie par le paramètre *neighborhoodSize* dans l'Algorithme 6. Dans les expérimentations, la taille de l'échantillon peut varier entre 100 et 1000 unités examinées.

5.2 Algorithme évolutionnaire et mémétique

5.2.1 Algorithme

Suivant la terminologie des algorithmes évolutionnaires, les solutions sont maintenant des « individus » composant une population qui va évoluer de génération en génération de manière à répondre aux exigences du problème. Des opérateurs de variations tels que des mutations vont modifier les individus, ceux-ci étant soumis à une sélection suivant leur adéquation aux

objectifs du problème.

Ici, nous spécifions une boucle principale type définissant un algorithme évolutionnaire. Nous pouvons décliner cette boucle principale selon un algorithme évolutionnaire classique, mais notre choix après expérimentation se porte sur un algorithme de type algorithme mémétique. Un algorithme mémétique est une extension d'un algorithme génétique, ou évolutionnaire, dans lequel est ajouté une recherche locale en tant qu'opérateur de modification de solution correspondant à une mutation d'un type particulier [2]. Nous n'utilisons pas d'opérateur de croisement. Nous utilisons nos méthodes de recherche locale et construction en tant qu'opérateurs de mutation dans la boucle évolutionnaire.

Le pseudo-code de la boucle évolutionnaire type est donné par l'Algorithme 7. Deux opérateurs de mutations sont spécifiés. Ceux sont les opérateurs *mutate* et *localSearch*. Deux opérateurs de sélection au niveau de la population peuvent être utilisés. Le premier d'entre eux est l'opérateur appelé *select* dans le pseudo-code. Il remplace $Pop/5$ individus de plus faible *fitness* dans la population par $Pop/5$ individus de plus grande *fitness*, avec Pop comme taille de la population d'individus. Le second opérateur de sélection et de classement est la version élitiste du premier. Il est appelé *selectElitist*. Il remplace $Pop/10$ individus ayant la *fitness* la plus faible de la population, par le meilleur individu $Best1$ rencontré durant l'exécution. En pratique, il s'avère que la sélection élitiste n'est pas efficace sur le problème. Celle-ci n'est donc pas utilisée en pratique sur le problème.

5.2.2 Version mémétique

Le détail des opérateurs de mutation pour l'algorithme mémétique est donné dans l'Algorithme 8. Dans ce cas, les opérations de mutation deviennent des appels aux procédures de construction et de recherche locale que nous avons spécifiées antérieurement. L'opérateur *mutate* effectue un appel de la procédure de construction itérée tandis que l'opérateur *localSearch* exécute un appel de la procédure de recherche locale.

L'intérêt recherché est de coupler la dynamique de diversi-fication et de sélection de l'approche évolutionnaire avec des recherches locales adaptées pour l'intensification de la recherche dans une zone réduite de l'espace des solutions. Le principe de l'algorithme évolutionnaire, via une population de solutions et des opérateurs de sélection, vise à augmenter la diversité des solutions potentielles. L'algorithme mémétique tire partie des avantages de la recherche locale et de la diversification au sein d'une population.

Algorithme 7: Boucle type d'un algorithme évolutionnaire

Sorties : $Best2$
début
  $Best1, Best2 \in$ Solution;
  P: Population;// 100 individus
  $Gen \in$ **Entier**;
  $Gen \leftarrow 0$;
  $P \leftarrow generate(P)$; // Générer les individus
  $Best1 \leftarrow getBest(P)$;
  // Répéter $MaxGen$ générations
  tant que $Gen < MaxGen$ **et** $\neg$ *(solution construite)* **faire**
    $Gen \leftarrow Gen + 1$;
    // Appliquer une opération/mutation de voisinage
    $P \leftarrow locaSearch(P)$;
    // Mémoriser le meilleur individu
    $Best1 \leftarrow getBest(P, Best1)$;
    $Best2 \leftarrow getBest(Best1, Best2)$;
    // Appliquer les opérateurs de sélection
    $P \leftarrow select(P, size(P)/5)$;
    // Appliquer une mutation
    $P \leftarrow mutate(P)$;
  retourner $Best2$;

Algorithme 8: Les opérateurs de variation de l'algorithme mémétique

```
Procédure generate(P ∈ Population)
début
    pour chaque individu I ∈ P faire
        I ← constructSolution(I) // Correspond à la
            construction spécifiée dans l'Algorithme (4)
Procédure mutate(P ∈ Population)
début
    pour chaque individu I ∈ P faire
        // Appliquer la mutation avec une probabilité de 0.5
        si rand(0, 1) > 0.9 alors
            I ← constructSolution(I) // Correspond à la
                construction spécifiée dans l'Algorithme (4)
Procédurelocalsearch(P ∈ Population)
début
    pour chaque individu I ∈ P faire
        I ← improveSolution(I) // Correspond à la recherche
            locale spécifiée à l'Algorithme (6)
```

Chapter 6

Jeux de tests et évaluation

L'évaluation des heuristiques nécessite l'utilisation d'instances de problème représentatives de la difficulté et de la diversité des cas réels d'application. Pour évaluer les méthodes d'optimisation, sont proposés un ensemble de jeux de tests, appelés benchmarks. Chaque dossier de test comporte des instances, issues de cas réels ou générées automatiquement, sur lesquelles sont appliqués des schémas de résolution suivant la configuration des paramètres choisis.

Un fichier de configuration type *config.cfg* du programme d'optimisation est fourni pour l'exécution des jeux de tests. Il définit une exécution standard avec l'ensemble des fonctionnalités d'optimisation activées. Il peut être utilisé avec toute instance sans exception, mais peut être modifié ou ajusté suivant la fonction ou l'option que l'on veut utiliser. Les variations des paramètres de configuration pour les différents tests de validation sont minimes par rapport a la configuration type. Nous décrivons les scénarios de tests proposés en indiquant dans chaque cas les fonctionnalités testées et les changements dans les paramètres de configuration de la configuration type.

Les tests de validation se décomposent en deux groupes, les tests avec génération d'instances automatique et les tests avec instances issues de cas réels ou instances spécifiques. Ils sont présentés ci-après avec des exemples de résultats. Noter qu'il est possible de lancer une exécution complète automatisée de tous les tests à l'aide de scripts systèmes d'exécution tels que la commande *./test/test_all.bat*.

Les dossiers de tests avec génération automatique d'instances sont les suivants :

- *test exemple type avec génération automatique*

Les dossiers de tests portant sur des instances issues de cas réels sont les suivants :

- *test exemple type*

6.1 Critères d'évaluation

Les valeurs des critères d'évaluation d'une solution sont renvoyées dans le fichier *output.stats*. Ils correspondent presque exactement aux objectifs du problème. Nous distinguons les objectifs des critères dans la mesure ou les premiers renvoient à la représentation interne de l'évaluation de la solution et les seconds renvoient aux valeurs présentées à l'utilisateur extérieur. Les critères correspondent aux objectifs, excepté qu'il peuvent être plus nombreux et complémentaires.

Les critères d'évaluation d'une solution renvoyés dans le fichier *output.stats* sont les suivants :

- *iteration* : numéro d'itération lors de l'évaluation
- *f_objectif_1* : premier objectif
- *f_objectif_2* : second objectif
- *f_objectif_3* : troisième objectif
- *duree*(s) : durée d'exécution en secondes
- *duree*(s.xx) : durée d'exécution en secondes et millisecondes

Noter que chaque répertoire de test comporte un fichier Excel pour l'analyse statistique des résultats de nom *result_analysis.xlsx*. Il est ainsi possible d'évaluer rapidement les performances moyennes de l'algorithme relativement à un grand nombre d'exécutions.

6.2 Test standard avec génération automatique

Il s'agit de tester des fonctionnalités sur des jeux de tests générés automatiquement à partir d'un modèle. Le test permet aussi d'illustrer la post-optimisation. Pour dissocier des fonctions, on utilise dans ce test un mode "post-optimisation" qui applique une nouvelle fonction, ou configuration,

sur la solution précédemment obtenue. Pour configurer l'application en mode post-optimisation, il suffit de spécifier les paramètres suivants :

- $constructFromScratchParam = false$
- $MAnbOfInternalConstructs = 0$

Le principe du test consiste à lancer une première phase d'optimisation avec des paramètres actifs et ensuite à lancer une post-optimisation en retirant l'option en question. L'ajout ou suppression des fonctions peut également être réalisé via les poids des objectifs. Les paramètres de configuration de départ sont les suivants :

- $utiliseFonction = true$

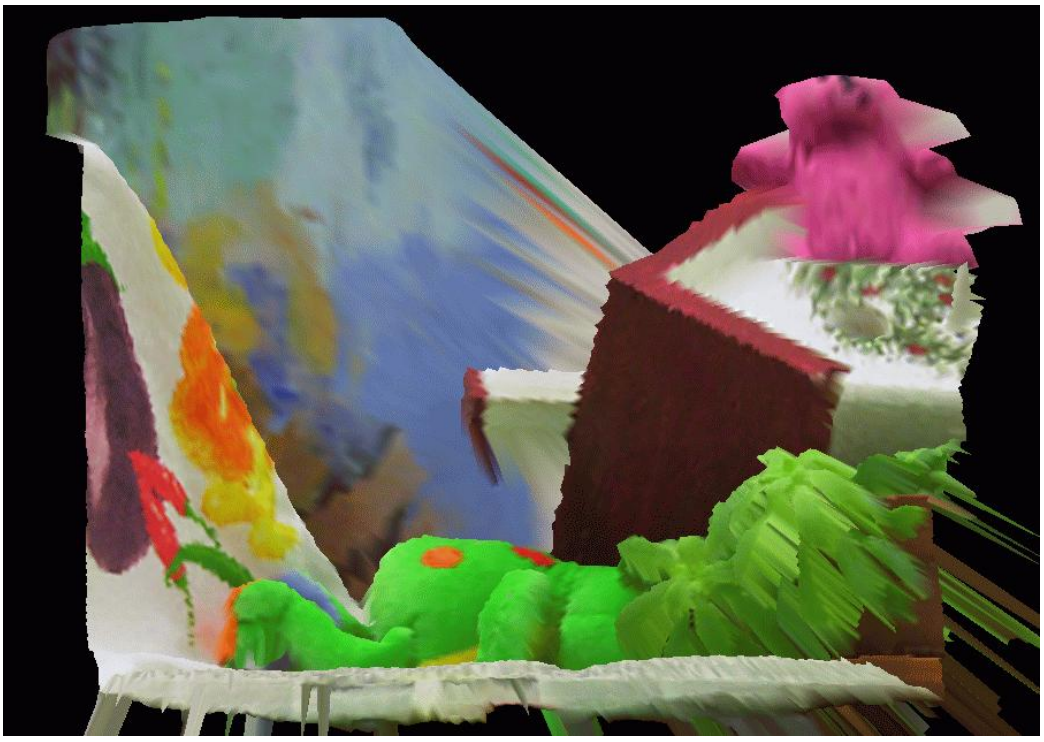

Figure 6.1: Solution type.

Un exemple de résultat est visualisé à la figure 6.1. On constate que les contraintes sont satisfaites. La solution est admissible sur l'ensemble des critères.

6.3 Test standard cas réel

Il s'agit d'un test issu de cas réels considéré difficile à résoudre. Il est résolu en moins de 0.1 secondes sur un PC Dual Core avec GPU. Dans ce cas difficile, pour accélérer au maximum la vitesse de résolution, il convient d'inhiber l'activation de certaines fonctions. Ces fonctions peuvent être inhibées avec les commandes :

- $utiliseFontion\_1 = false$
- $utiliseFonction\_2 = false$

Une solution est représentée à la figure 6.2.

Figure 6.2: Solution type.

Conclusion

L'objectif de ce document était de présenter les principes de la conception des algorithmes d'optimisation pour un problème type. Nous retenons en définitive deux approches d'optimisation efficaces sur le problème, la recherche locale *First Improvement*, et l'algorithme Mémétique.

Bibliography

[1] D.S. Johnson and L.A. McGeoch. The Traveling Salesman Problem: A Case Study in Local Optimization. In *Aarts, E.H.L. and Lenstra, J.K (eds) Local Search in Combinatorial Optimization*, pages 215–310. John Wiley and Sons, London, 1997. 8, 20

[2] P. Moscato and C. Cotta. A gentle introduction to memetic algorithms. In *Handbook of metaheuristics*, pages 105–144. Kluwer Academic Publishers, Boston MA, 2003. 10, 25

[3] W.M. Spears, K.A. De Jong, T. Back, D.B. Fogel, and H. de Garis. An overview of evolutionary computation. In *Machine Learning: ECML-93, Lecture notes in computer science*, volume 667, pages 442–459, 1993. 10